Java Design Patterns

Java Design Patterns

Gang of Four (GoF)

Design Patterns: Elements of Reusable Object-Oriented Software (1994) is a software engineering book describing software design patterns.

The book was written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (the "Gang of Four"), with a foreword by Grady Booch.

This book has been influential to the field of software engineering and is regarded as an important source for object-oriented design theory and practice

Java Design Patterns

- Design pattern provides a general reusable solution for the common problems occurs in software design. These are the best practices, used by the experienced developers.
- Patterns are not complete code, but it can use as a template which can be applied to a problem
- The patterns typically show relationships and interactions between classes or objects.
- The idea is to speed up the development process by providing well tested, proven development/design paradigm.

Java Design Patterns

Design patterns can be categorized in the following categories:

- Creational patterns
- Structural patterns
- Behavior patterns

J2EE Design Patterns

J2EE Patterns

- MVC Pattern
- Data Access Object Pattern
- Front Controller Pattern
- Transfer Object Pattern
- Intercepting Filter Pattern
- Service Locator Pattern
- Composite Entity Pattern

Creational patterns

Creational patterns

These design patterns are all about class instantiation or object creation.

These patterns can be further categorized into Class-creational patterns and object-creational patterns.

While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

Important creational design patterns are

- Factory Method
- Abstract Factory
- Builder
- Singleton
- Prototype.

There are two recurring themes in these patterns.

First, they all encapsulate knowledge about which concrete classes the system uses.

Second, they hide how instances of these classes are created and put together.

Creational patterns give you a lot of flexibility in what gets created, who creates it, how it gets created, and when.

Structural patterns

Structural patterns

These design patterns are about organizing different classes and objects to form larger structures and provide new functionality.

Important structural design patterns are

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Structural patterns are concerned with how classes and objects are composed to form larger structures.
Structural class patterns use inheritance to compose interfaces or implementations.

Behavioral patterns

Behavioral patterns

Behavioral patterns are about identifying common communication patterns between objects and realize these patterns.

Important behavioral patterns are

- Chain of responsibility
- Command,
- Interpreter
- Iterator
- Mediator
- Memento
- Observer

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them.

These patterns characterize complex control flow that’s difficult to follow at run-time.

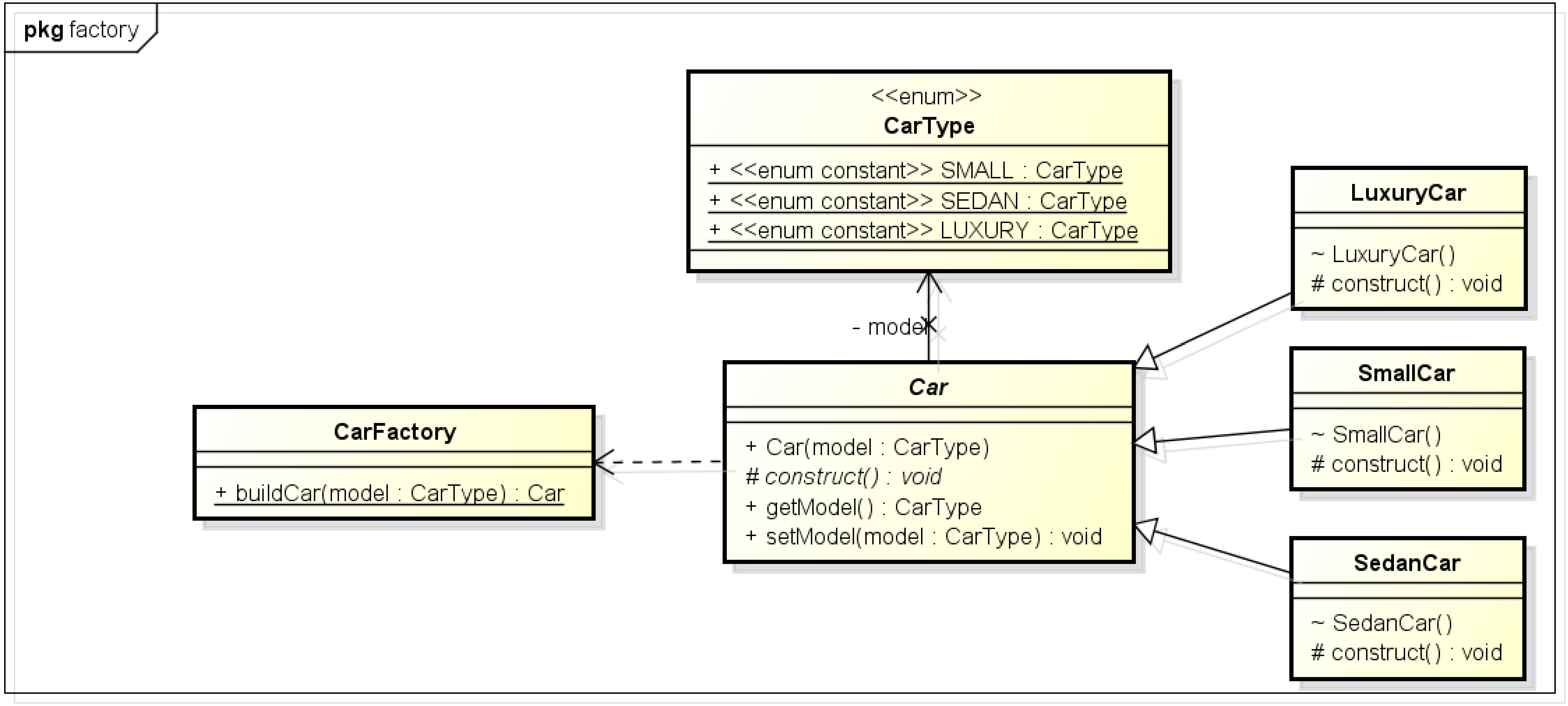
They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected.

Creational pattern : Factory pattern

Factory pattern

- The Factory Method Pattern gives us a way to encapsulate the instantiations of concrete types.
- The Factory Method pattern encapsulates the functionality required to select and instantiate an appropriate class, inside a designated method referred to as a factory method.
- The Factory Method selects an appropriate class from a class hierarchy based on the application context and other influencing factors. It then instantiates the selected class and returns it as an instance of the parent class type.
- The advantage of this approach is that the application objects can make use of the factory method to get access to the appropriate class instance. This eliminates the need for an application object to deal with the varying class selection criteria.

Creational pattern : Factory pattern



Creational pattern : Factory pattern

Factory pattern

Factory pattern is most suitable where there is some complex object creation steps are involved.

To ensure that these steps are centralized and not exposed to composing classes, factory pattern should be used.

Realtime examples of factory pattern in JDK itself.

- java.sql.DriverManager#getConnection()
- java.net.URL#openConnection()
- java.lang.Class#newInstance()
- java.lang.Class#forName()

Creational pattern : Abstract Factory pattern

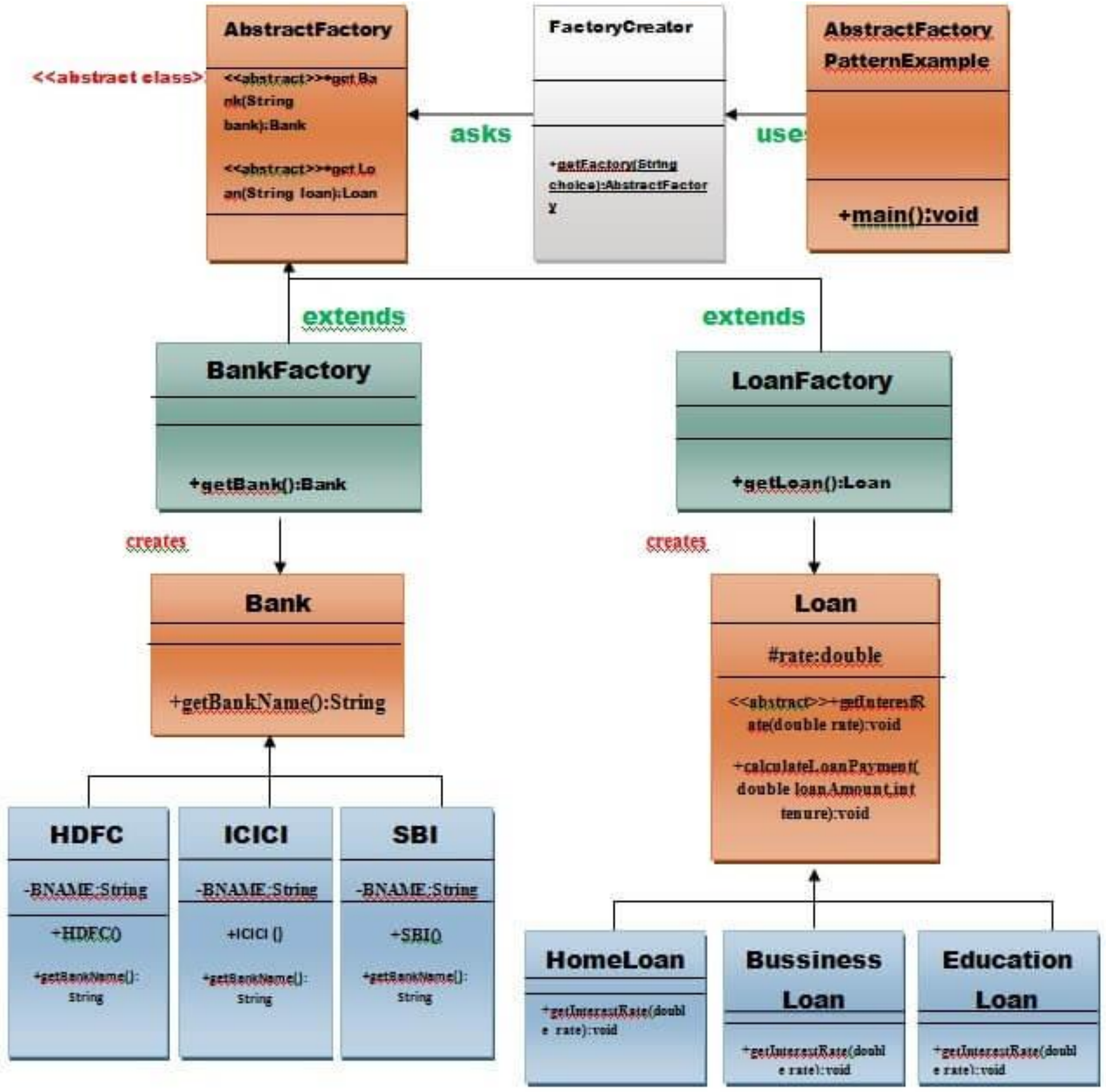
Abstract Factory design pattern

Abstract Factory design pattern is one of the Creational pattern.

Abstract Factory pattern is almost similar to Factory Pattern is considered as another layer of abstraction over factory pattern.

Abstract Factory patterns work around a super-factory which creates other factories.

Creational pattern : Abstract Factory pattern



Creational pattern : Builder pattern

Builder pattern

Using the Builder pattern, the process of constructing such a complex object can be designed more effectively.
The Builder pattern suggests moving the construction logic out of the object class to a separate class referred to as a builder class.

There can be more than one such builder classes, each with different implementations for the series of steps to construct the object.

Each builder implementation results in a different representation of the object.

The intent of the Builder Pattern is to separate the construction of a complex object from its representation, so that the same construction process can create different representations.

This type of separation reduces the object size

Creational pattern : Builder pattern

To illustrate the use of the Builder Pattern, let’s help a Car company which shows its different cars using a graphical model to its customers.

The company has a graphical tool which displays the car on the screen. The requirement of the tool is to provide a car object to it.

The car object should contain the car’s specifications.

The graphical tool uses these specifications to display the car.

The company has classified its cars into different classifications like Sedan or Sports Car.

There is only one car object, and our job is to create the car object according to the classification.

For example, for a Sedan car, a car object according to the sedan specification should be built or, if a sports car is required, then a car object according to the sports car specification should be built.
Currently, the Company wants only these two types of cars, but it may require other types of cars also in the future.
We will create two different builders, one of each classification, i.e., for sedan and sports cars. These two builders will help us in building the car object according to its specification.

Another form of the Builder Pattern

There is another form of the Builder Pattern other than what we have seen so far.
Sometimes there is an object with a long list of properties, and most of these properties are optional.

Consider an online form which needs to be filled in order to become a member of a site.

You need to fill all the mandatory fields but you can skip the optional fields or sometimes it may look valuable to fill some of the optional fields.

The question is, what sort of constructor should we write for such a class?

Well writing a constructor with long list of parameters is not a good choice, this could frustrate the client especially if the important fields are only a few.

This could increase the scope of error; the client may provide a value accidentally to a wrong field.

Another way is to use **telescoping constructors**, in which you provide a constructor with only the required parameters, another with a single optional parameter, a third with two optional parameters, and so on, culminating in a constructor with all the optional parameters.
The telescoping constructor works, but it is hard to write client code when there are many parameters, and it is even harder to read it.

Another form of the Builder Pattern

There is a third alternative that combines the safety of the telescoping constructor pattern with the readability of the JavaBeans pattern.

It is a form of the Builder pattern. Instead of making the desired object directly, the client calls a constructor with all of the required parameters and gets a builder object.

Then the client calls setter-like methods on the builder object to set each optional parameter of interest.

Finally, the client calls a parameter less build method to generate the object.

Builder Pattern in JDK

- java.lang.StringBuilder#append() (unsynchronized)
- java.lang.StringBuffer#append() (synchronized)
- java.nio.ByteBuffer#put()(alsoonCharBuffer,ShortBuffer,IntBuffer,LongBuffer,FloatBufferandDoubleBuffer)
- javax.swing.GroupLayout.Group#addComponent()
- All implementations of java.lang.Appendable

Creational pattern : Singleton pattern

Singleton pattern

Sometimes it’s important for some classes to have exactly one instance.

There are many objects we only need one instance of them and if we, instantiate more than one, we’ll run into all sorts of problems like incorrect program behavior, overuse of resources, or inconsistent results.

You may require only one object of a class, for example, when you are a creating the context of an application, or a thread manageable pool, registry settings, a driver to connect to the input or output console etc.

More than one object of that type clearly will cause inconsistency to your program.

The Singleton Pattern ensures that a class has only one instance, and provides a global point of access to it.

Creational pattern : Singleton pattern

Singleton pattern

```
public class SingletonLazy {
        private static SingletonLazy s = null;
        private SingletonLazy(){}

        public static synchronized SingletonLazy getInstance(){
                if(s==null){
                        s = new SingletonLazy();
                }
                return s;
         }
}
```

Creational pattern : Singleton pattern

Singleton pattern

If you want to use synchronization, there is another technique known as “double-checked locking” to reduce the use of synchronization.
With the double-checked locking, we first check to see if an instance is created, and if not, then we synchronize. This way, we only synchronize the first time.

```
public class SingletonLazyDoubleCheck {
        private volatile static SingletonLazyDoubleCheck sc = null;
        private SingletonLazyDoubleCheck(){}
        public static SingletonLazyDoubleCheck getInstance(){
                if(sc==null){
                        synchronized(SingletonLazyDoubleCheck.class){
                                if(sc==null){
                                        sc = new SingletonLazyDoubleCheck();
                                }
                        }
                }
        return sc;
        }
}
```

Different ways of creating Singleton instances

Singleton pattern

1. We can create a singleton class using **enums**.

The Enum constants are implicitly static and final and you cannot change their values once created.

```
public class SingletoneEnum {
	public enum SingleEnum{
		SINGLETON_ENUM;
	}
}
```

Different ways of creating Singleton instances

Singleton pattern

2. **Bill Pugh Singleton Implementation:**

Prior to Java5, memory model had a lot of issues and above methods caused failure in certain scenarios in multithreaded environment. So, Bill Pugh suggested a concept of inner static classes to use for singleton..

```
// Java code for Bill Pugh Singleton Implementation
public class GFG {

private GFG() {
        // private constructor
}

// Inner class to provide instance of class
private static class BillPughSingleton {
        private static final GFG INSTANCE = new GFG();
}

public static GFG getInstance() {
        return BillPughSingleton.INSTANCE;
}
}
```

When the singleton class is loaded, inner class is not loaded and hence doesn't create object when loading the class.

Inner class is created only when getInstance() method is called. So it may seem like eager initialization but it is lazy initialization.

Examples of Singleton class

Examples of Singleton class

1.java.lang.Runtime :
Java provides a class Runtime in its java.lang package which is singleton in nature.
Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the getRuntime() method.

An application cannot instantiate this class so multiple objects can’t be created for this class. Hence Runtime is a singleton class.

2.java.awt.Desktop : The Desktop class allows a Java application to launch associated applications registered on the native desktop to handle a URI or a file.

Creational Pattern : Prototype

Prototype pattern

The Prototype design pattern is used to specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
The concept is to copy an existing object rather than creating a new instance from scratch, something that may include costly operations.

The existing object acts as a prototype and contains the state of the object. The newly copied object may change same properties only if required.

This approach saves costly resources and time, especially when the object creation is a heavy process.
In Java, there are certain ways to copy an object in order to create a new one. One way to achieve this is using the Cloneable interface.

Java provides the clone method, which an object inherits from the Object class. You need to implement the Cloneable interface and override this clone() method according to your needs.

Prototype Pattern in JDK

- java.lang.Object#clone()
- java.lang.Cloneable

Structural patterns

Structural patterns

Structural patterns are concerned with how classes and objects are composed to form larger structures..

Important structural design patterns are

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Structural patterns : Adapter pattern

Adapter pattern

An Adapter pattern acts as a connector between two incompatible interfaces that otherwise cannot be connected directly.

An Adapter wraps an existing class with a new interface so that it becomes compatible with the client’s interface.

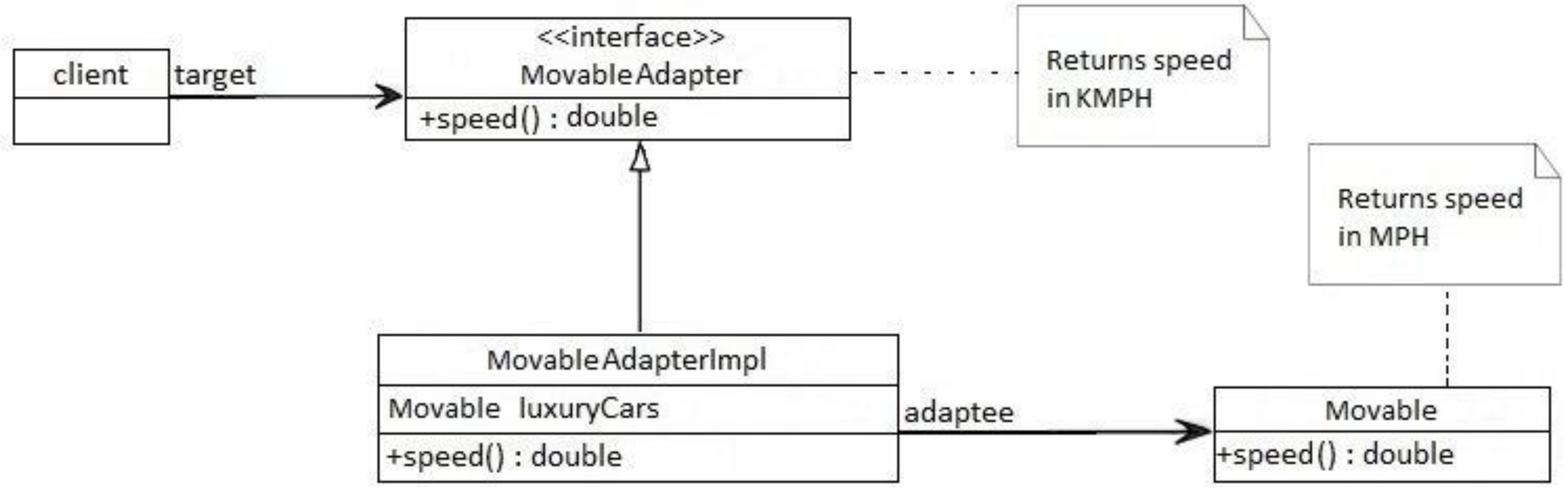
The main motive behind using this pattern is to convert an existing interface into another interface that the client expects. It’s usually implemented once the application is designed.

Adapter Pattern Example

Consider a scenario in which there is an app that's developed in the US which returns the top speed of luxury cars in miles per hour (MPH).

Now we need to use the same app for our client in the UK that wants the same results but in kilometers per hour (km/h).

To deal with this problem, we'll create an adapter which will convert the values and give us the desired results:



Adapter Pattern Example

First, we’ll create the original interface *Movable* which is supposed to return the speed of some luxury cars in miles per hour:

```
public interface Movable {
    // returns speed in MPH
    double getSpeed();
}
```

Create one concrete implementation of this interface:

```
public class BugattiVeyron implements Movable {

    @Override
    public double getSpeed() {
        return 268;
    }
}
```

Adapter Pattern Example

Now we'll create an adapter interface *MovableAdapter* that will be based on the same *Movable* class.

It may be slightly modified to yield different results in different scenarios:

```
public interface MovableAdapter {
    // returns speed in KM/H
    double getSpeed();
}
```

The implementation of this interface will consist of private method *convertMPHtoKMPH()* that will be used for the conversion:

```
public class MovableAdapterImpl implements MovableAdapter {
    private Movable luxuryCars;
    // standard constructors
    @Override
    public double getSpeed() {
        return convertMPHtoKMPH(luxuryCars.getSpeed());
    }
    private double convertMPHtoKMPH(double mph) {
        return mph * 1.60934;
    }
}
```

Structural patterns : Adapter pattern

Adapter pattern

When to use Adapter Pattern

The Adapter pattern should be used when:

- There is an existing class, and its interface does not match the one you need.
- You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don’t necessarily have compatible interfaces.
- There are several existing subclasses to be use, but it’s impractical to adapt their interface by sub classing every one. An object adapter can adapt the interface of its parent class.

Adapter Design Pattern Example in JDK

Some of the adapter design pattern found in JDK classes are;

- java.util.Arrays#asList()
- java.io.InputStreamReader(InputStream) (returns a Reader)
- java.io.OutputStreamWriter(OutputStream) (returns a Writer)

Structural patterns : Bridge pattern

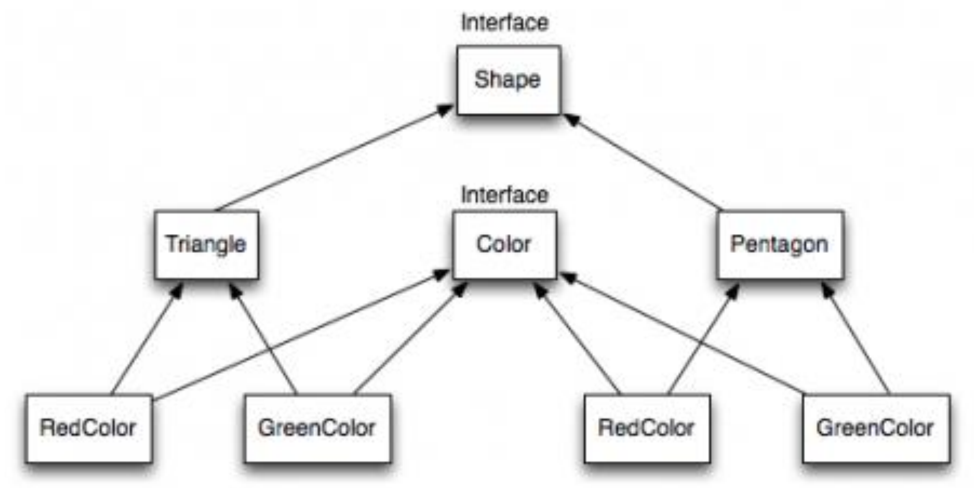
Bridge pattern

The official definition for Bridge design pattern introduced by Gang of Four (GoF) is to decouple an abstraction from its implementation so that the two can vary independently.

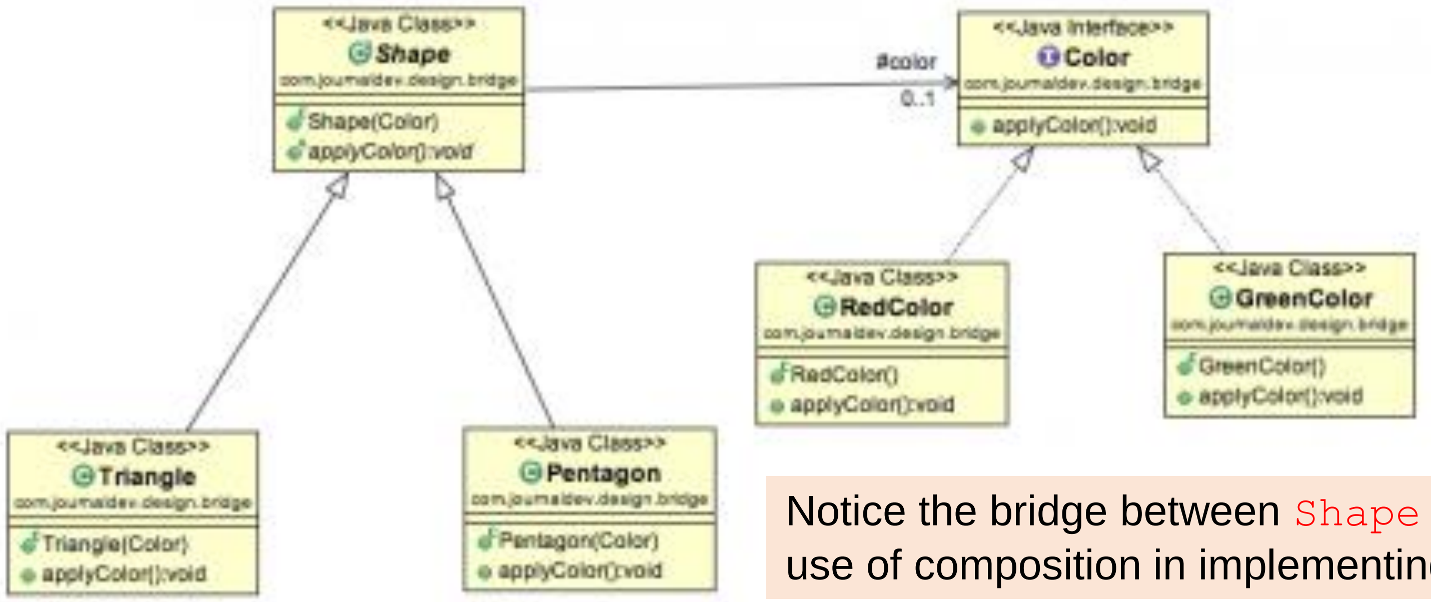
This means to create a bridge interface that uses OOP principles to separate out responsibilities into different abstract classes.

Structural patterns : Bridge pattern example

Bridge pattern



Now we will use bridge design pattern to decouple the interfaces from implementation.
UML diagram for the classes and interfaces after applying bridge pattern will look like below image.



Notice the bridge between `Shape` and `Color` interfaces and use of composition in implementing the bridge pattern.

Structural patterns : Composite pattern example

Composite pattern

The composite pattern is meant to allow treating individual objects and compositions of objects, or "composites" in the same way.
It can be viewed as a tree structure made up of types that inherit a base type, and it can represent a single part or a whole hierarchy of objects.

We can break the pattern down into:

- **component** – is the base interface for all the objects in the composition. It should be either an interface or an abstract class with the common methods to manage the child composites.
- **leaf** – implements the default behavior of the base component. It doesn't contain a reference to the other objects.
- **composite** – has leaf elements. It implements the base component methods and defines the child-related operations.
- **client** – has access to the composition elements by using the base component object.

Structural patterns : Decorator pattern

Decorator pattern

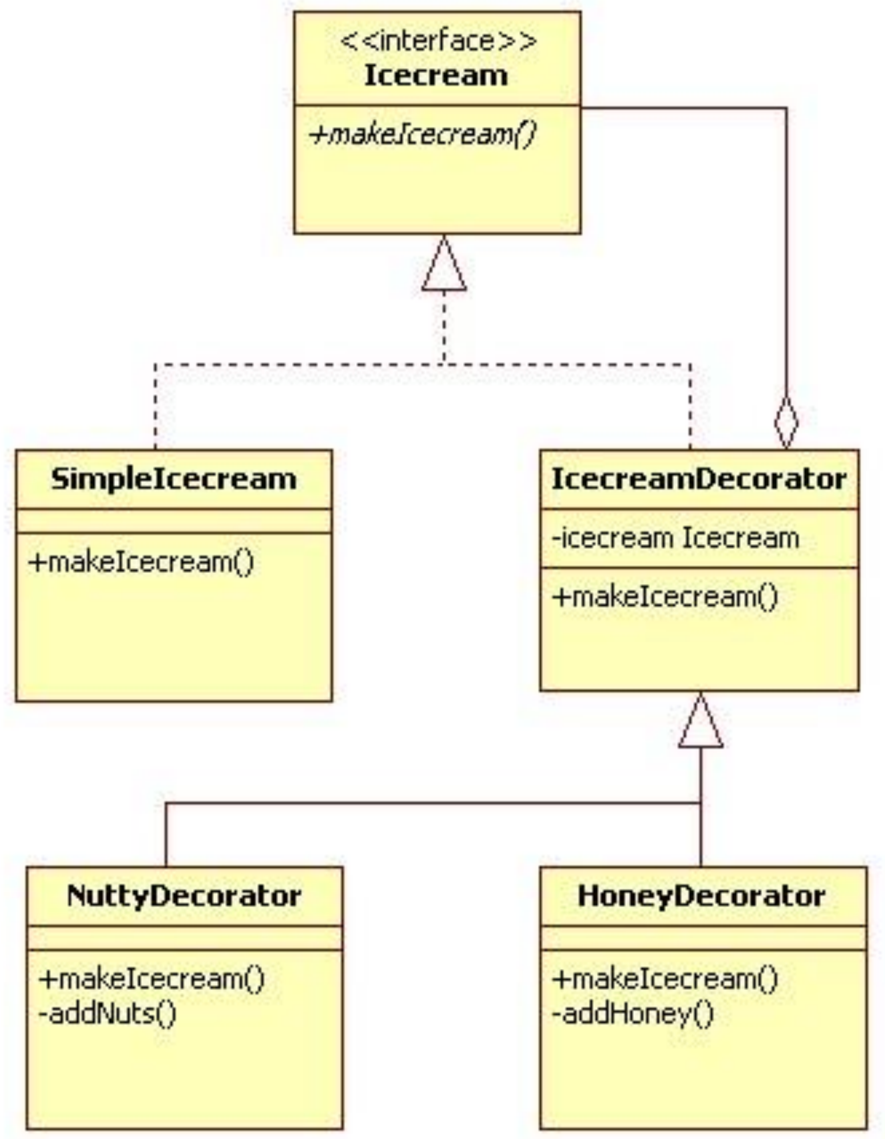
A Decorator pattern can be used to attach additional responsibilities to an object either statically or dynamically.
A Decorator provides an enhanced interface to the original object.

In the implementation of this pattern, we prefer composition over an inheritance – so that we can reduce the overhead of sub classing again and again for each decorating element.

The recursion involved with this design can be used to decorate our object as many times as we require.

Structural patterns : Decorator pattern

Decorator pattern



Structural patterns : Decorator pattern

Decorator pattern

Decorator Design Pattern in Java

- java.io.BufferedInputStream(InputStream)
- java.io.DataInputStream(InputStream)
- java.io.BufferedOutputStream(OutputStream)
- java.util.zip.ZipOutputStream(OutputStream)
- java.util.Collections#checked[List|Map|Set|SortedSet|SortedMap]()

Structural patterns : Facade pattern

Facade pattern

A facade encapsulates a complex subsystem behind a simple interface.

It hides much of the complexity and makes the subsystem easy to use.

Besides a much simpler interface, there’s one more benefit of using this design pattern.

It decouples a client implementation from the complex subsystem.

This allows to make changes to the existing subsystem and don’t affect a client.

Structural patterns : Facade pattern

Facade pattern

Facade Design Pattern in Java API

ExternalContext of **javax.faces.context.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.

Remember facade does not reduce the complexity. It only hides it from external systems and clients. So the primary beneficiary of facade patterns are client applications and other systems only.

It provides a simple interface to clients i.e. instead of presenting complex subsystems, we present one simplified interface to clients. It can also help us to reduce the number of objects that a client needs to deal with.

Structural patterns : Flyweight pattern

Flyweight pattern

The flyweight pattern is based on a factory which recycles created objects by storing them after creation. Each time an object is requested, the factory looks up the object in order to check if it's already been created. If it has, the existing object is returned – otherwise, a new one is created, stored and then returned.

The flyweight object's state is made up of an invariant component shared with other similar objects (**intrinsic**) and a variant component which can be manipulated by the client code (**extrinsic**).

It's very important that the flyweight objects are immutable: any operation on the state must be performed by the factory.

This pattern is used to reduce the memory footprint. It can also improve performance in applications where object instantiation is expensive.

Structural patterns : Flyweight pattern

Flyweight pattern

Implementation

The main elements of the pattern are:

- an interface which defines the operations that the client code can perform on the flyweight object
- one or more concrete implementations of our interface
- a factory to handle objects instantiation and caching

Flyweight Design Pattern Example in JDK

All the wrapper classes valueOf method uses cached objects showing use of Flyweight design pattern.

Another example is Java String class String Pool implementation.

Structural patterns : Proxy pattern

Proxy pattern

Proxy design pattern intent according to GoF is:
Provide a surrogate or placeholder for another object to control access to it.

The definition itself is very clear and proxy design pattern is used when we want to provide controlled access of a functionality.

Proxy means 'in place of', representing' or 'in place of' or 'on behalf of' are literal meanings of proxy and that directly explains **Proxy Design Pattern**.

Proxies are also called surrogates, handles, and wrappers.

A real world example can be a cheque or credit card is a proxy for what is in our bank account. It can be used in place of cash, and provides a means of accessing that cash when required. And that's exactly what the Proxy pattern does – "**Controls and manage access to the object they are protecting**".

Structural patterns : Proxy pattern

Proxy pattern

Benefits:

•One of the advantages of Proxy pattern is security.

•This pattern avoids duplication of objects which might be huge size and memory intensive. This in turn increases the performance of the application.

•The remote proxy also ensures about security by installing the local code proxy (stub) in the client machine and then accessing the server with help of the remote code.

Drawbacks/Consequences:

This pattern introduces another layer of abstraction which sometimes may be an issue if the Real classes are accessed by some of the clients directly and some of them might access the Proxy classes. This might cause disparate behaviour.

Proxy Pattern in JDK

The following cases are examples of usage of the Proxy Pattern in the JDK.

- **java.lang.reflect.Proxy**
- **java.rmi.* (whole package)**

Behavioral patterns

Behavioral patterns

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them.

Important behavioral patterns are

- Chain of responsibility
- Iterator
- Mediator
- Memento
- Observer

Behavioral patterns : Chain of responsibility

Chain of responsibility pattern

Chain of responsibility pattern is used to achieve loose coupling in software design where a request from client is passed to a chain of objects to process them.

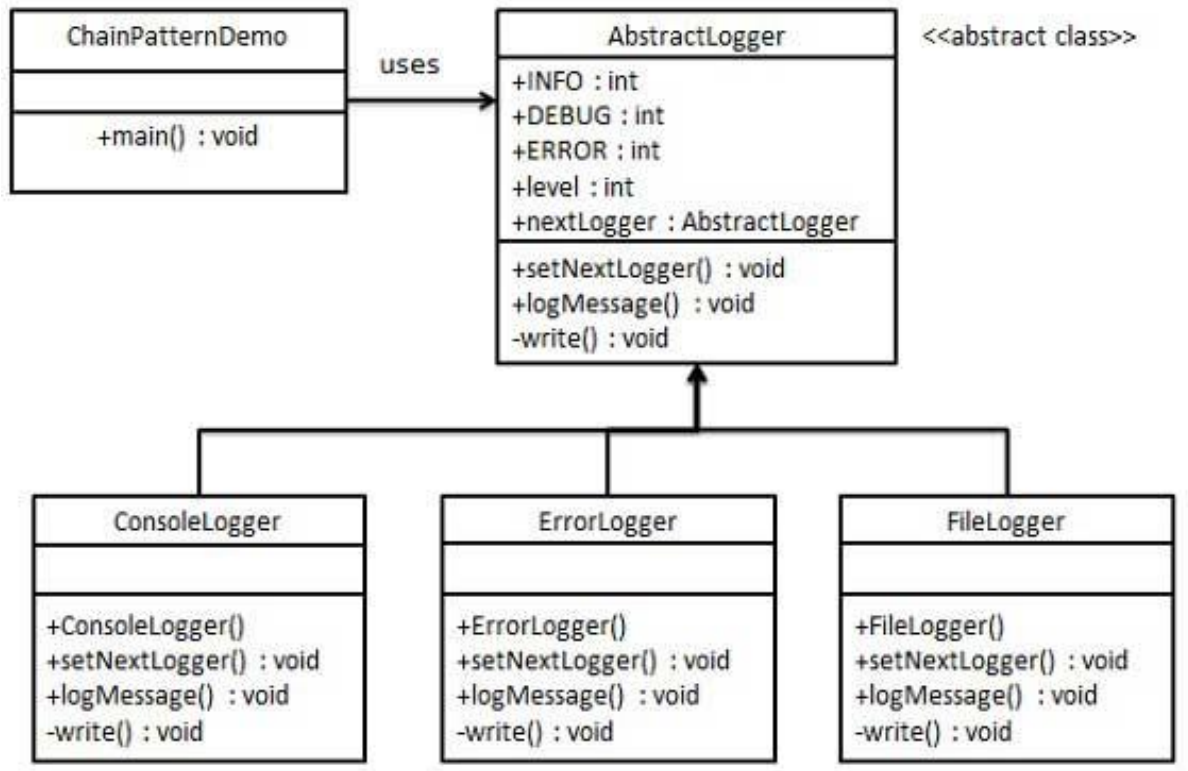
Then the object in the chain will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.

Chain of Responsibility is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

Chain of responsibility example

Chain of responsibility pattern

Create an abstract class *AbstractLogger* with a level of logging. Then create three types of loggers extending the *AbstractLogger*. Each logger checks the level of message to its level and print accordingly otherwise does not print and pass the message to its next logger.



Behavioral patterns : Chain of responsibility

Chain of responsibility pattern

Chain of Responsibility Design Pattern Example
One of the example of Chain of Responsibility pattern is **ATM Cash Deposit machine**. The user enters the amount to be dispensed and the machine dispense amount in terms of defined currency bills such as 100, 500, 2000 etc.

Chain of Responsibility Pattern Example in JDK

- When any exception occurs in the try block, its send to the first catch block to process. If the catch block is not able to process it, it forwards the request to next object in chain i.e next catch block. If even the last catch block is not able to process it, the exception is thrown outside of the chain to the calling program.

- java.util.logging.Logger#log()

- javax.servlet.Filter#doFilter()

Behavioral patterns : Iterator pattern

Iterator pattern

Iterator pattern is used to provide a standard way to traverse through a group of Objects.

Iterator pattern is widely used in Java Collection Framework. Iterator interface provides methods for traversing through a collection.

Behavioral patterns : Observer pattern

Observer pattern

Observer is a behavioral design pattern. It specifies communication between objects: *observable* and *observers*.

An *observable* is an object which notifies *observers* about the changes in its state.

For example, a news agency can notify channels when it receives news. Receiving news is what changes the state of the news agency, and it causes the channels to be notified.

Behavioral patterns : Observer pattern

Observer pattern

```java
public interface Channel {
    public void update(Object o);
}
```

```java
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);
        }
    }
}
```

```java
/*
 * NewsChannel class overrides update()
 * method which is invoked when state of
NewsAgency changes
 */
public class NewsChannel implements Channel {
    private String news;

    public String getNews() {
return news;
}

public void setNews(String news) {
this.news = news;
}

@Override
    public void update(Object news) {
        this.setNews((String) news);
    }
}
```

In the tester class, add an instance of NewsChannel to the list of observers, and change the state of NewsAgency, the instance of NewsChannel will be updated:

Observer pattern : Implementation with Java API, *Observer*

Observer pattern **Implementation with Java API, *Observer***

The *java.util.Observer* interface defines the *update()* method, so there's no need to define it ourselves

```
public class ONewsChannel implements Observer {

    private String news;

    @Override
    public void update(Observable o, Object news) {
        this.setNews((String) news);
    }
}
```

Note : The second argument comes from *Observable*..
To define the observable, we need to extend
Java's *Observable* class:

```
public class ONewsAgency extends Observable {
    private String news;

    public void setNews(String news) {
        this.news = news;
        setChanged();
        notifyObservers(news);
    }
}
```

Note that we don't need to call the observer's *update()* method directly. We just call *stateChanged()*and *notifyObservers()*, and the *Observable* class is doing the rest for us.
Also, it contains a list of observers and exposes methods to maintain that list – *addObserver()* and *deleteObserver().*

Tester class:

```
ONewsAgency observable = new ONewsAgency();
ONewsChannel observer = new ONewsChannel();
observable.addObserver(observer);
observable.setNews("news");
```

Observer pattern : Implementation with *PropertyChangeListener*

Observer pattern **Implementation with *PropertyChangeListener***

In this implementation, an observable must keep a reference to the *PropertyChangeSupport* instance. It

ss is changed.

```
public class PCLNewsAgency {
    private String news;
    private PropertyChangeSupport support;

    public PCLNewsAgency() {
        support = new PropertyChangeSupport(this);
    }
    public void addPropertyChangeListener(PropertyChangeListen
        support.addPropertyChangeListener(pcl);
    }

    public void removePropertyChangeListener(PropertyChangeListener pcl) {
        support.removePropertyChangeListener(pcl);
    }
    public void setNews(String value) {
        support.firePropertyChange("news", this.news, value);
        this.news = value;
    } PCLNewsAgency observable = new PCLNewsAgency();
PCLNewsChannel observer = new PCLNewsChannel();

observable.addPropertyChangeListener(observer);
observable.setNews("news");
```

Observers should implement *PropertyChangeListener*:

```
public class PCLNewsChannel implements
PropertyChangeListener {
    private String news;
    public void propertyChange(PropertyChangeEvent evt) {
        this.setNews((String) evt.getNewValue());
    }
}
```

Tester class:

```
PCLNewsAgency observable = new PCLNewsAgency();
PCLNewsChannel observer = new PCLNewsChannel();

observable.addPropertyChangeListener(observer);
observable.setNews("news");
```

Behavioral patterns : Observer pattern

Observer pattern

Observer design pattern is also called as publish-subscribe pattern.

Some of it’s implementations are;

- java.util.EventListener in Swing
- javax.servlet.http.HttpSessionBindingListener
- javax.servlet.http.HttpSessionAttributeListener

Behavioral patterns : Mediator pattern

Mediator pattern

In object-oriented programming, we should always try to **design the system in such a way that components are loosely coupled and reusable**.

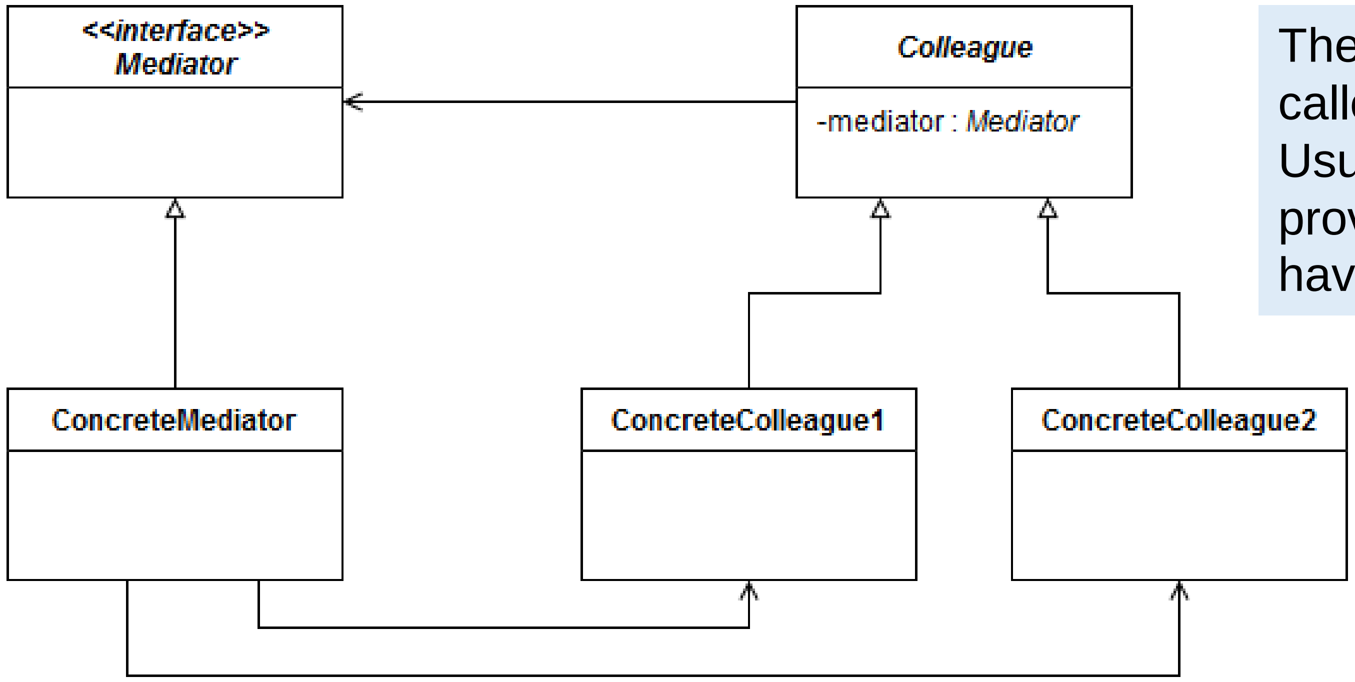
This approach makes our code easier to maintain and test.
In real life, however, we often need to deal with a complex set of dependent objects. This is when the Mediator Pattern may come in handy.

The intent of the Mediator Pattern is to reduce the complexity and dependencies between tightly coupled objects communicating directly with one another.

This is achieved by creating a mediator object that takes care of the interaction between dependent objects. Consequently, all the communication goes through the mediator.

This promotes loose coupling, as a set of components working together no longer have to interact directly. Instead, they only refer to the single mediator object.

Mediator Pattern’s UML diagram



The system objects that communicate each other are called Colleagues.
Usually we have an [interface or abstract class]() that provides the contract for communication and then we have concrete implementation of mediators.

In the above UML diagram, we can identify the following participants:

- *Mediator* defines the interface the *Colleague* objects use to communicate
- *Colleague* defines the abstract class holding a single reference to the *Mediator*
- *ConcreteMediator* encapsulates the interaction logic between *Colleague* objects
- *ConcreteColleague1* and *ConcreteColleague2* communicate only through the *Mediator*

As we can see, ***Colleague* objects do not refer to each other directly. Instead, all the communication is carried out by the *Mediator***.

Mediator Pattern Usage

Air traffic controller is a great example of mediator pattern where the airport control room works as a mediator for communication between different flights. Mediator works as a router between objects and it can have it's own logic to provide way of communication.

Mediator Pattern in JDK

Design Patterns are used almost everywhere in JDK. The following are the usages of the Mediator Pattern in JDK.

java.util.concurrent.ScheduledExecutorService (all scheduleXXX() methods)
java.util.concurrent.ExecutorService (the invokeXXX() and submit() methods)
java.util.concurrent.Executor#execute()
java.util.Timer (all scheduleXXX() methods)
java.lang.reflect.Method#invoke()

Mediator Design Pattern Important Points

•Mediator pattern is useful when the communication logic between objects is complex, we can have a central point of communication that takes care of communication logic.
•Java Message Service (JMS) uses Mediator pattern along with Observer pattern to allow applications to subscribe and publish data to other applications.
•We should not use mediator pattern just to achieve lose-coupling because if the number of mediators will grow, then it will become hard to maintain them.

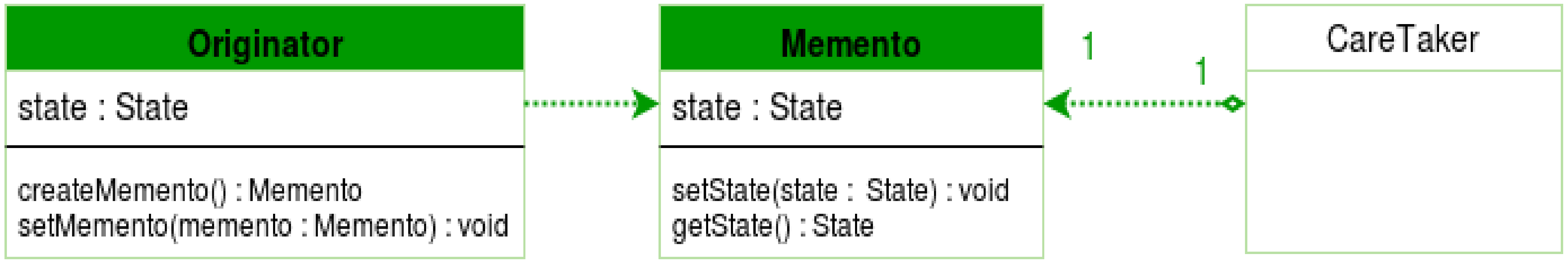
Memento Design Pattern

Memento pattern is a behavioral design pattern. Memento pattern is used to restore state of an object to a previous state. As your application is progressing, you may want to save checkpoints in your application and restore back to those checkpoints later.

Memento Pattern in JDK
- java.util.Date
- java.io.Serializable

UML Diagram Memento design pattern

Design components
- **originator :** the object for which the state is to be saved. It creates the memento and uses it in future to undo.
- **memento :** the object that is going to maintain the state of originator. Its just a POJO.
- **caretaker :** the object that keeps track of multiple memento. Like maintaining savepoints.


Originator
state : State
createMemento() : Memento
setMemento(memento : Memento) : void
Memento
state : State
setState(state : State) : void
getState() : State
1
1
CareTaker

Thank You!