Design Patterns

Creational Design Patterns

Singleton Design Pattern

Following are some reasons which make sense to me for using Enum to implement Singleton pattern in Java. By the way If you like articles on design pattern than you can also check my post on Builder design pattern and Decorator design pattern .

This is by far biggest advantage, if you have been writing Singletons prior to Java 5 than you know that even with double checked locking you can have more than one instances. Though that issue is fixed with Java memory model improvement and guarantee provided by volatile variables from Java 5 onwards but it still tricky to write for many beginners. Compared to double checked locking with synchronization Enum singletons are cake walk. If you don't believe than just compare below code for conventional singleton with double checked locking and Enum

Singleton using Enum in Java

This is the way we generally declare Enum Singleton , it may contain instace variable and instance method but for sake of simplicity I haven't used any, just beware that if you are using any instance method than you need to ensure thread-safety of that method if at all it affect the state of object. By default creation of Enum instance is thread safe but any other method on Enum is programmers' responsibility.

```
/**
* Singleton pattern example using Java Enumj
*/

public enum EasySingleton{
    INSTANCE;
}
```

You can acess it by EasySingleton.INSTANCE, much easier than calling getInstance() method on Singleton.

Singleton example with double checked locking

Below code is an example of double checked locking in Singleton pattern, here getInstance() method checks two times to see whether INSTANCE is null or not and that's why it's called double checked locking pattern, remember that double checked locking is broker before Java 5 but with the guranteed of volatile variable in Java 5 memory model, it should work perfectly.

```
/**
* Singleton pattern example with Double checked Locking
*/

public class DoubleCheckedLockingSingleton{
     private volatile DoubleCheckedLockingSingleton INSTANCE;

     private DoubleCheckedLockingSingleton(){}

     public DoubleCheckedLockingSingleton getInstance(){
         if(INSTANCE == null){
            synchronized(DoubleCheckedLockingSingleton.class){
                //double checking Singleton instance
                if(INSTANCE == null){
                    INSTANCE = new DoubleCheckedLockingSingleton();
                }
            }
         }
         return INSTANCE;
     }
}
```

You can call DoubleCheckedLockingSingleton.getInstance() to get access of this Singleton class.

Now Just look at amount of code needed to create a lazy loaded thread-safe Singleton. With Enum Singleton pattern you can have that in one line because creation of Enum instance is thread-safe and guaranteed by JVM.

People may argue that there are better way to write Singleton instead of Double checked locking approach but every approach has there own advantages and disadvantages like I mostly prefer static field Singleton initialized during class loading as shown in below example, but keep in mind that is not a lazy loaded Singleton:

Singleton pattern with static factory method

This is one of my favorite methods to implement Singleton pattern in Java, Since Singleton instance is static and final variable it initialized when class is first loaded into memory so creation of instance is inherently thread-safe.

```
/**
* Singleton pattern example with static factory method
*/

public class Singleton{
    //initailzed during class loading
    private static final Singleton INSTANCE = new Singleton();

    //to prevent creating another instance of Singleton
    private Singleton(){}
```

```
    public static Singleton getSingleton(){
        return INSTANCE;
    }
}
```

You can call Singleton.getSingleton() to get access of this class.

2) Enum Singletons handled Serialization by themselves

Another problem with conventional Singletons are that once you implement serializable interface they are no longer remain Singleton because readObject() method always return a new instance just like constructor in Java. you can avoid that by using readResolve() method and discarding newly created instance by replacing with Singeton as shwon in below example :

```
    //readResolve to prevent another instance of Singleton
    private Object readResolve(){
        return INSTANCE;
    }
```

This can become even more complex if your Singleton Class maintain state, as you need to make them transient, but witn Enum Singleton, Serialization is guarnateed by JVM.

3) Creation of Enum instance is thread-safe

As stated in point 1 since creation of Enum instance is thread-safe by default you don't need to worry about double checked locking.

Read more: http://javarevisited.blogspot.com/2012/07/why-enum-singleton-are-better-in-java.html#ixzz4NTrXnx7A

What is Singleton class? Have you used Singleton before?

Singleton is a class which has only one instance in whole application and provides a `getInstance()` method to access the singleton instance. There are many classes in JDK which is implemented using Singleton pattern like `java.lang.Runtime` which provides `getRuntime()` method to get access of it and used to get free memory and total memory in Java.

Which classes are candidates of Singleton? Which kind of class do you make Singleton in Java?

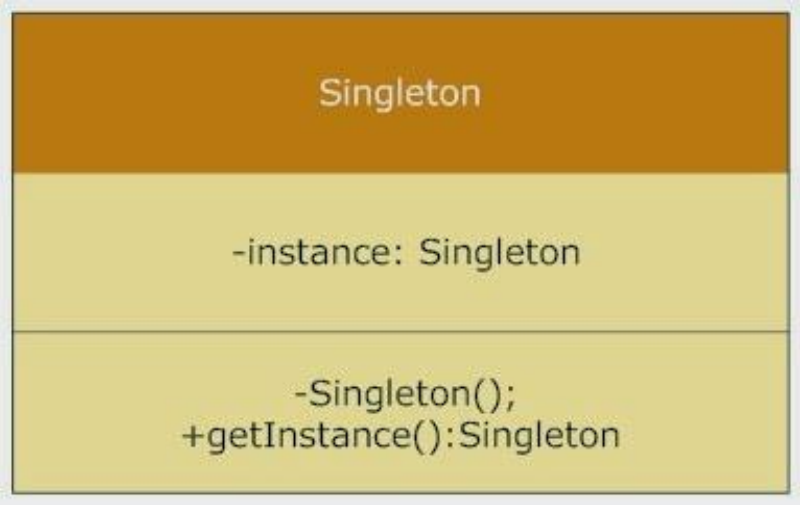
Here they check whether candidate has enough experience on usage of singleton or not. Does he is familiar of advantage/disadvantage or alternatives available for singleton in Java or not.

Answer : Any class which you want to be available to whole application and whole only one instance is viable is candidate of becoming Singleton. One example of this is `Runtime` class , since on whole java application only one runtime environment can be possible making `Runtime` Singleton is right decision. Another example is a utility classes like Popup in GUI application, if you want to show popup with message you can have one PopUp class on whole GUI application and anytime just get its instance, and call `show()` with message.

Can you write code for getInstance() method of a Singleton class in Java?

Most of the java programmer fail here if they have mugged up the singleton code because you can ask lots of follow-up question based upon the code they have written. I have seen many programmer write Singleton `getInstance()` method with double checked locking but they are not really familiar with the caveat associated with double checking of singleton prior to Java 5.

Answer : Until asked don't write code using double checked locking as it is more complex and chances of errors are more but if you have deep knowledge of double checked locking, volatile variable and lazy loading than this is your chance to shine. I have shared code examples of writing singleton classes using enum, using static factory and with double checked locking in my recent post Why Enum Singletons are better in Java, please see there.



Is it better to make whole getInstance() method synchronized or just critical section is enough? Which one you will prefer?
This is really nice question and I mostly asked to just quickly check whether candidate is aware of performance trade off of unnecessary locking or not. Since locking only make sense when we need to create instance and rest of the time its just read only access so locking of critical section is always better option. read more about synchronization on How Synchronization works in Java

Answer : This is again related to double checked locking pattern, well synchronization is costly and when you apply this on whole method than call to `getInstance()` will be synchronized and contented. Since synchronization is only needed during initialization on singleton instance, to prevent creating another instance of Singleton, It's better to only synchronize critical section and not whole method. Singleton pattern is also closely related to factory design pattern where `getInstance()` serves as static factory method.

What is lazy and early loading of Singleton and how will you implement it?
This is another great Singleton interview question in terms of understanding of concept of loading and cost associated with class loading in Java. Many of which I have interviewed not really familiar with this but its good to know concept.

Answer : As there are many ways to implement Singleton like using double checked locking or Singleton class with static final instance initialized during class loading. Former is called lazy loading because Singleton instance is created only when client calls `getInstance()` method while later is called early loading because Singleton instance is created when class is loaded into memory.

Give me some examples of Singleton pattern from Java Development Kit?
This is open question to all, please share which classes are Singleton in JDK. Answer to this question is `java.lang.Runtime`

Answer : There are many classes in Java Development Kit which is written using singleton pattern, here are few of them:

1. Java.lang.Runtime with getRuntime() method
2. Java.awt.Toolkit with getDefaultToolkit()
3. Java.awt.Desktop with getDesktop()

What is double checked locking in Singleton?
One of the most hyped question on Singleton pattern and really demands complete understanding to get it right because of Java Memory model caveat prior to Java 5. If a guy

comes up with a solution of using volatile keyword with Singleton instance and explains it then it really shows it has in depth knowledge of Java memory model and he is constantly updating his Java knowledge.

Answer : Double checked locking is a technique to prevent creating another instance of Singleton when call to `getInstance()` method is made in multi-threading environment. In Double checked locking pattern as shown in below example, singleton instance is checked two times before initialization. See here to learn more about double-checked-locking in Java.

```
public static Singleton getInstance(){
    if(_INSTANCE == null){
        synchronized(Singleton.class){
        //double checked locking - because second check of Singleton instance with lock
            if(_INSTANCE == null){
                _INSTANCE = new Singleton();
            }
        }
    }
    return _INSTANCE;
}
```

Double checked locking should only be used when you have requirement for lazy initialization otherwise use Enum to implement singleton or simple static final variable.

How do you prevent for creating another instance of Singleton using clone() method?
This type of questions generally comes some time by asking how to break singleton or when Singleton is not Singleton in Java.

Answer : Preferred way is not to implement Cloneable interface as why should one wants to create `clone()` of Singleton and if you do just throw Exception from `clone()` method as "Can not create clone of Singleton class".

How do you prevent for creating another instance of Singleton using reflection?
Open to all. In my opinion throwing exception from constructor is an option.
Answer: This is similar to previous interview question. Since constructor of Singleton class is supposed to be private it prevents creating instance of Singleton from outside but Reflection can access private fields and methods, which opens a threat of another instance. This can be avoided by throwing Exception from constructor as "Singleton already initialized"

How do you prevent for creating another instance of Singleton during serialization?

Another great question which requires knowledge of Serialization in Java and how to use it for persisting Singleton classes. This is open to you all but in my opinion use of readResolve() method can sort this out for you.

Answer: You can prevent this by using `readResolve()` method, since during serialization `readObject()` is used to create instance and it return new instance every time but by using readResolve you can replace it with original Singleton instance. I have shared code on how to do it in my post Enum as Singleton in Java. This is also one of the reason I have said that use Enum to create Singleton because serialization of enum is taken care by JVM and it provides guaranteed of that.

When is Singleton not a Singleton in Java?

There is a very good article present in Sun's Java site which discusses various scenarios when a Singleton is not really remains Singleton and multiple instance of Singleton is possible. Here is the link of that
article http://java.sun.com/developer/technicalArticles/Programming/singletons/

Apart from these questions on Singleton pattern, some of my reader contribute few more questions, which I included here. Thank you guys for your contribution.

Why you should avoid the singleton anti-pattern at all and replace it with DI?

Answer : Singleton Dependency Injection: every class that needs access to a singleton gets the object through its constructors or with a DI-container.

Why Singleton is Anti pattern

With more and more classes calling `getInstance()` the code gets more and more tightly coupled, monolithic, not testable and hard to change and hard to reuse because of not configurable, hidden dependencies. Also, there would be no need for this clumsy double checked locking if you call `getInstance` less often (i.e. once).

How many ways you can write Singleton Class in Java?

Answer : I know at least four ways to implement Singleton pattern in Java

1. Singleton by synchronizing `getInstance()` method
2. Singleton with public static final field initialized during class loading.
3. Singleton generated by static nested class, also referred as Singleton holder pattern.
4. From Java 5 on-wards using Enums

How to write thread-safe Singleton in Java?
Answer : Thread safe Singleton usually refers to write thread safe code which creates one and only one instance of Singleton if called by multiple thread at same time. There are many ways to achieve this like by using double checked locking technique as shown above and by using Enum or Singleton initialized by class loader.

At last few more questions for your practice, contributed by Mansi, Thank you Mansi

14) Singleton vs Static Class?
15) When to choose Singleton over Static Class?
16) Can you replace Singleton with Static Class in Java?
17) Difference between Singleton and Static Class in java?
18) Advantage of Singleton over Static Class?

Read more: http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html#ixzz4NVAxz7rl

Factory Method Design Pattern

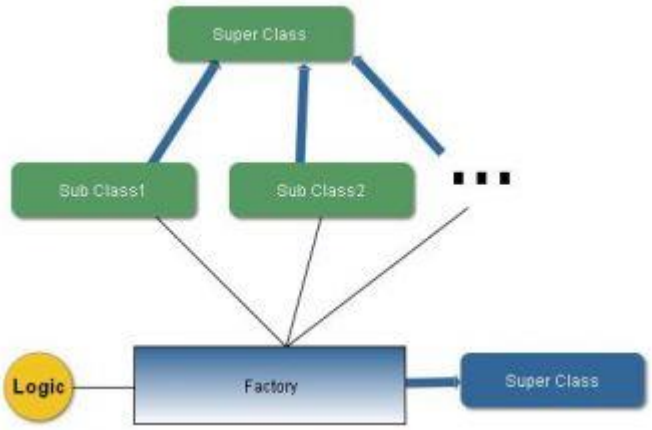
A factory method pattern is a creational pattern. It is used to instantiate an object from one among a set of classes based on a logic.

Assume that you have a set of classes which extends a common super class or interface. Now you will create a concrete class with a method which accepts one or more arguments. This method is our factory method. What it does is, based on the arguments passed factory method does logical operations and decides on which sub class to instantiate. This factory method will have the super class as its return type. So that, you can program for the interface and not for the implementation. This is all about factory method design pattern.

Sample factory method design pattern implementation in Java API

For a reference of how the factory method design pattern is implemented in Java, you can have a look at SAXParserFactory. It is a factory class which can be used to intantiate SAX based parsers to pares XML. The method newInstance is the factory method which instantiates the sax parsers based on some predefined logic.

Block diagram for The Design Pattern



Sample Java Source Code for Factory Method Design Pattern

Based on comments received from users, I try to keep my sample java source code as simple as possible for a novice to understand.

Base class:

```
package com.javapapers.sample.designpattern.factorymethod;

//super class that serves as type to be instantiated for factory method
pattern
public interface Pet {

 public String speak();

}
```

First subclass:

```
package com.javapapers.sample.designpattern.factorymethod;

//sub class 1 that might get instantiated by a factory method pattern
public class Dog implements Pet {

 public String speak() {
 return "Bark bark...";
 }
}
```

Second subclass:

```
package com.javapapers.sample.designpattern.factorymethod;

//sub class 2 that might get instantiated by a factory method pattern
public class Duck implements Pet {
 public String speak() {
 return "Quack quack...";
 }
}
```

Factory class:

```
package com.javapapers.sample.designpattern.factorymethod;

//Factory method pattern implementation that instantiates objects based on
logic
public class PetFactory {

 public Pet getPet(String petType) {
 Pet pet = null;

 // based on logic factory instantiates an object
 if ("bark".equals(petType))
 pet = new Dog();
 else if ("quack".equals(petType))
 pet = new Duck();
 return pet;
 }
}
```

Using the factory method to instantiate

```
package com.javapapers.sample.designpattern.factorymethod;

//using the factory method pattern
public class SampleFactoryMethod {

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

```
 //creating the factory
 PetFactory petFactory = new PetFactory();

 //factory instantiates an object
 Pet pet = petFactory.getPet("bark");

 //you don't know which object factory created
 System.out.println(pet.speak());
 }

}
```

Output of the above sample program for Factory Method Pattern

Bark bark

Abstract Factory Design Pattern

Factory of factories. To keep things simple you can understand it like, you have a set of 'related' factory method design pattern. Then you will put all those set of simple factories inside a factory pattern. So in turn you need not be aware of the final concrete class that will be instantiated. You can program for the interface using the top factory.

There is also a view that abstract factory is 'also' implemented using prototype instead of factory methords pattern. Beginners for now please don't yourself with that. Just go with factory methods pattern.

As there is a word 'abstract' in the pattern name don't mistake and confuse it with java 'abstract' keyword. It is not related to that. This abstract is from object oriented programming paradim.

Sample abstract factory design pattern implementation in Java API

XML API implements abstract factory. There is a class name SchemaFactory. This acts as a factory and supports implemenation of multiple schemas using abstract factory design pattern.

Sample Java Source Code for Factory Method Design Pattern

Following is the interface, that will be returned as the final end product from the factories.

```
package com.javapapers.sample.designpattern.abstractfactory;

public interface Animal {
        public void breathe();
}
```

Following is the interface for which the factory implementation should be done. Inturn all abstract factory will return this type.

```
package com.javapapers.sample.designpattern.abstractfactory;

public interface AnimalFactory {
	public Animal createAnimal();
}
```

One of the factory from a predefined set which will instantiate the above interface.

```
package com.javapapers.sample.designpattern.abstractfactory;

public class SeaFactory implements AnimalFactory {

	public Animal createAnimal() {
		return new Shark();
	}

}
```

Second factory from a predefined set which will instantiate the Animal interface.

```
package com.javapapers.sample.designpattern.abstractfactory;

public class LandFactory implements AnimalFactory {
	public Animal createAnimal() {
		return new Elephant();
	}
}
```

Implementation of an Animal. This class is grouped with the first abstract factory.

```
package com.javapapers.sample.designpattern.abstractfactory;

public class Shark implements Animal {
	public void breathe() {
		System.out.println("I breathe in water! He he!");
	}
}
```

Implementation of an Animal. This class is grouped with the second abstract factory.

```
package com.javapapers.sample.designpattern.abstractfactory;

public class Elephant implements Animal {
```

```
	public void breathe() {
		System.out.println("I breathe with my lungs. Its easy!");
	}
}
```

Following class consumes the abstract factory.

```
package com.javapapers.sample.designpattern.abstractfactory;

public class Wonderland {
	public Wonderland(AnimalFactory factory) {
		Animal animal = factory.createAnimal();
		animal.breathe();
	}
}
```

Testing the abstract factory design pattern.

```
package com.javapapers.sample.designpattern.abstractfactory;

public class SampleAbstractFactory {

	public static void main(String args[]){
		new Wonderland(createAnimalFactory("water"));
	}

	public static AnimalFactory createAnimalFactory(String type){
		if("water".equals(type))
			return new SeaFactory();
		else
			return new LandFactory();
	}
}
```

Output of the above sample program for abstract factory pattern

```
I breathe in water! He he!
```

Builder Design Pattern

Builder pattern is used to construct a complex object step by step and the final step will return the object. The process of constructing an object should be generic so that it can be used to create different representations of the same object.

Complex Object Construction

For example, you can consider construction of a home. Home is the final end product (object) that is to be returned as the output of the construction process. It will have many steps, like

basement construction, wall construction and so on roof construction. Finally the whole home object is returned. Here using the same process you can build houses with different properties.

GOF says,

"Separate the construction of a complex object from its representation so that the same construction process can create different representations" [GoF 94]

What is the difference between abstract factory and builder pattern?

Abstract factory may also be used to construct a complex object, then what is the difference with builder pattern? In builder pattern emphasis is on 'step by step'. Builder pattern will have many number of small steps. Those every steps will have small units of logic enclosed in it. There will also be a sequence involved. It will start from step 1 and will go on upto step n and the final step is returning the object. In these steps, every step will add some value in construction of the object. That is you can imagine that the object grows stage by stage. Builder will return the object in last step. But in abstract factory how complex the built object might be, it will not have step by step object construction.

Sample builder design pattern implementation in Java API

DocumentBuilderFactory , StringBuffer, StringBuilder are some examples of builder pattern usage in java API.

Sample Java Source Code for Builder Pattern

Following is the interface, that will be returned as the product from the builder.

```
package com.javapapers.sample.designpattern.builder;

public interface HousePlan {

        public void setBasement(String basement);

        public void setStructure(String structure);

        public void setRoof(String roof);

        public void setInterior(String interior);
}
```

Following is the interface for which the factory implementation should be done. Inturn all abstract factory will return this type.

```
package com.javapapers.sample.designpattern.abstractfactory;

public interface AnimalFactory {
        public Animal createAnimal();
```

```
}
```

Concrete class for the above interface. The builder constructs an implementation for the following class.

```
package com.javapapers.sample.designpattern.builder;

public class House implements HousePlan {

	private String basement;
	private String structure;
	private String roof;
	private String interior;

	public void setBasement(String basement) {
		this.basement = basement;
	}

	public void setStructure(String structure) {
		this.structure = structure;
	}

	public void setRoof(String roof) {
		this.roof = roof;
	}

	public void setInterior(String interior) {
		this.interior = interior;

	}

}
```

Builder interface. We will have multiple different implementation of this interface in order to facilitate, the same construction process to create different representations.

```
package com.javapapers.sample.designpattern.builder;

public interface HouseBuilder {

	public void buildBasement();

	public void buildStructure();

	public void bulidRoof();

	public void buildInterior();
```

```
	public House getHouse();
}
```

First implementation of a builder.

```
package com.javapapers.sample.designpattern.builder;

public class IglooHouseBuilder implements HouseBuilder {

	private House house;

	public IglooHouseBuilder() {
		this.house = new House();
	}

	public void buildBasement() {
		house.setBasement("Ice Bars");
	}

	public void buildStructure() {
		house.setStructure("Ice Blocks");
	}

	public void buildInterior() {
		house.setInterior("Ice Carvings");
	}

	public void bulidRoof() {
		house.setRoof("Ice Dome");
	}

	public House getHouse() {
		return this.house;
	}
}
```

Second implementation of a builder. Tipi is a type of eskimo house.

```
package com.javapapers.sample.designpattern.builder;

public class TipiHouseBuilder implements HouseBuilder {
	private House house;

	public TipiHouseBuilder() {
		this.house = new House();
	}

	public void buildBasement() {
```

```
                house.setBasement("Wooden Poles");
        }

        public void buildStructure() {
                house.setStructure("Wood and Ice");
        }

        public void buildInterior() {
                house.setInterior("Fire Wood");
        }

        public void bulidRoof() {
                house.setRoof("Wood, caribou and seal skins");
        }

        public House getHouse() {
                return this.house;
        }

}
```

Following class constructs the house and most importantly, this maintains the building sequence of object.

```
package com.javapapers.sample.designpattern.builder;

public class CivilEngineer {

        private HouseBuilder houseBuilder;

        public CivilEngineer(HouseBuilder houseBuilder){
                this.houseBuilder = houseBuilder;
        }

        public House getHouse() {
                return this.houseBuilder.getHouse();
        }

        public void constructHouse() {
                this.houseBuilder.buildBasement();
                this.houseBuilder.buildStructure();
                this.houseBuilder.bulidRoof();
                this.houseBuilder.buildInterior();
        }
}
```

Testing the sample builder design pattern.

```
package com.javapapers.sample.designpattern.builder;

public class BuilderSample {
	public static void main(String[] args) {
		HouseBuilder iglooBuilder = new IglooHouseBuilder();
		CivilEngineer engineer = new CivilEngineer(iglooBuilder);

		engineer.constructHouse();

		House house = engineer.getHouse();

		System.out.println("Builder constructed: "+house);
	}
}
```

Output of the above sample program for builder pattern

```
Builder constructed: com.javapapers.sample.designpattern.builder.
```

Prototype Design Pattern

When creating an object is time consuming and a costly affair and you already have a most similar object instance in hand, then you go for prototype pattern. Instead of going through a time consuming process to create a complex object, just copy the existing similar object and modify it according to your needs.

Its a simple and straight forward design pattern. Nothing much hidden beneath it. If you don't have much experience with enterprise grade huge application, you may not have experience in creating a complex / time consuming instance. All you might have done is use the new operator or inject and instantiate.

If you are a beginner you might be wondering, why all the fuss about prototye design pattern and do we really need this design pattern? Just ignore, all the big guys requires it. For you, just understand the pattern and sleep over it. You may require it one day in future.

Prototype pattern may look similar to builder design pattern. There is a huge difference to it. If you remember, "the same construction process can create different representations" is the key in builder pattern. But not in the case of prototype pattern.

So, how to implement the prototype design pattern? You just have to copy the existing instance in hand. When you say copy in java, immediately cloning comes into picture. Thats why when you read about prototype pattern, all the literature invariably refers java cloning.

Simple way is, clone the existing instance in hand and then make the required update to the cloned instance so that you will get the object you need. Other way is, tweak the cloning method itself to suit your new object creation need. Therefore whenever you clone that object you will directly get the new object of desire without modifying the created object explicitly.

The prototype design pattern mandates that the instance which you are going to copy should provide the copying feature. It should not be done by an external utility or provider.

But the above, other way comes with a caution. If somebody who is not aware of your tweaking the clone business logic uses it, he will be in issue. Since what he has in hand is not the exact clone. You can go for a custom method which calls the clone internally and then modifies it according to the need. Which will be a better approach.

Always remember while using clone to copy, whether you need a shallow copy or deep copy. Decide based on your business needs. If you need a deep copy, you can use serialization as a hack to get the deep copy done. Using clone to copy is entirey a design decision while implementing the prototype design pattern. Clone is not a mandatory choice for prototype pattern.

In prototype pattern, you should always make sure that you are well knowledgeable about the data of the object that is to be cloned. Also make sure that instance allows you to make changes to the data. If not, after cloning you will not be able to make required changes to get the new required object.

Following sample java source code demonstrates the prototype pattern. I have a basic bike in hand with four gears. When I want to make a different object, an advance bike with six gears I copy the existing instance. Then make necessary modifications to the copied instance. Thus the prototype pattern is implemented. Example source code is just to demonstrate the design pattern, please don't read too much out of it. I wanted to make things as simple as possible.

Sample Java Source Code for Prototype Design Pattern

```
package com.javapapers.sample.designpattern.prototype;

class Bike implements Cloneable {
        private int gears;
        private String bikeType;
        private String model;
        public Bike() {
                bikeType = "Standard";
                model = "Leopard";
                gears = 4;
        }

        public Bike clone() {
                return new Bike();
        }

        public void makeAdvanced() {
                bikeType = "Advanced";
```

```
                model = "Jaguar";
                gears = 6;
        }
        public String getModel(){
                return model;
        }
}

public class Workshop {
        public Bike makeJaguar(Bike basicBike) {
                basicBike.makeAdvanced();
                return basicBike;
        }
        public static void main(String args[]){
                Bike bike = new Bike();
                Bike basicBike = bike.clone();
                Workshop workShop = new Workshop();
                Bike advancedBike = workShop.makeJaguar(basicBike);
                System.out.println("Prototype Design Pattern:
"+advancedBike.getModel());
        }
}
```

Structural Design Patterns

Flyweight Design Pattern

Flyweight is used when there is a need to create high number of objects of almost similar nature. High number of objects consumes high memory and flyweight design pattern gives a solution to reduce the load on memory by sharing objects. It is achieved by segregating object properties into two types intrinsic and extrinsic. In this article lets see about this in detail with a real world example and respective java implementation.

Intent as stated by GoF is, “Use sharing to support large numbers of fine-grained objects efficiently”. Sharing is key in flyweight pattern and we need to judiciously decide if this pattern can be applied.

When to Use Flyweight Design Pattern

We need to consider following factors when choosing flyweight,

- Need to create large number of objects.
- Because of the large number when memory cost is a constraint.
- When most of the object attributes can be made external and shared.
- The application must not mandate unique objects, as after implementation same object will be used repeatedly.
- Its better when extrinsic state can be computed rather than stored. (explained below)

Flyweight is all about memory and sharing. Nowadays an average desktop comes with 500 GB hard disk, 4GB ram and with this you can stuff your whole home inside and will still have remaining space to put an elephant in it. Do we really need to bother about memory and usage? Since the cost has come down there is no restriction to use it effectively. Think about mobile devices that are increasing everyday and they still have memory constraint.

Even if you have huge memory, in some cases the application may need efficient use of it. For example assume we are working with an application that maps stars from universe. In this application if we are going to create an object for every star then think of it how much memory we will need. Gang of Four (GoF) have given an example of text editors in their book. If we create an object for every character in a file, think of it how many objects we will create for a long document. What will be the application performance.

How to Apply Flyweight

The object which we are going to create in high number should be analyzed before going for flyweight. Idea is to create lesser number of objects by reusing the same objects. Create smaller groups of objects and they should be reused by sharing. Closely look at objects properties and they can be segregated as two types intrinsic and extrinsic. Sharing is judged with respect to a context. Lets take the example of editors.

Consider a simple text editor where we can use only alphabet set A to Z. If we are going to create 100 page document using this editor we may have 200000 (2000 X 100) characters (assuming 2000 characters / page). Without flyweight we will create 200000 objects to have fine grained control. With such fine control, every character can have its own characteristics like color, font, size, etc. How do we apply flyweight here?

Intrinsic and Extrinsic State

Create only 26 objects for (A to Z) mapping every unique characters. These 26 objects will have intrinsic state as its character. That is object ‘a’ will have state as character ‘a’. Then what happens to color, font and size? Those are the extrinsic state and will be passed by client code. 26 objects will be in store, client code will get the needed character/object and pass the extrinsic state to it with respect to the context. With respect to context means, ‘a’ in first line may come in red color and the same character may come in blue color in different line.

Flyweight Implementation

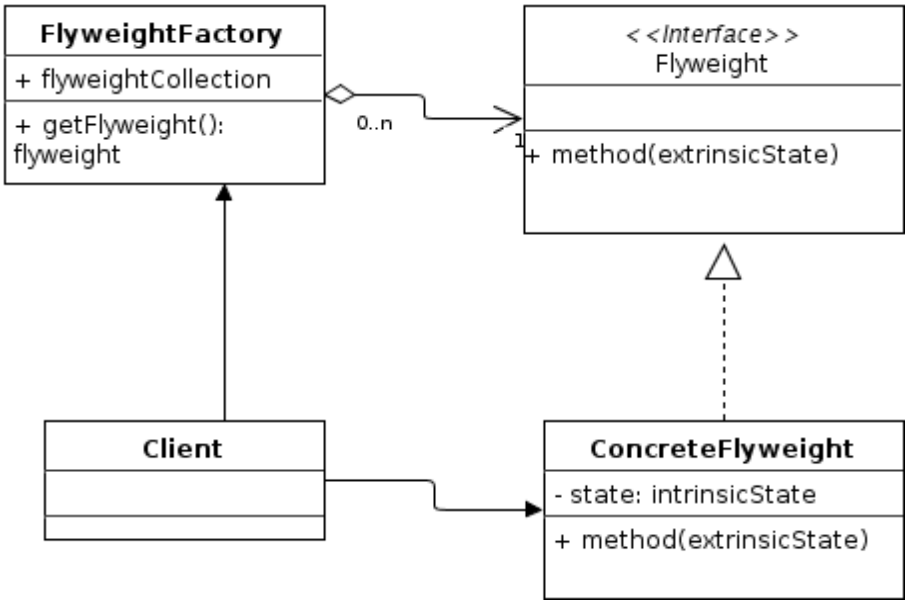
The object with intrinsic state is called flyweight object. When we implement flyweight we create concrete objects and have the intrinsic state stored in that. To create those concrete objects we will have factory and that is called Flyweight factory. This factory is to ensure that the objects are shared and we don't end up creating duplicate objects.

Let us take an example scenario of drawing. We need to draw different geometrical shapes like rectangles and ovals in huge number. Every shape may vary in colour, size, fill type, font used. For implementation sake lets limit our shapes to two rectangle and oval. Every shape will be accompanied by a label which directly maps it with the shape. That is all rectangles will have label as 'R' and all ovals will have label as 'O'.

Now our flyweight will have intrinsic state as label only. Therefore we will have only two flyweight objects. The varying properties colour, size, fill type and font will be extrinsic. We will have a flyweight factory that will maintain the two flyweight objects and distribute to client accordingly. There will be an interface for the flyweights to implement so that we will have a common blueprint and that is the flyweight interface.

Client code will use random number generators to create extrinsic properties. We are not storing the extrinsic properties anywhere, we will calculate on the fly and pass it. Use of random number generator is for convenience.

UML for Flyweight



Java Code Implementation for Flyweight

```
package com.javapapers.designpattern.flyweight;

import java.awt.Color;
```

```
import java.awt.Graphics;

public interface MyShape {
	public void draw(Graphics g, int x, int y, int width, int height,
			Color color, boolean fill, String font);
}
```

```
package com.javapapers.designpattern.flyweight;

import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;

public class MyOval implements MyShape {

	private String label;

	public MyOval(String label) {
		this.label = label;

	}

	public void draw(Graphics oval, int x, int y, int width, int height,
			Color color, boolean fill, String font) {
		oval.setColor(color);
		oval.drawOval(x, y, width, height);
		oval.setFont(new Font(font, 12, 12));
		oval.drawString(label, x + (width / 2), y);
		if (fill)
			oval.fillOval(x, y, width, height);
	}
}
```

```
package com.javapapers.designpattern.flyweight;

import java.awt.Color;
import java.awt.Font;
```

```
import java.awt.Graphics;

public class MyRectangle implements MyShape {

	private String label;

	public MyRectangle(String label) {
		this.label = label;

	}

	public void draw(Graphics rectangle, int x, int y, int width, int
height,
			Color color, boolean fill, String font) {
		rectangle.setColor(color);
		rectangle.drawRect(x, y, width, height);
		rectangle.setFont(new Font(font, 12, 12));
		rectangle.drawString(label, x + (width / 2), y);
		if (fill)
			rectangle.fillRect(x, y, width, height);
	}
}
```

```
package com.javapapers.designpattern.flyweight;

import java.util.HashMap;

public class ShapeFactory {

	private static final HashMap shapes = new HashMap();

	public static MyShape getShape(String label) {
		MyShape concreteShape = (MyShape) shapes.get(label);

		if (concreteShape == null) {
			if (label.equals("R")) {
				concreteShape = new MyRectangle(label);
			} else if (label.equals("O")) {
				concreteShape = new MyOval(label);
```

```
                        }
                        shapes.put(label, concreteShape);
                }
                return concreteShape;
        }
}
```

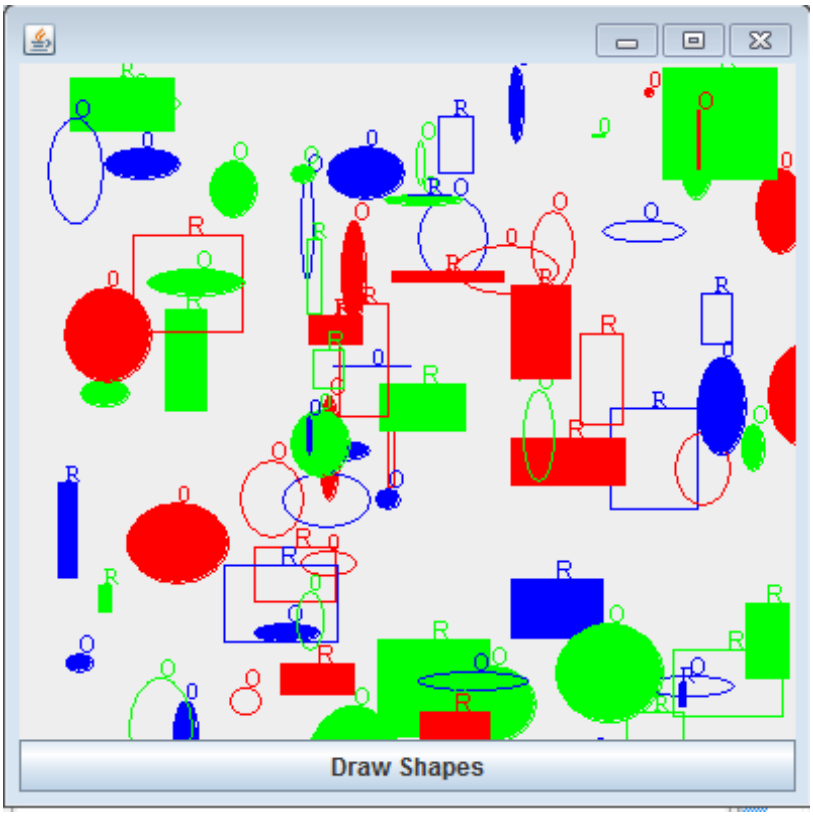
```
package com.javapapers.designpattern.flyweight;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Container;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Client extends JFrame {

        private static final int WIDTH = 400;
        private static final int HEIGHT = 400;

        private static final String shapes[] = { "R", "O" };
        private static final Color colors[] = { Color.red, Color.green,
Color.blue };
        private static final boolean fill[] = { true, false };
        private static final String font[] = { "Arial", "Courier" };

        public Client() {
                Container contentPane = getContentPane();

                JButton startButton = new JButton("Draw Shapes");
                final JPanel panel = new JPanel();

                contentPane.add(panel, BorderLayout.CENTER);
                contentPane.add(startButton, BorderLayout.SOUTH);
```

```
                setSize(WIDTH, WIDTH);
                setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                setVisible(true);

                startButton.addActionListener(new ActionListener() {
                        public void actionPerformed(ActionEvent event) {
                                Graphics g = panel.getGraphics();
                                for (int i = 0; i < 100; ++i) {
                                        MyShape shape =
ShapeFactory.getShape(getRandomShape());
                                        shape.draw(g, getRandomX(),
getRandomY(), getRandomWidth(),
                                                                getRandomHeight(),
getRandomColor(),
                                                                getRandomFill(),
getRandomFont());
                                }
                        }
                });
        }

        private String getRandomShape() {
                return shapes[(int) (Math.random() * shapes.length)];
        }

        private int getRandomX() {
                return (int) (Math.random() * WIDTH);
        }

        private int getRandomY() {
                return (int) (Math.random() * HEIGHT);
        }

        private int getRandomWidth() {
                return (int) (Math.random() * (WIDTH / 7));
        }

        private int getRandomHeight() {
                return (int) (Math.random() * (HEIGHT / 7));
        }
```

```
        private Color getRandomColor() {
                return colors[(int) (Math.random() * colors.length)];
        }

        private boolean getRandomFill() {
                return fill[(int) (Math.random() * fill.length)];
        }

        private String getRandomFont() {
                return font[(int) (Math.random() * font.length)];
        }

        public static void main(String[] args) {
                Client client = new Client();
        }
}
```

Download Flyweight Java Source Code

Flyweight Implementation Output



Use of Flyweight in JDK

java.lang.Integer#valueOf(int) (also on Boolean, Byte, Character, Short, Long, Float and Double). API doc says, "...this method is likely to yield significantly better space and time performance by caching frequently requested values....".

Design Pattern Related to Flyweight

Factory design pattern and singleton design pattern is used in implementing the flyweight.

Composite Design Pattern

When we want to represent part-whole hierarchy, use tree structure and compose objects. We know tree structure what a tree structure is and some of us don't know what a part-whole hierarchy is. A system consists of subsystems or components. Components can further be divided into smaller components. Further smaller components can be divided into smaller elements. This is a part-whole hierarchy.

Everything around us can be a candidate for part-whole hierarchy. Human body, a car, a computer, lego structure, etc. A car is made up of engine, tyre, … Engine is made up of electrical components, valves, … Electrical components is made up of chips, transistor, … Like this a component is part of a whole system. This hierarchy can be represented as a tree structure using composite design pattern.

"Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly." is the intent by GoF.

Real World Example

In this article, let us take a real world example of part-whole hierarchy and use composite design pattern using java. As a kid, I have spent huge amount of time with lego building blocks. Last week I bought my son an assorted kit lego and we spent the whole weekend together building structures.

Let us consider the game of building blocks to practice composite pattern. Assume that our kit has only three unique pieces (1, 2 and 4 blocks) and let us call these as primitive blocks as they will be the end nodes in the tree structure. Objective is to build a house and it will be a step by step process. First using primitive blocks, we should construct multiple windows, doors, walls, floor and let us call these structures. Then use all these structure to create a house.

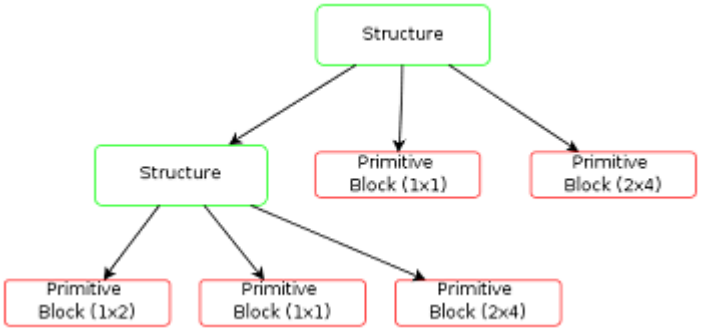
Primitive blocks combined together gives a structure. Multiple structures assembled together gives a house.

Important Points

- Importance of composite pattern is, the group of objects should be treated similarly as a single object.
- Manipulating a single object should be as similar to manipulating a group of objects. In sync with our example, we join primitive blocks to create structures and similarly join structures to create house.
- Recursive formation and tree structure for composite should be noted.
- Clients access the whole hierarchy through the components and they are not aware about if they are dealing with leaf or composites.

Tree for Composite

When we get a recursive structure the obvious choice for implementation is a tree. In composite design pattern, the part-whole hierarchy can be represented as a tree. Leaves (end nodes) of a tree being the primitive elements and the tree being the composite structure.



Uml Design for Composite Pattern

Component: (structure)

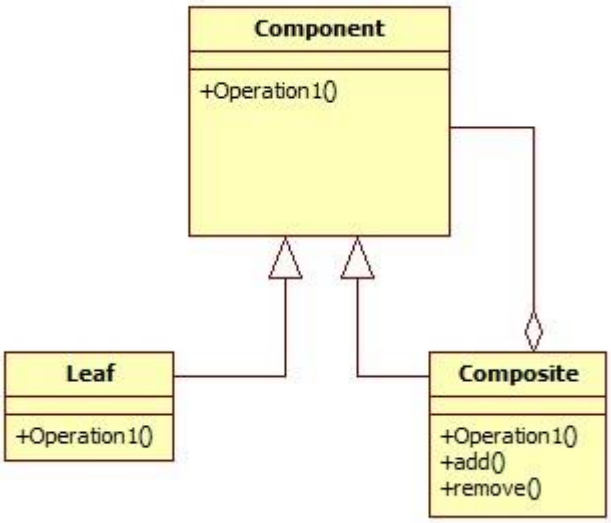
1. Component is at the top of hierarchy. It is an abstraction for the composite.
2. It declares the interface for objects in composition.
3. (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.

Leaf: (primitive blocks)

1. The end nodes of the tree and will not have any child.
2. Defines the behaviour for single objects in the composition

Composite: (group)

1. Consists of child components and defines behaviour for them
2. Implements the child related operations.



Composite Pattern Implementation

```
package com.javapapers.designpattern.composite;

public class Block implements Group {

    public void assemble() {
        System.out.println("Block");
    }
}
package com.javapapers.designpattern.composite;

public interface Group {
    public void assemble();
}
package com.javapapers.designpattern.composite;

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

public class Structure implements Group {
        // Collection of child groups.
        private List groups = new ArrayList();

        public void assemble() {
                for (Group group : groups) {
                        group.assemble();
```

```
                }
        }

        // Adds the group to the structure.
        public void add(Group group) {
                groups.add(group);
        }

        // Removes the group from the structure.
        public void remove(Group group) {
                groups.remove(group);
        }
}
package com.javapapers.designpattern.composite;

public class ImplementComposite {
         public static void main(String[] args) {
                //Initialize three blocks
                Block block1 = new Block();
                Block block2 = new Block();
                Block block3 = new Block();

                //Initialize three structure
                Structure structure = new Structure();
                Structure structure1 = new Structure();
                Structure structure2 = new Structure();

                //Composes the groups
                structure1.add(block1);
                structure1.add(block2);

                structure2.add(block3);

                structure.add(structure1);
                structure.add(structure2);

                structure.assemble();
            }
}
```

Usage of Composite Design Pattern in Sun/Oracle JDK

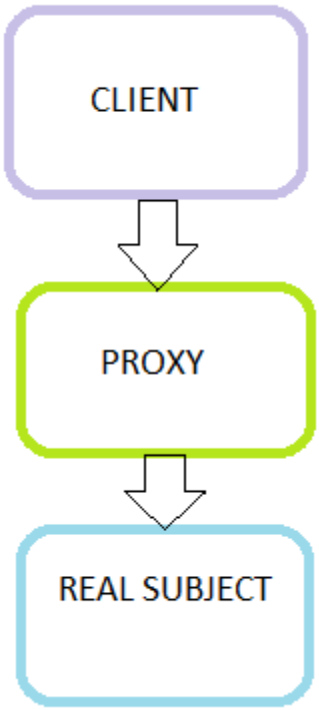
- java.awt.Container#add(Component) – in awt, we have containers and components – a classic implementation
- javax.faces.component.UIComponent#getChildren()

Proxy Design Pattern

"Provide a surrogate or placeholder for another object to control access to it" is the intent provided by GoF.

Proxy means 'in place of'. In attendance roll call, we give proxy for our friends in college right? 'Representing' or 'in place of' or 'on behalf of' are literal meanings of proxy and that directly explains proxy design pattern. It is one of the simplest and straight forward design pattern.

Proxy design pattern gets second rank in popularity in interviews. Guess who gets the first rank? none other than singleton design pattern.



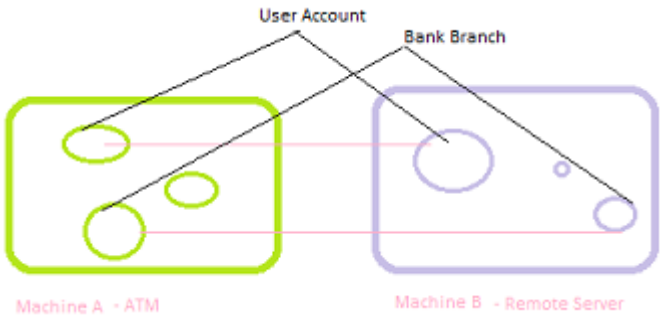
Possible Usage Scenarios

- Remote Proxy – Represents an object locally which belongs to a different address space. Think of an ATM implementation, it will hold proxy objects for bank information that exists in the remote server. RMI is an example of proxy implmenetation for this type in java.

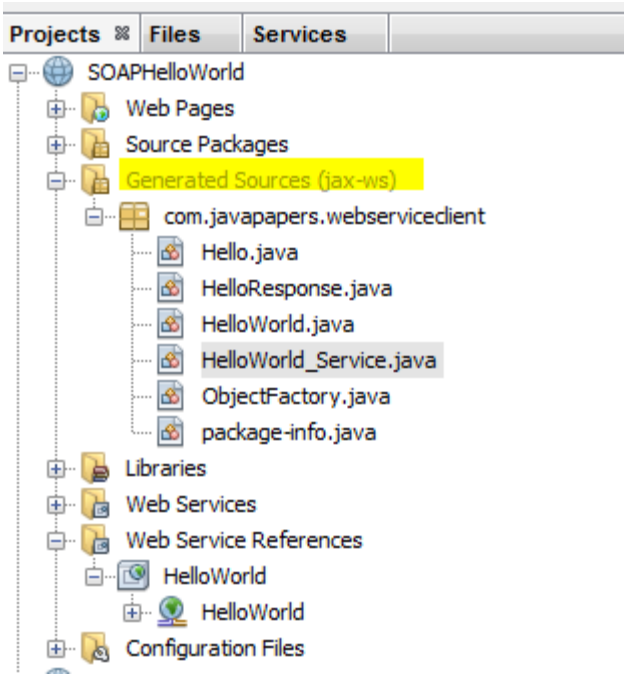
- Virtual Proxy – In place of a complex or heavy object use a skeleton representation. When an underlying image is huge in size, just represent it using a virtual proxy object and on demand load the real object. You feel that the real object is expensive in terms of instantiation and so without the real need we are not going to use the real object. Until the need arises we will use the virtual proxy.
- Protection Proxy – Are you working on a MNC? If so, you might be well aware of the proxy server that provides you internet. Saying more than provides, the right word is censores internet. The management feels its better to censor some content and provide only work related web pages. Proxy server does that job. This is a type of proxy design pattern. Lighter part of censor is, we search for something critical in Google and click the result and you get this page is blocked by proxy server. You never know why this page is blocked and you feel this is genuine. How do you overcome that, over a period you learn to live with it.
- Smart Reference – Just we keep a link/reference to the real object a kind of pointer.

Proxy Design Pattern Example

Remote Proxy:



Sometime back I wrote an article on A helloworld for Soap Web Service. A part of it contains implementation of proxy design pattern. The client has the stub files generated which acts as a proxy for the classes in server side.



Java's Support for Proxy Design Pattern

From JDK 1.3 java has direct support for implementing proxy design pattern. We need not worry on mainting the reference and object creation. Java provides us the needed utilities. Following example implementation explains on how to use java's api for proxy design pattern.

```
package com.javapapers.designpattern.proxy;

public interface Animal {

	public void getSound();

}
package com.javapapers.designpattern.proxy;

public class Lion implements Animal {

	public void getSound() {
		System.out.println("Roar");
	}

}
package com.javapapers.designpattern.proxy;

import java.lang.reflect.InvocationHandler;
```

```
import java.lang.reflect.Method;

public class AnimalInvocationHandler implements InvocationHandler {
        public AnimalInvocationHandler(Object realSubject) {
                this.realSubject = realSubject;
        }

        public Object invoke(Object proxy, Method m, Object[] args) {
                Object result = null;
                try {
                        result = m.invoke(realSubject, args);
                } catch (Exception ex) {
                        ex.printStackTrace();
                }
                return result;
        }

        private Object realSubject = null;
}
package com.javapapers.designpattern.proxy;

import java.lang.reflect.Proxy;

public class ProxyExample {

        public static void main(String[] args) {
                Animal realSubject = new Lion();
                Animal proxy = (Animal)
Proxy.newProxyInstance(realSubject.getClass()
                                .getClassLoader(),
realSubject.getClass().getInterfaces(),
                                new AnimalInvocationHandler(realSubject));
                proxy.getSound();
        }
}
```

Download Proxy Example Source

Important Points

- A proxy may hide information about the real object to the client.
- A proxy may perform optimization like on demand loading.

- A proxy may do additional house-keeping job like audit tasks.
- Proxy design pattern is also known as surrogate design pattern.

Proxy Design Pattern Usage in Java API

- java.rmi.* – RMI package is based on proxy design pattern

Adapter vs Proxy Design Pattern

Adapter design pattern provides a different interface from the real object and enables the client to use it to interact with the real object. But, proxy design pattern provides the same interface as in the real object.

Decorator vs Proxy Design Patter

Decorator design pattern adds behaviour at runtime to the real object. But, Proxy does not change the behaviour instead it controls the behaviour.

This Design Patterns tutorial was added on 01/04/2012.

Bridge Design Pattern

"Decouple an abstraction from its implementation so that the two can vary independently" is the intent for bridge design pattern as stated by GoF.

Bridge design pattern is a modified version of the notion of "prefer composition over inheritance".

Problem and Need for Bridge Design Pattern

When there are inheritance hierarchies creating concrete implementation, you loose flexibility because of interdependence. Oops! these kind of sentencies shows that the author(I) didn't understand and tries to escape! Okay, I will decrypt this sentence in the coming paragraphs.

Decouple implentation from interface and hiding implementation details from client is the essense of bridge design pattern.

Elements of Bridge Design Pattern

- Abstraction – core of the bridge design pattern and defines the crux. Contains a reference to the implementer.
- Refined Abstraction – Extends the abstraction takes the finer detail one level below. Hides the finer elements from implemetors.
- Implementer – This interface is the higer level than abstraction. Just defines the basic operations.
- Concrete Implementation – Implements the above implementer by providing concrete implementation.

Example for core elements of Bridge Design Pattern

Vehicle -> Abstraction
manufacture()

Car -> Refined Abstraction 1
manufacture()

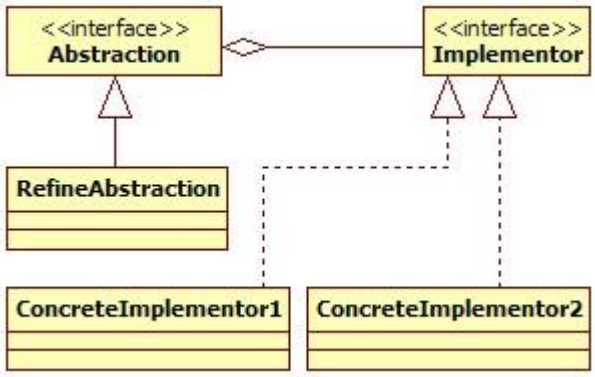
Bike -> Refined Abstraction 2
manufacture()

Workshop -> Implementor
work()

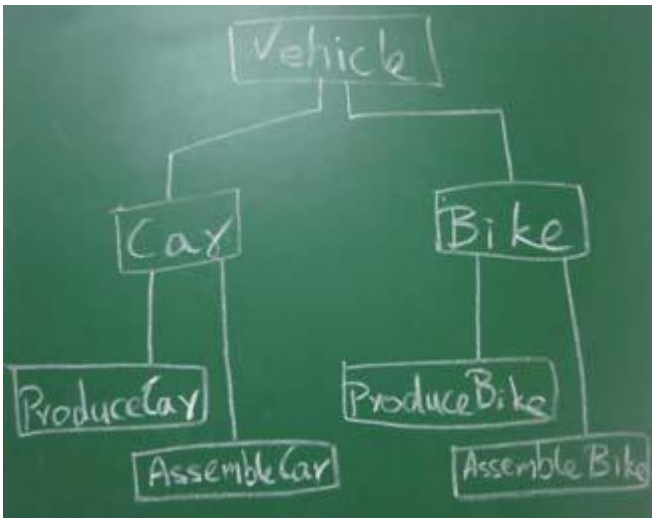
Produce -> Concrete Implementation 1
work()

Assemble -> Concrete Implementation 2
work()

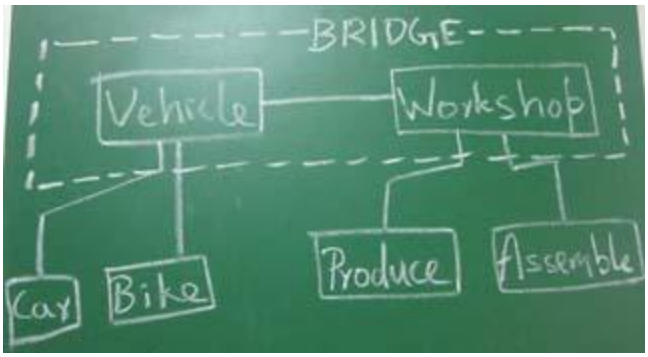
Generic UML Diagram for Bridge Design Pattern



Before Bridge Design Pattern



After Bridge Design Pattern



Sample Java Code for Bridge Design Pattern

```
package com.javapapers.designpattern;

/**
 * abstraction in bridge pattern
 * */
abstract class Vehicle {
        protected Workshop workShop1;
        protected Workshop workShop2;

        protected Vehicle(Workshop workShop1, Workshop workShop2) {
                this.workShop1 = workShop1;
                this.workShop2 = workShop2;
```

```
        }

        abstract public void manufacture();
}

package com.javapapers.designpattern;

/**
 * Refine abstraction 1 in bridge pattern
 */
public class Car extends Vehicle {

        public Car(Workshop workShop1, Workshop workShop2) {
                super(workShop1, workShop2);
        }

        @Override
        public void manufacture() {
                System.out.print("Car ");
                workShop1.work();
                workShop2.work();

        }

}

package com.javapapers.designpattern;

/**
 * Refine abstraction 2 in bridge pattern
 */
public class Bike extends Vehicle {

        public Bike(Workshop workShop1, Workshop workShop2) {
                super(workShop1, workShop2);
        }

        @Override
        public void manufacture() {
                System.out.print("Bike ");
                workShop1.work();
```

```
                workShop2.work();
        }

}
```

```
package com.javapapers.designpattern;

/**
 * Implementor for bridge pattern
 * */
public interface Workshop {
        abstract public void work();
}
```

```
package com.javapapers.designpattern;

/**
 * Concrete implementation 1 for bridge pattern
 * */
public class Produce implements Workshop {

        @Override
        public void work() {
                System.out.print("Produced");
        }

}
```

```
package com.javapapers.designpattern;

/**
 * Concrete implementation 2 for bridge pattern
 * */
public class Assemble implements Workshop {

        @Override
        public void work() {
                System.out.println(" Assembled.");
        }

}
```

```
package com.javapapers.designpattern;

/*
 * Demonstration of bridge design pattern
 */
public class BridgePattern {

        public static void main(String[] args) {

                Vehicle vehicle1 = new Car(new Produce(), new Assemble());
                vehicle1.manufacture();
                Vehicle vehicle2 = new Bike(new Produce(), new Assemble());
                vehicle2.manufacture();

        }
}

Output:
Car Produced Assembled.
Bike Produced Assembled.
```

Summary of Bridge Design Pattern

- Creates two different hierarchies. One for abstraction and another for implementation.
- Avoids permanent binding by removing the dependency between abstraction and implementation.
- We create a bridge that coordinates between abstraction and implementation.
- Abstraction and implementation can be extended separately.
- Should be used when we have need to switch implementation at runtime.
- Client should not be impacted if there is modification in implementation of abstraction.
- Best used when you have multiple implementations.

Bridge Vs Adapter Design Pattern

The adapter design pattern helps it two incompatible classes to work together. But, bridge design pattern decouples the abstraction and implementation by creating two different hierarchies.

Adapter Design Pattern

An adapter helps two incompatible interfaces to work together. This is the real world definition for an adapter. Adapter design pattern is used when you want two different classes with

incompatible interfaces to work together. The name says it all. Interfaces may be incompatible but the inner functionality should suit the need.

In real world the easy and simple example that comes to mind for an adapter is the travel power adapter. American socket and plug are different from British. Their interface are not compatible with one another. British plugs are cylindrical and American plugs are recangularish. You can use an adapter in between to fit an American (rectangular) plug in British (cylindrical) socket assuming voltage requirements are met with.

How to implement adapter design pattern?

Adapter design pattern can be implemented in two ways. One using the inheritance method and second using the composition method. Just the implementation methodology is different but the purpose and solution is same.

Adapter implementation using inheritance

When a class with incompatible method needs to be used with another class you can use inheritance to create an adapter class. The adapter class which is inherited will have new compatible methods. Using those new methods from the adapter the core function of the base class will be accessed. This is called “is-a” relationship. The same real world example is implemented using java as below. Dont worry too much about logic, following example source code attempts to explain adapter design pattern and the goal is simplicity.

```
public class CylindricalSocket {
        public String supply(String cylinStem1, String cylinStem1) {
                System.out.println("Power power power...");
        }
}

public class RectangularAdapter extends CylindricalSocket {
        public String adapt(String rectaStem1, Sting rectaStem2) {
```

```
                //some conversion logic
                String cylinStem1 = rectaStem1;
                String cylinStem2 = rectaStem2;
                return supply(cylinStem1, cylinStem2);
        }
}

public class RectangularPlug {
        private String rectaStem1;
        private String rectaStem2;
        public getPower() {
                RectangulrAdapter adapter = new RectangulrAdapter();
                String power = adapter.adapt(rectaStem1, rectaStem2);
                System.out.println(power);
        }
}
```

Adapter implementation using composition

The above implementation can also be done using composition. Instead of inheriting the base class create adapter by having the base class as attribute inside the adapter. You can access all the methods by having it as an attribute. This is nothing but "has-a" relationship. Following example illustrates this approach. Difference is only in the adapter class and other two classes are same. In most scenarios, prefer composition over inheritance. Using composition you can change the behaviour of class easily if needed. It enables the usage of tools like dependency injection.

```
public class CylindricalSocket {
        public String supply(String cylinStem1, String cylinStem1) {
                System.out.println("Power power power...");
        }
}

public class RectangularAdapter {
        private CylindricalSocket socket;

        public String adapt(String rectaStem1, Sting rectaStem2) {
                //some conversion logic
                socket = new CylindricalSocket();
                String cylinStem1 = rectaStem1;
                String cylinStem2 = rectaStem2;
                return socket.supply(cylinStem1, cylinStem2);
```

```
        }
}

public class RectangularPlug {
        private String rectaStem1;
        private String rectaStem2;
        public getPower() {
                RectangulrAdapter adapter = new RectangulrAdapter();
                String power = adapter.adapt(rectaStem1, rectaStem2);
                System.out.println(power);
        }
}
```

Adapter design pattern in java API

java.io.InputStreamReader(InputStream)
java.io.OutputStreamWriter(OutputStream)

Decorator Design Pattern

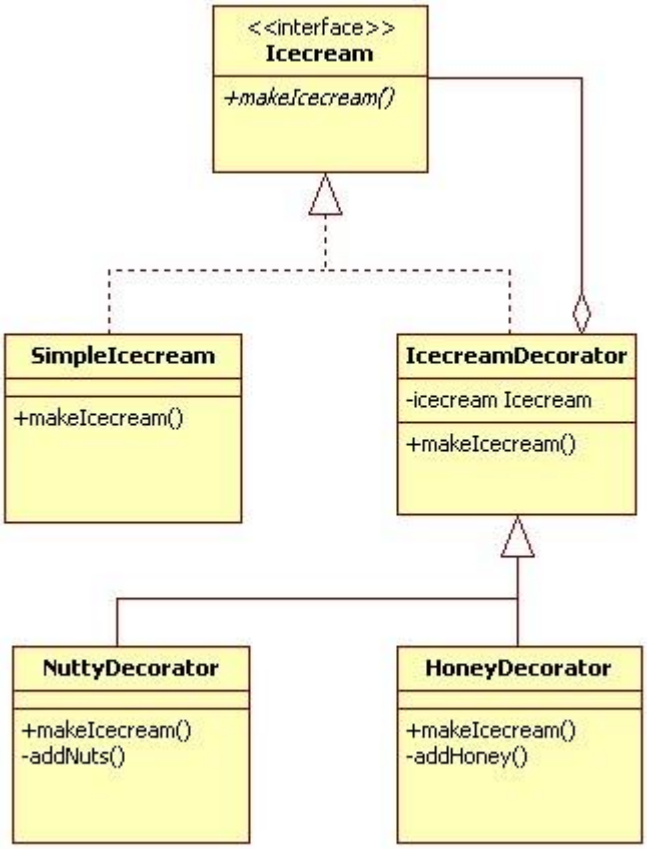
To extend or modify the behaviour of ‘an instance’ at runtime decorator design pattern is used. Inheritance is used to extend the abilities of ‘a class’. Unlike inheritance, you can choose any single object of a class and modify its behaviour leaving the other instances unmodified.

In implementing the decorator pattern you construct a wrapper around an object by extending its behavior. The wrapper will do its job before or after and delegate the call to the wrapped instance.

Design of decorator pattern

You start with an interface which creates a blue print for the class which will have decorators. Then implement that interface with basic functionalities. Till now we have got an interface and an implementation concrete class. Create an abstract class that contains (aggregation relationship) an attribute type of the interface. The constructor of this class assigns the interface type instance to that attribute. This class is the decorator base class. Now you can extend this class and create as many concrete decorator classes. The concrete decorator class will add its own methods. After / before executing its own method the concrete decorator will call the base instance's method. Key to this decorator design pattern is the binding of method and the base instance happens at runtime based on the object passed as parameter to the constructor. Thus dynamically customizing the behavior of that specific instance alone.

Decorator Design Pattern – UML Diagram



Implementation of decorator pattern

Following given example is an implementation of decorator design pattern. Icecream is a classic example for decorator design pattern. You create a basic icecream and then add toppings to it as you prefer. The added toppings change the taste of the basic icecream. You can add as many topping as you want. This sample scenario is implemented below.

```
package com.javapapers.sample.designpattern;



public interface Icecream {

  public String makeIcecream();

}
```

The above is an interface depicting an icecream. I have kept things as simple as possible so that the focus will be on understanding the design pattern. Following class is a concrete implementation of this interface. This is the base class on which the decorators will be added.

```
package com.javapapers.sample.designpattern;

public class SimpleIcecream implements Icecream {

  @Override
  public String makeIcecream() {
    return "Base Icecream";
  }

}
```

Following class is the decorator class. It is the core of the decorator design pattern. It contains an attribute for the type of interface. Instance is assigned dynamically at the creation of decorator using its constructor. Once assigned that instance method will be invoked.

```
package com.javapapers.sample.designpattern;

abstract class IcecreamDecorator implements Icecream {

  protected Icecream specialIcecream;

  public IcecreamDecorator(Icecream specialIcecream) {
    this.specialIcecream = specialIcecream;
  }

  public String makeIcecream() {
    return specialIcecream.makeIcecream();
  }
```

```
}
```

Following two classes are similar. These are two decorators, concrete class implementing the abstract decorator. When the decorator is created the base instance is passed using the constructor and is assigned to the super class. In the makeIcecream method we call the base method followed by its own method addNuts(). This addNuts() extends the behavior by adding its own steps.

```
package com.javapapers.sample.designpattern;

public class NuttyDecorator extends IcecreamDecorator {

  public NuttyDecorator(Icecream specialIcecream) {
    super(specialIcecream);
  }

  public String makeIcecream() {
    return specialIcecream.makeIcecream() + addNuts();
  }

  private String addNuts() {
    return " + cruncy nuts";
  }
}
package com.javapapers.sample.designpattern;

public class HoneyDecorator extends IcecreamDecorator {

  public HoneyDecorator(Icecream specialIcecream) {
    super(specialIcecream);
  }

  public String makeIcecream() {
    return specialIcecream.makeIcecream() + addHoney();
  }

  private String addHoney() {
    return " + sweet honey";
  }
}
```

Execution of the decorator pattern

I have created a simple icecream and decorated that with nuts and on top of it with honey. We can use as many decorators in any order we want. This excellent flexibility and changing the behaviour of an instance of our choice at runtime is the main advantage of the decorator design pattern.

```
package com.javapapers.sample.designpattern;

public class TestDecorator {

  public static void main(String args[]) {
    Icecream icecream = new HoneyDecorator(new NuttyDecorator(new
SimpleIcecream()));
    System.out.println(icecream.makeIcecream());
  }

}
```

Output

```
Base Icecream + cruncy nuts + sweet honey
```

Decorator Design Pattern in java API

java.io.BufferedReader;
java.io.FileReader;
java.io.Reader;

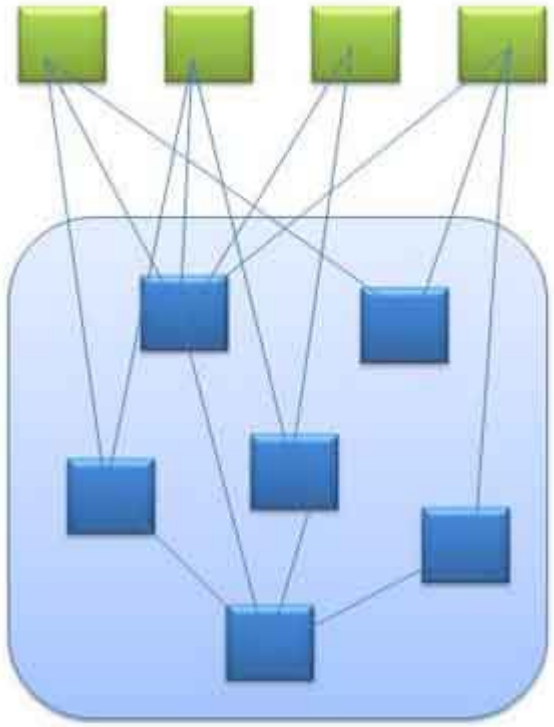
The above readers of java API are designed using decorator design pattern.

Facade Design Pattern

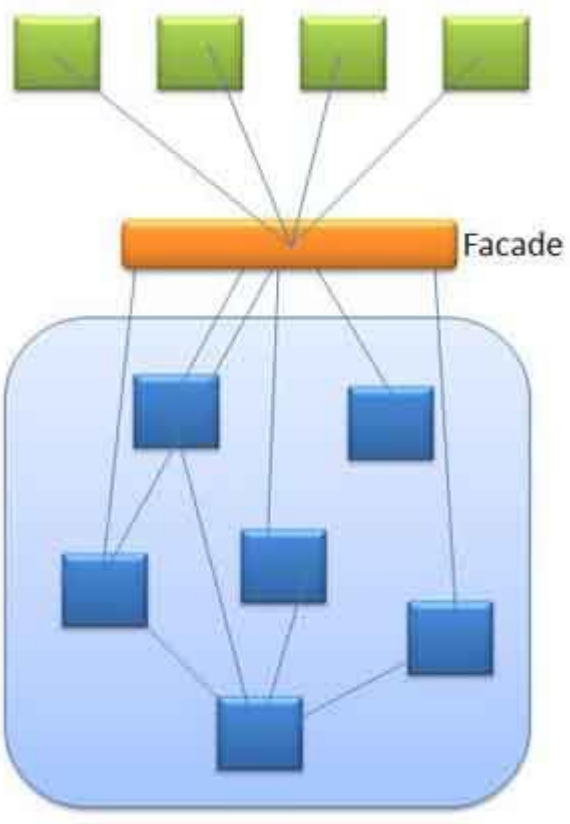
GoF definition for facade design pattern is, “Provide a unified interface to a set of interfaces in a subsystem. Facade Pattern defines a higher-level interface that makes the subsystem easier to use.”

How do we infer the above definition? Think of a component that solves a complex business problem. That component may expose lot of interfaces to interact with it. To complete a process flow we may have to interact with multiple interfaces.

To simplify that interaction process, we introduce facade layer. Facade exposes a simplified interface (in this case a single interface to perform that multi-step process) and internally it interacts with those components and gets the job done for you. It can be taken as one level of abstraction over an existing layer.



Facade design pattern is one among the other design patterns that promote loose coupling. It emphasizes one more important aspect of design which is abstraction. By hiding the complexity behind it and exposing a simple interface it achieves abstraction.



Real World Examples for Facade Pattern

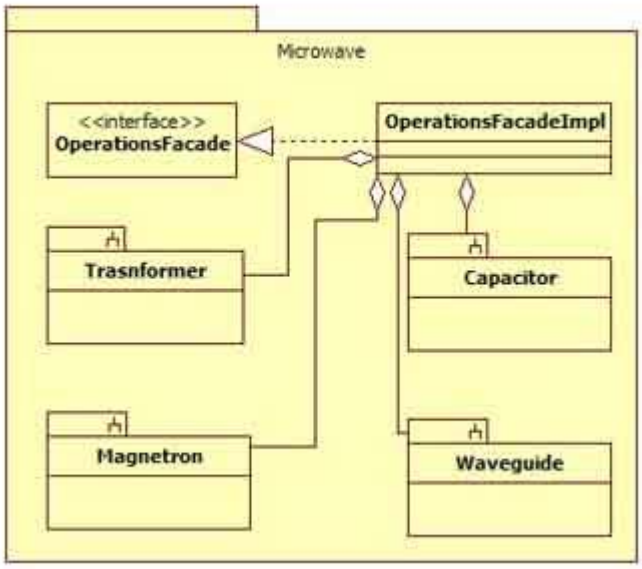
I wish to give you couple of real world examples. Lets take a car, starting a car involves multiple steps. Imagine how it would be if you had to adjust n number of valves and controllers. The facade you have got is just a key hole. On turn of a key it send instruction to multiple subsystems and executes a sequence of operation and completes the objective. All you know is a key turn which acts as a facade and simplifies your job.

Similarly consider microwave oven, it consists of components like trasnformer, capacitor, magnetron, waveguide and some more. To perform an operation these different components needs to be activated in a sequence. Every components has different outputs and inputs. Imagine you will have separate external controller for all these components using which you will heat the food. It will be complicated and cumbersome.

In this scenario, oven provides you preprogrammed switches which can be considered as a facade. On click on of a single switch the job gets done. That single menu switch works as an abstraction layer between you and the internal components.

These are realworld examples for facade design pattern. In software scenario, you can have interfaces which acts as a facade. Methods in these interfaces contains the interaction sequence, formatting and converting data for input for components. As such it will not hold the business logic.

UML Diagram for Facade Design Pattern



Common Mistakes while Implementing Facade Design Pattern

In my experience the common mistakes I have seen is,

- just for the sake of introducing a facade layer developers tend to create additional classes. Layered architecture is good but assess the need for every layer. Just naming a class as ABCDFacade.java doesn'r really make it a facade.
- Creating a java class and 'forcing' the UI to interact with other layers through it and calling it a facade layer is one more popular mistake. Facade layer should not be forced and its always optional. If the client wishes to interact with components directly it should be allowed to bypass the facade layer.
- Methods in facade layer has only one or two lines which calls the other components. If facade is going to be so simple it invalidates its purpose and clients can directly do that by themselves.
- A controller is not a facade.
- Facade is 'not' a layer that imposes security and hides important data and implementation.
- Don't create a facade layer in advance. If you feel that in future the subsystem is going to evolve and become complicated to defend that do not create a stub class and name it a facade. After the subsystem has become complex you can implement the facade design pattern.
- Subsystems are not aware of facade and there should be no reference for facade in subsystems.

Summary of Facade Design Pattern

- Facade provides a single interface.
- Programmers comfort is a main purpose of facade.

- Simplicity is the aim of facade pattern.
- Facade design pattern is used for promoting subsystem independence and portability.
- Subsystem may be dependent with one another. In such case, facade can act as a coordinator and decouple the dependencies between the subsystems.
- Translating data to suit the interface of a subsystem is done by the facade.

Facade Vs Mediator Design Pattern

Mediator design pattern may look very similar to facade design pattern in terms of abstraction. Mediator abstracts the functionality of the subsystems in this way it is similar to the facade pattern. In the implementation of mediator pattern, subsystem or peers components are aware of the mediator and that interact with it. In the case of facade pattern, subsystems are not aware of the existence of facade. Only facade talks to the subsystems.

Facade Design Pattern in Java API

ExternalContext behaves as a facade for performing cookie, session scope and similar operations. Underlying classes it uses are HttpSession, ServletContext, javax.servlet.http.HttpServletRequest and javax.servlet.http.HttpServletResponse.