LinkedHashMap

The reason for this behavior is that contract of ConcurrentHashMap does not guarantee that concurrent modifications will be reflected into an already existing iterators

**The first group is *copy-on-write* collections**

CopyOnWriteArrayList, CopyOnWriteArraySet.

**The second group is *Compare-And-Swap* or *CAS* collections**

ConcurrentLinkedQueue ConcurrentSkipListMap

**The third group is concurrent collections using a special**

LinkedBlockingQueue, ConcurrentHashMap

You know, ArrayList is not thread-safe so it’s not safe to use in multi-threaded applications. We can achieve thread-safe feature for an ArrayList by using a synchronized wrapper like this: **lock object**

**List<String> unsafeList = new ArrayList<>();**

**List<String> safeList = Collections.synchronizedList(unsafeList);**

**safeList.add("Boom"); // safe to use with multiple threads**

However, this synchronized list has a limitation: all of its read and write methods (add, set, remove, iterator, etc) are synchronized on the list object itself. That means if a thread is executing add() method, it blocks other threads which want to get the iterator to access elements in the list, for example. Also, only one thread can iterate the list’s elements at a time, which can be inefficient. That’s quite rigid.

CopyOnWriteArrayList :-

/\*\*

\* A thread-safe variant of {@link java.util.ArrayList} in which all mutative

\* operations ({@code add}, {@code set}, and so on) are implemented by

\* making a fresh copy of the underlying array.

\*

\* <p>This is ordinarily too costly, but may be <em>more</em> efficient

\* than alternatives when traversal operations vastly outnumber

\* mutations, and is useful when you cannot or don't want to

\* synchronize traversals, yet need to preclude interference among

\* concurrent threads. The "snapshot" style iterator method uses a

\* reference to the state of the array at the point that the iterator

\* was created. This array never changes during the lifetime of the

\* iterator, so interference is impossible and the iterator is

\* guaranteed not to throw {@code ConcurrentModificationException}.

\* The iterator will not reflect additions, removals, or changes to

\* the list since the iterator was created. Element-changing

\* operations on iterators themselves ({@code remove}, {@code set}, and

\* {@code add}) are not supported. These methods throw

\* {@code UnsupportedOperationException}.

\*

\* <p>All elements are permitted, including {@code null}.

\*

\* <p>Memory consistency effects: As with other concurrent

\* collections, actions in a thread prior to placing an object into a

\* {@code CopyOnWriteArrayList}

\* <a href="package-summary.html#MemoryVisibility"><i>happen-before</i></a>

\* actions subsequent to the access or removal of that element from

\* the {@code CopyOnWriteArrayList} in another thread.

\*

\* <p>This class is a member of the

\* <a href="{@docRoot}/../technotes/guides/collections/index.html">

\* Java Collections Framework</a>.

\*

\* **@since** 1.5

\* **@author** Doug Lea

\* **@param** <E> the type of elements held in this collection

\*/

CopyOnWriteArraySet

\* A {@link java.util.Set} that uses an internal {@link CopyOnWriteArrayList}

\* for all of its operations. Thus, it shares the same basic properties:

\* <ul>

\* <li>It is best suited for applications in which set sizes generally

\* stay small, read-only operations

\* vastly outnumber mutative operations, and you need

\* to prevent interference among threads during traversal.

\* <li>It is thread-safe.

\* <li>Mutative operations ({@code add}, {@code set}, {@code remove}, etc.)

\* are expensive since they usually entail copying the entire underlying

\* array.

\* <li>Iterators do not support the mutative {@code remove} operation.

\* <li>Traversal via iterators is fast and cannot encounter

\* interference from other threads. Iterators rely on

\* unchanging snapshots of the array at the time the iterators were

\* constructed.

\* </ul>

\*

\*

Solid Design Priniciple

|  |  |  |
| --- | --- | --- |
| SRP | [The Single Responsibility Principle](https://docs.google.com/open?id=0ByOwmqah_nuGNHEtcU5OekdDMkk) | A class should have one, and only one, reason to change. |
| OCP | [The Open Closed Principle](http://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgN2M5MTkwM2EtNWFkZC00ZTI3LWFjZTUtNTFhZGZiYmUzODc1&hl=en) | You should be able to extend a classes behavior, without modifying it. |
| LSP | [The Liskov Substitution Principle](http://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgNzAzZjA5ZmItNjU3NS00MzQ5LTkwYjMtMDJhNDU5ZTM0MTlh&hl=en) | Derived classes must be substitutable for their base classes. |
| ISP | [The Interface Segregation Principle](http://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgOTViYjJhYzMtMzYxMC00MzFjLWJjMzYtOGJiMDc5N2JkYmJi&hl=en) | Make fine grained interfaces that are client specific. |
| DIP | [The Dependency Inversion Principle](http://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgMjdlMWIzNGUtZTQ0NC00ZjQ5LTkwYzQtZjRhMDRlNTQ3ZGMz&hl=en) | Depend on abstractions, not on concretions. |

When we have non-cohesive interfaces, the ISP guides us to create multiple, smaller, cohesive interfaces.

Interface Segregation Principle :- What the Interface Segregation Principle says is that your interface should not be bloated with methods that implementing classes don’t require. For such interfaces, also called “fat interfaces”, implementing classes are unnecessarily forced to provide implementations (dummy/empty) even for those methods that they don’t need. In addition, the implementing classes are subject to change when the interface changes. An addition of a method or change to a method signature requires modifying all the implementation classes even if some of them don’t use the method.

The Interface Segregation Principle advocates segregating a “fat interface” into smaller and [highly cohesive](https://en.wikipedia.org/wiki/Cohesion_%28computer_science%29) interfaces, known as “role interfaces”. Each “role interface” declares one or more methods for specific behavior. Thus clients, instead of implementing a “fat interface”, can implement only those “role interfaces” whose methods are relevant to them.

Dependency Inversion Principle :-

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.B. Abstractions should not depend on details. Details should depend on abstractions.”

**Example of DIP –In below example BinarySearch is directly dependent on sorting algorithm i.e BubbleSortAlgorithm.**

DIP is one of the core principles that the Spring Framework enables. Have a look at this example:

public class BinarySearchImpl {

public int binarySearch(int[] numbers, int numberToSearchFor) {

BubbleSortAlgorithm bubbleSortAlgorthm = new BubbleSortAlgorithm();

int[] sortedNumbers = bubbleSortAlgorithm.sort(numbers);

*//...*

}

}

**Instead of**

Better Approach for BinarySearchImpl is to make use of an interface - sort algorithm. Here is how our modified code would look like:

public intrface SortAlgorithm {

public int[] sort(int[] numbers);

}

@Component

public class BinarySearchImpl {

@Autowired

private SortAlgorithm sortAlgorithm;

public BinarySearchImpl(SortAlgorithm sortAlgorithm) {

super();

this.sortAlgorithm = sortAlgorithm;

}

public int[] binarySearch(int[] numbers, int numberToSearchFor) {

int[] sortedNumbers = sortAlgorithm.sort(numbers);

*//...*

}

}

**Other Design Principle** –

Encapsulate what varies

Favour composition over inheritance

**HAS-A vs IS-A** relationtionship - Composition is has-A relationship and Inheritence is IS-A relaionship

## Behavioral Design Patterns

Behavioral patterns provide solution for the better interaction between objects and how to provide lose coupling and flexibility to extend easily.

Strategy Design Pattern :- It is ea

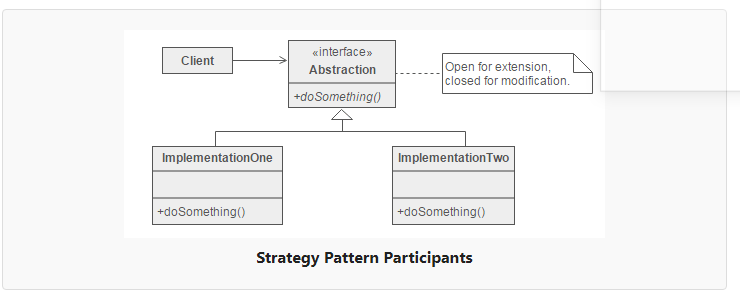


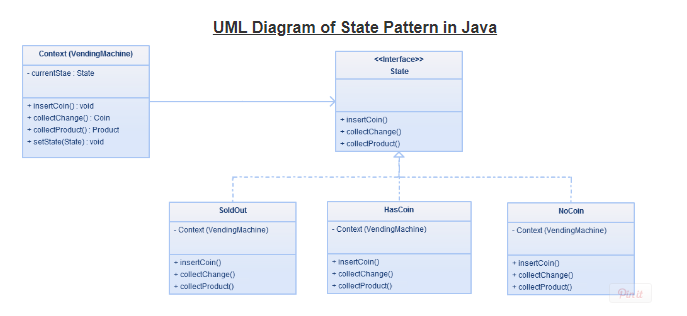
Strategy pattern is used when we have multiple algorithms for a specific task and the client decides the actual implementation be used at runtime.

Strategy design pattern in Java is used to encapsulate related set of algorithms to provide runtime flexibility to client. Client can choose any algorithm at runtime, without changing Context class, which uses Strategy object. Some of the popular example of Strategy pattern is writing code, which uses algorithms e.g. encryption, compression or sorting algorithm.

UML of Strategy Pattern :

* This pattern defines a set of related algorithm and encapsulate them in separated classes, and allows client to choose any algorithm at run time.
* It allows to add new algorithm without modifying existing algorithms or context class, which uses algorithm or strategies
* Strategy is a behavioral pattern in Gang of Four Design pattern list.
* Strategy pattern is based upon Open Closed design principle of SOLID principals.
* Combination of Collections.sort() and Comparator interface is an solid example of Strategy pattern. - method where client actually passes suitable comparator based on the requirement in runtime to the method and the method is generic to accept any comparator type. Based on the comparator being passed, same collection can be sorted differently.
* Appenders, Layouts and Filters in [Log4j](https://howtodoinjava.com/log4j2/).

  
  
State Design Pattern :- State design pattern allows an object to behave differently at different state. Since real world object often has state, and they behave differently at different state



#### Design participants

* **State** – The interface define operations which each state must handle.
* **Concrete States** – The classes which contain the state specific behavior.
* **Context** – Defines an interface to client to interact. It maintains references to concrete state object which may be used to define current state of object. It delegates state-specific behavior to different State objects.

In this example we are simulating package courier delivery system where package can be in different states.

PackageState .java - <ineterface >

Acknwledged.java shipped.java IIntranstion.java OutofDelivery.java Delivered.java

Read more: <http://javarevisited.blogspot.com/2014/04/difference-between-state-and-strategy-design-pattern-java.html#ixzz56yCL0UNk>

**Creational Pattern**

**Factory pattern**

**Abstract Factory Pattern**

**Singleton Pattern**

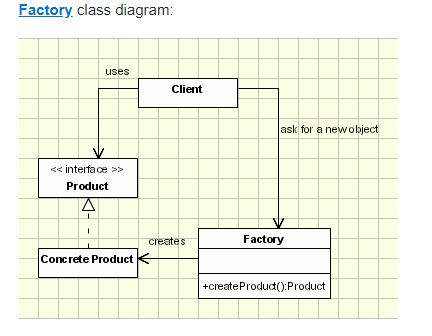
**Builder Pattern**

**Prototype Pattern**

|  |  |
| --- | --- |
| **DESIGN PATTERN NAME** | **PURPOSE** |
| [Builder](https://howtodoinjava.com/design-patterns/creational/builder-pattern-in-java/) | Builder design pattern is an alternative way to construct complex objects and should be used only when we want to build different types of immutable objects using same object building process. |
| [Prototype](https://howtodoinjava.com/design-patterns/creational/prototype-design-pattern-in-java/) | Prototype design pattern is used in scenarios where application needs to create a large number of instances of a class, which have almost same state or differ very little. |
| [Factory](https://howtodoinjava.com/design-patterns/creational/implementing-factory-design-pattern-in-java/) | Factory design pattern is most suitable when complex object creation steps are involved. To ensure that these steps are centralized and not exposed to composing classes. |
| [Abstract factory](https://howtodoinjava.com/design-patterns/creational/abstract-factory-pattern-in-java/) | Abstract factory pattern is used whenever we need another level of abstraction over a group of factories created using factory pattern. |
| [Singleton](https://howtodoinjava.com/design-patterns/singleton-design-pattern-in-java/) | Singleton enables an application to have one and only one instance of a class per JVM. |

**Factory Design Pattern :-** Creates objects without exposing the instantiation logic to the client.

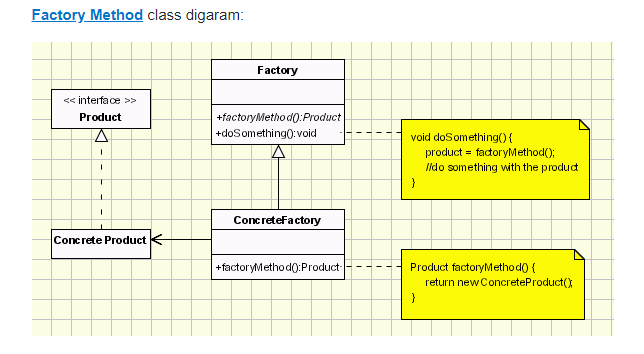
**UML Diagram : -**



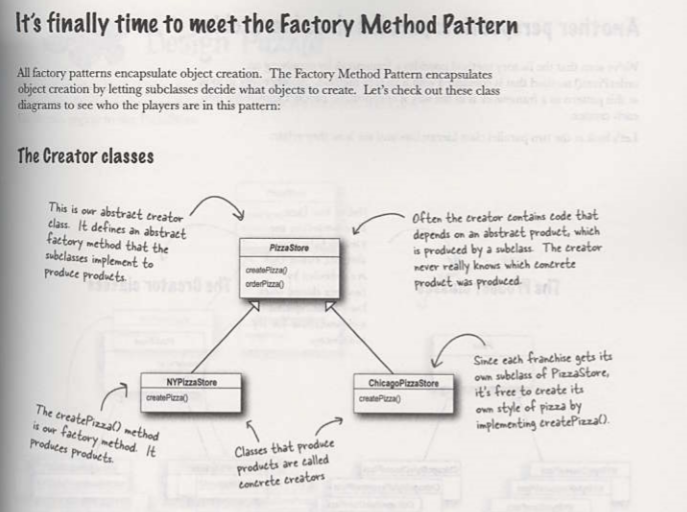
**Example** :- Client is PizzaStore which keeps SimplePizzaFactory instance which returns Pizza Object depend upon type of pizza or Object which client requires.

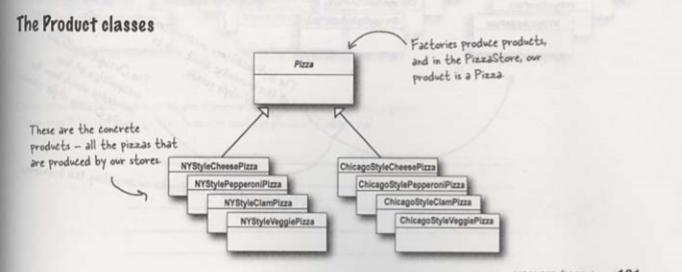


**FactoryMethodPattern :-** Define an interface for creating an object, but let the subclasses decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses



**Example**





*Abstract Factory*: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

## Abstract Factory vs. Factory Method

The methods of an Abstract Factory are implemented as Factory Methods. Both the Abstract Factory Pattern and the Factory Method Pattern decouples the client system from the actual implementation classes through the abstract types and factories. The Factory Method creates objects through inheritance where the Abstract Factory creates objects through composition.

**Buider Pattern:**

**Protype Pattern:** Prototype pattern allows you to create an instance by copying an existing instance.

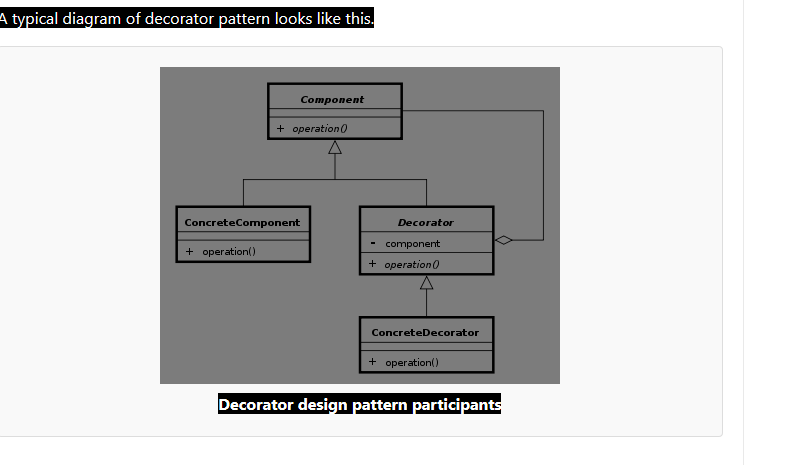
Benefits : -

**Structural Pattern**

**Decorator Pattern** –

Attach the additional responsibility to object dynamically. Decorators provide a flexible altenative to subclassing for extending functionality.

You can attach any number of decorator to wrap a component.



## ****Common usage of decorator pattern:****

1) Java iO library classes e.g. BufferedInputStream bs = new BufferedInputStream(new FileInputStream(new File(“File1.txt”)));

2) In decorator column data in display-tag jsp library e.g.

|  |
| --- |
| <display:table name="reportsViewResultTable" class="demoClass" id="reportsQueryViewResultTable">      <display:column title="Report Id" sortable="true" property="reportDisplayId" decorator="com.comp.FirstColumnDataDecorator"></display:column>  </display:table> |

3) Decorators are used in **sitemesh**, to give a consistent UI experience.

**Adapter Design Pattern**

Adapter design pattern is one of the [**structural design pattern**](https://howtodoinjava.com/category/design-patterns/structural/) and its used so that two unrelated interfaces can work together. The object, that joins these unrelated interfaces, is called an Adapter.

It converts the interface of a class to another interface the client expect. Adapter pattern lets the classes to work together which could not otherwise incompatible because of incompatible type.

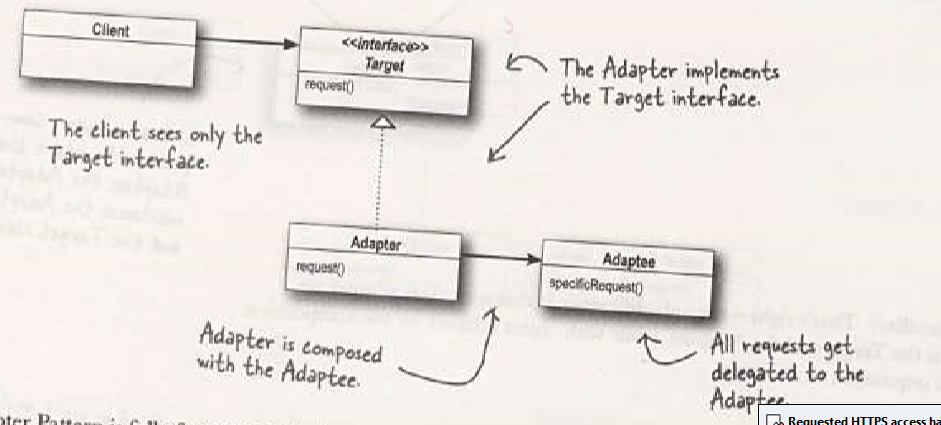
1. Adapter implements target interface (Duck)

2. Adaptee (turkey) is passed via constructor and stored internally

3. Calls by client code are delegated to the appropriate methods in the adaptee

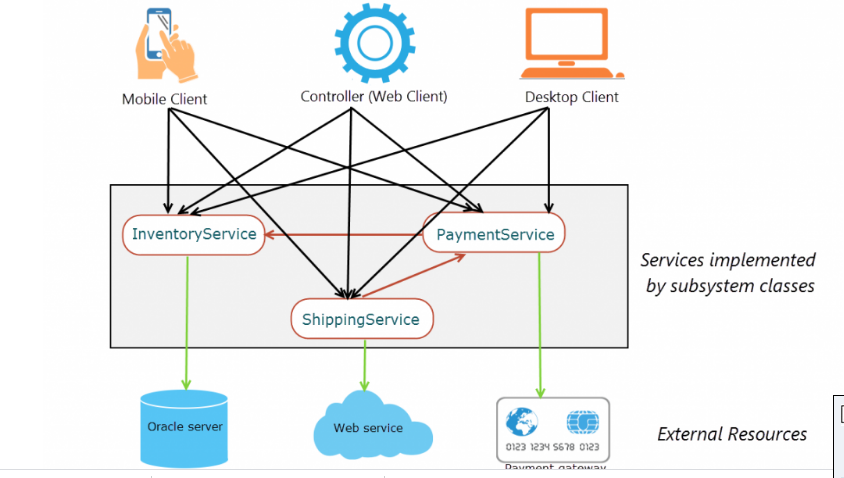
4. Adapter is full-fledged class, could contains additional vars and methods to get its job done; can be used polymorphically as a Duck

UML



**Façade Pattern :-**

**Tightly coupled vs loosly coupled system :**



As you see in above figure :three client webclient, mobile client and desktopclient

Client is directly communicating with the services of the Order fulfillment .

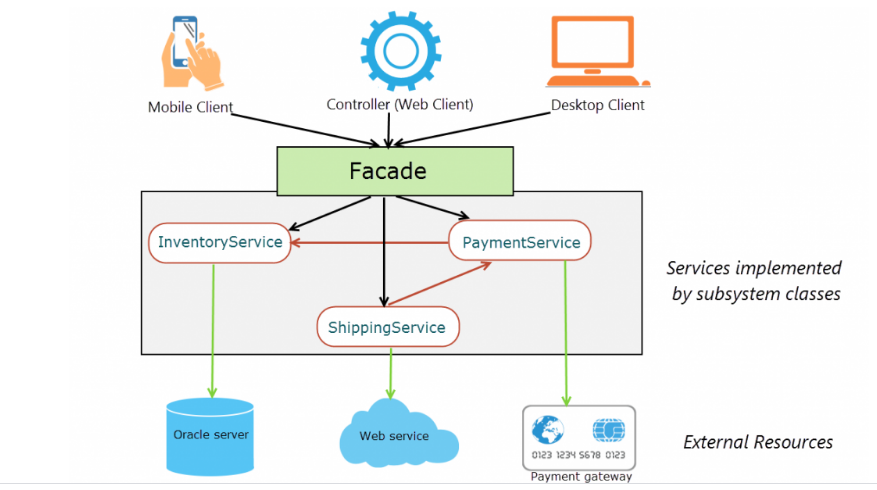
Client needs to know the internal of the subsystem classes.

In it Our client is **tightly coupled** with the subsystem classes.due to which any change in the subsystem will propagated to client layer.

Instead of client tightly coupled to the subsystem, we need an interface makes the subsytem easier to use. Client just place order they really don’t need to know Inventory,payment,shipping and other subsystem.

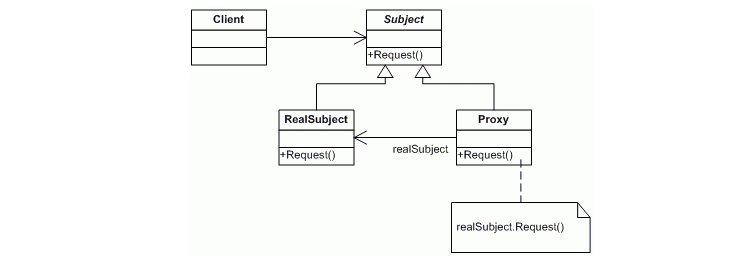
**Facade pattern** hides the complexities of the system and provides an interface to the client using which the client can access the system.

The Facade pattern is a way of providing a simple way for the clients to interact with the subsystems.



**Proxy Pattern** :- The Proxy pattern uses a proxy (surrogate) object “in place of” another object. The objective of a proxy object is to control the creation of and access to the real object it represents. A common use of a proxy is to defer the cost of instantiating of an object (that is expensive to create) until it is actually needed by clients.

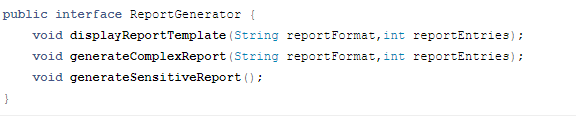
UML



It is extensively used in SpringAOP, RMI.

Example :- In a report viewer application. It generates the report on basis of data, it generate simplereport, complexreport and sensitive report.

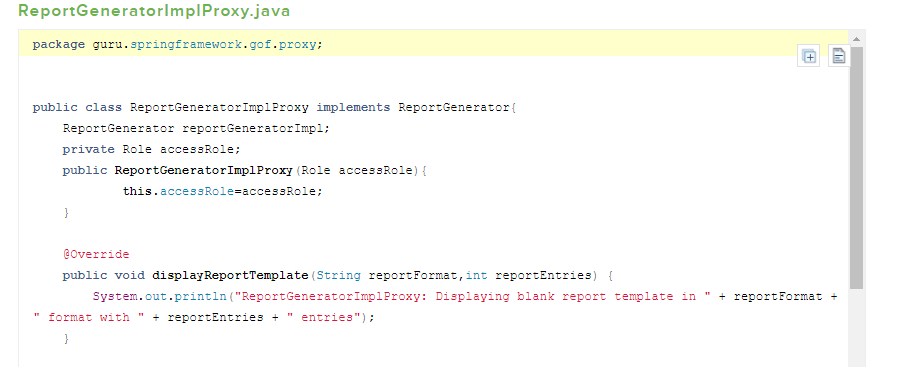
This is Subject Interface :-



RealSubject :-

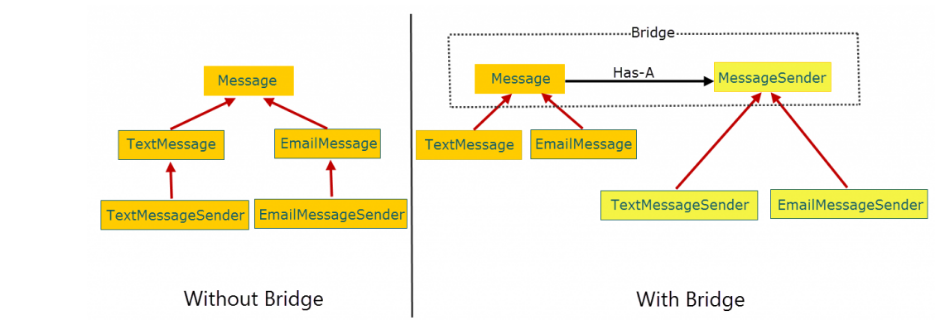


ReportGenratorImplProxy : - It implements ReportGenrator and has a ReportGenratorImpl object. When ever client invokes the proxy object. Inside proxy object it would check whether the use has access or not and act as a proxy before invoking the real method.



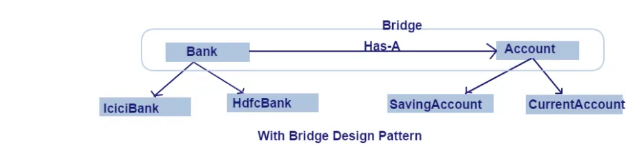


**Bridge Design Pattern** is used to decouples an abstraction used the client code from its implementation that means it separates the abstraction and its implementation in separate class hierarchies.



With the bridge pattern, the abstraction maintains a **Has-A** relationship with the implementation instead of a **IS-A** relationship. The **Has-A** relationship is achieved through composition where the abstraction maintains a reference of the implementation and forwards client requests to it.





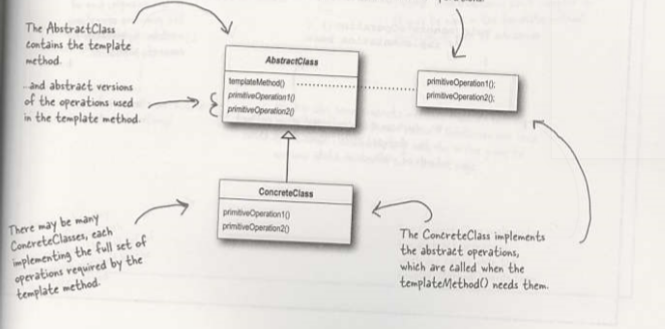
**Behavioural Pattern** :

Strategy Design Pattern :

State Design Pattern:

Template Method Pattern : -It defines the steps of an algorithm that allows subclasses to implement those steps or method.

UML Diagram :-



Command Pattern :-

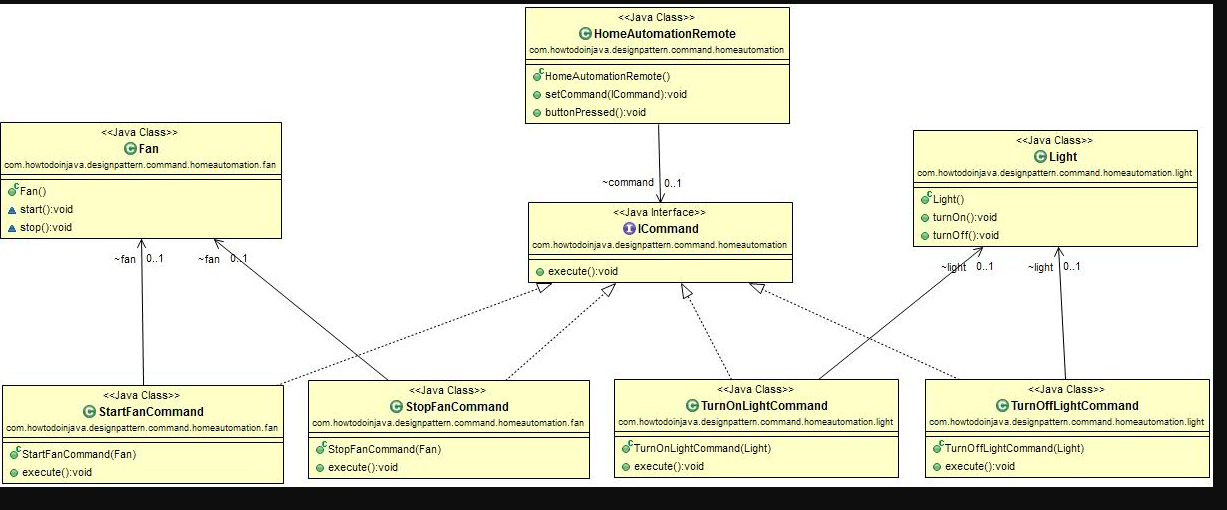
Client program is also responsible to attach receiver to the command and then command to the invoker class

Participants for command design pattern are:

* **Command interface** – for declaring an operation.
* **Concrete command classes** – which extends the Command interface, and has execute method for invoking business operation methods on receiver. It internally has reference of the receiver of command.
* **Invoker** – which is given the command object to carry out the operation.
* **Receiver** – which execute the operation.

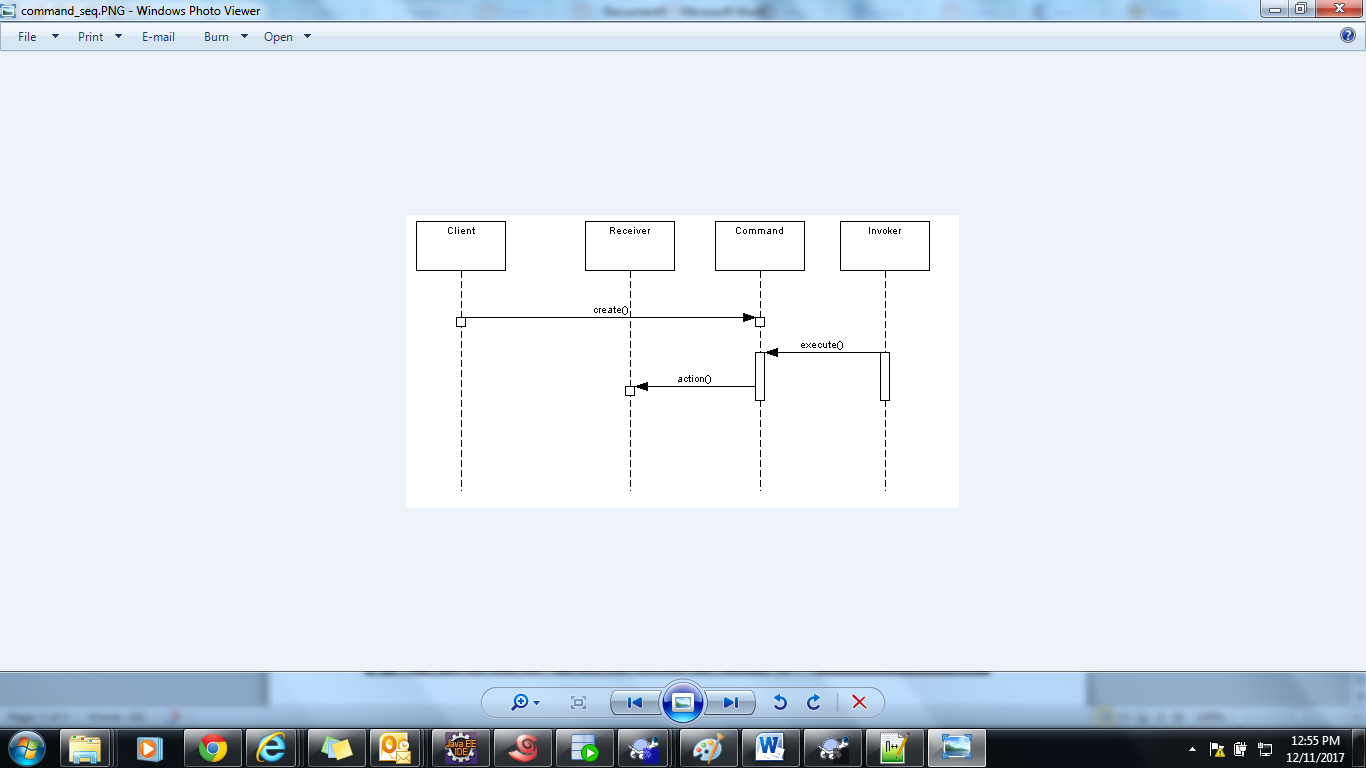
Home Automation Remote Control System:-

* ICommand interface which is the **command interface**
* Light is one of a **receiver** component. It can accept multiple commands related to Light like turn on and off
* Fan is also another type of a **receiver** component. It can accept multiple commands related to Fan like turn on and off
* HomeAutomationRemote is the **invoker** object, which asks the command to carry out the request. Here Fan on/off, Light on/off.
* StartFanCommand,StopFanCommand,TurnOffLightCommand,TurnOnLightCommand etc. are different type of **command implementations**.



In simple words, **Command design pattern** is used to *separate a request for an action from the object which actually performs the action*. This **decoupling between Invoker and Receiver** object provides a uniform ways to perform different types of actions. This decoupling is achieved using a **Command object**, which is usually an interface with methods like execute(). The Requestor or Invoker only knows about Command object, and doesn't care of the actual object which process the request, which can be different.  
  
Read more: <http://javarevisited.blogspot.com/2016/05/command-design-pattern-in-java-example-code.html#ixzz50w0inDFK>

**Command** declares an interface for all commands, providing a simple **execute()**method which asks the **Receiver**of the command to carry out an operation. The **Receiver**has the knowledge of what to do to carry out the request.  The **Invoker**holds a command and can get the **Command**to execute a request by calling the execute method. The **Client** creates **ConcreteCommand**s and sets a **Receiver**for the command. The **ConcreteCommand**defines a binding between the action and the receiver. When the **Invoker**calls execute the ConcreteCommand will run one or more actions on the Receiver.



Observer Pattern :-

The observer pattern allows for the Open Closed principle. This principle states that a class should be open for extensions without the need to change the class.

In observer design pattern multiple observer objects registers with a subject for change notification. When the state of subject changes, it notifies the observers. Objects that listen or watch for change are called observers and the object that is being watched for is called subject.

Pattern involved is also called as publish-subscribe pattern. Model view controller (MVC) architecture’s core uses the observer design pattern.

**Advantage of Observer Design Pattern in Java:**

Main advantage is **loose coupling** between objects called observer and observable. The subject only know the list of observers it don’t care about how they have their implementation.All the observers are notified by subject in a single event call as **Broadcast communication**

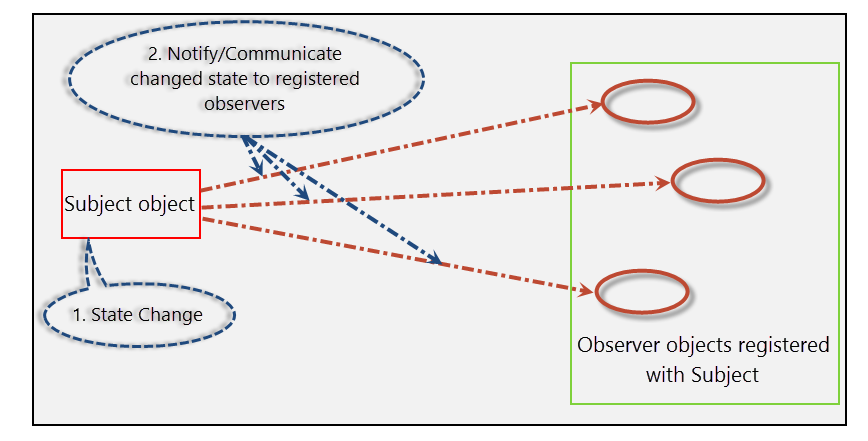
**Disadvantage of Observer Design Pattern in Java:**

          The disadvantage is that the sometime if any problem comes, [debugging](http://javarevisited.blogspot.com/2011/07/java-debugging-tutorial-example-tips.html) becomes very difficult because flow of control is implicitly between **observers** and **observable** we can predict that now observer is going to fire and if there is chain between observers then debugging become more complex.

          Another issue is Memory management because subject will hold all the reference of all the observers if we not unregister the object it can create the memory issue.

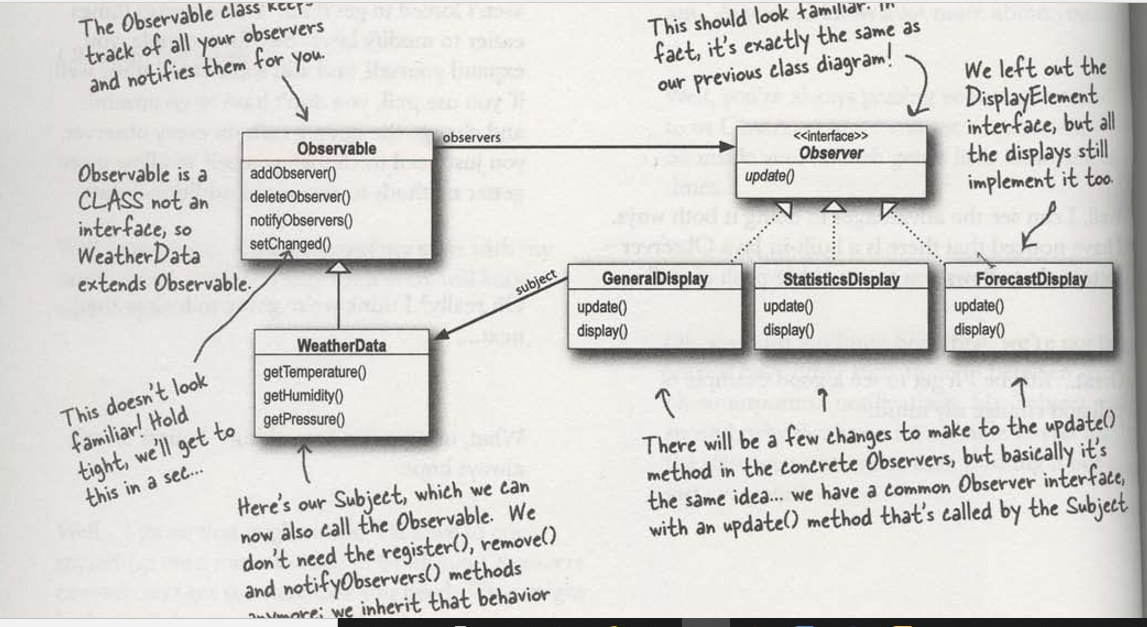
Read more: <http://javarevisited.blogspot.com/2011/12/observer-design-pattern-java-example.html#ixzz51bD5wHEO>

UML Diagram



Let’s now summarize the participants of the Observer pattern.

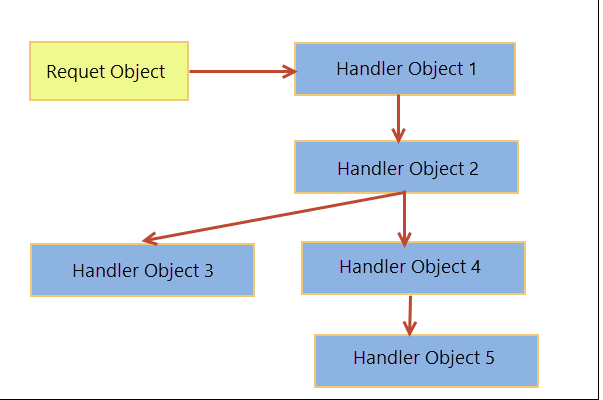
* **Subject** (Subject interface): Provides an interface to attach and detach **Observer** objects.
* **ConcreteSubject** (Product class): Implements the Subject interface. A **ConcreteSubject** sends notification to Observer objects when its state change.
* **Observer** (**Observer** interface): Provides an interface for objects that should be notified of changes in a Subject.
* **ConcreteObserver** (Bidder class): Implements Observer to receive notifications from the Subject and keep its state consistent with the state of the Subject.



Chain of Responsibility Pattern :-

1. More than one objects may handle a request, and the handler isn’t known a priori. The handler should be ascertained automatically.
2. You want to issue a request to one of several objects without specifying the receiver explicitly.
3. The set of objects that can handle a request should be specified dynamically.

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.

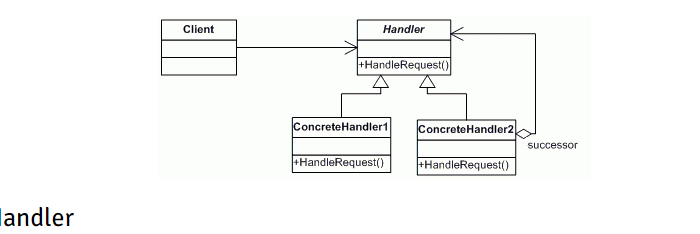


Chain of Responsibility in JDK

Try/Catch is example of chain of responsibility

The following are the usages of the Chain of Responsibility Pattern in Java.

1. java.util.logging.Logger#log()
2. javax.servlet.Filter#doFilter()

UML Diagram : 

* Client doesn’t know which part of the chain will be processing the request and it will send the request to the first object in the chain.
* Each object in the chain will have it’s own implementation to process the request, either full or partial or to send it to the next object in the chain.
* Every object in the chain should have reference to the next object in chain to forward the request to, its achieved by [java composition](https://www.journaldev.com/1325/composition-in-java-example).
* Creating the chain carefully is very important otherwise there might be a case that the request will never be forwarded to a particular processor or there are no objects in the chain who are able to handle the request. In my implementation, I have added the check for the user entered amount to make sure it gets processed fully by all the processors but we might not check it and throw exception if the request reaches the last object and there are no further objects in the chain to forward the request to. This is a design decision.

**Mediator Pattern :**

Iterator Pattern :-



Template Method Pattern :-







Decorator Pattern –

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

**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 subclassing 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.

This Decorator pattern does this without violating the **Single Responsibility Principle of SOLID** principle of object oriented programming.

Also , follows Open closed principle –as open for extension but closed for modification

### ****When to Use Decorator Pattern****

* When we wish to add, enhance or even remove the behavior or state of objects
* When we just want to modify the functionality of a single object of class and leave others unchanged

### ****Key Points of Differentiation****

* **Although Proxy and Decorator patterns have similar structures, they differ in intention**; while Proxy’s prime purpose is to facilitate ease of use or controlled access, a Decorator attaches additional responsibilities
* Both Proxy and Adapter patterns hold reference to the original object
* All the decorators from this pattern can be used recursively, infinite number of times, which is neither possible with other models

UML Diagram of Decorator Pattern

December 30, 2017

## UML Structure for Decorator Design Pattern

Let’s see the following figure that illustrates the classes and objects participating in this pattern are:

**Component (Account)**

* It is an interface for objects that can have responsibilities added to them dynamically.

**ConcreteComponent (SavingAccount)**

* It is a concrete class of component interface and it defines an object to which additional responsibilities can be attached.

**Decorator (AccountDecorator)**

* It has a reference to a Component object and defines an interface that conforms to Component’s interface.

**ConcreteDecorator (SeniorCitizen, Privilege)**

* It is concrete implementation of Decorator and it adds responsibilities to the component.

package com.doj.patterns.structural.decorator;

public interface Account {

String getTotalBenefits();

Saving Account

package com.doj.patterns.structural.decorator;

/\*\*

\* @author Dinesh.Rajput

\*

\*/

public class SavingAccount implements Account {

@Override

public String getTotalBenefits() {

return "This account has 4% interest rate with per day $5000 withdrwal limit";

}

}

**CurrentAccount.java**

package com.doj.patterns.structural.decorator;

/\*\*

\* @author Dinesh.Rajput

\*

\*/

public class CurrentAccount implements Account{

@Override

public String getTotalBenefits() {

return "There is no withdrwal limit for current account";

}

}

**AccountDecorator.java**

package com.doj.patterns.structural.decorator;

/\*\*

\* @author Dinesh.Rajput

\*

\*/

public abstract class AccountDecorator implements Account{

abstract String applyOtherBenefits();

}

**Privilege.java**

package com.doj.patterns.structural.decorator;

/\*\*

\* @author Dinesh.Rajput

\*

\*/

public class Privilege extends AccountDecorator {

Account account;

public Privilege(Account account) {

super();

this.account = account;

}

@Override

public String getTotalBenefits() {

return account.getTotalBenefits() + " other benefits are "+applyOtherBenefits();

}

@Override

String applyOtherBenefits() {

return " an accident insurance of up to $1,600 and an overdraft facility of $84";

}

}

**DecoratorPatternMain.java**

package com.doj.patterns.structural.decorator;

/\*\*

\* @author Dinesh.Rajput

\*

\*/

public class DecoratorPatternMain {

public static void main(String[] args) {

/\*Saving account with no decoration\*/

Account basicSavingAccount = new SavingAccount();

System.out.println(basicSavingAccount.getTotalBenefits());

/\*Saving account with senior citizen benefits decoration\*/

Account seniorCitizenSavingAccount = new SavingAccount();

seniorCitizenSavingAccount = new SeniorCitizen(seniorCitizenSavingAccount);

System.out.println(seniorCitizenSavingAccount.getTotalBenefits());

/\*Saving account with privilege decoration\*/

Account privilegeCitizenSavingAccount = new SavingAccount();

privilegeCitizenSavingAccount = new Privilege(privilegeCitizenSavingAccount);

System.out.println(privilegeCitizenSavingAccount.getTotalBenefits());

}

}

## Facade Design Pattern

Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.

**Facade (BankingServiceFacade)**

* This is a Facade interface knows which subsystem classes are responsible for a request. This interface is responsible to delegate client requests to appropriate subsystem objects.

**Subsystem classes (AccountService, TransferService, PaymentService)**

* These interfaces are actually subsystem functionality functionalities of Banking Process System application. These are responsible to handle process assigned by the Facade object. No interfaces in this category has reference of Facade object, even they don’t have implementation details of Facade. These are totally independent from Facade object.

Client Calls the facade interface to access subsystem.p

**Key Point to differentiate between Proxy/Adapter/Decorator**

* The proxy provides the same interface as the object it’s holding the reference to, and it doesn’t modify the data in any manner; it’s in contrast to Adapter and Decorator patterns which alter and decorate the functionalities of pre-existing instances respectively
* The Proxy usually has the information about the real subject at the compile time itself whereas Decorator and Adapter get injected at runtime, knowing only the actual object’s interface

### ****When to Use Proxy****

**Understanding how** to use a pattern is important.

**Understanding when** to use it is critical.

Let’s talk about when to use the Proxy pattern:

* **When we want a simplified version of a complex or heavy object.** In this case, we may represent it with a skeleton object which loads the original object on demand, also called as lazy initialization. This is known as the Virtual Proxy
* **When the original object is present in different address space, and we want to represent it locally**. We can create a proxy which does all the necessary boilerplate stuff like creating and maintaining the connection, encoding, decoding, etc., while the client accesses it as it was present in their local address space. This is called the Remote Proxy
* **When we want to add a layer of security to the original underlying object to provide controlled access based on access rights of the client**. This is called Protection Proxy

Proxy Pattern:



Remote Proxy-



Restful Web service –

ResponseEntity.created(location)

@ControllerAdvice

@ExceptionHandler

Hateoas – It is concept in which we send link to additional related resource to the consumer of web service

To implement there are different ways.

* 1. Specify the following dependency in pom spring-boot-starter-hateos
  2. Re

Filtering in restful webservice

Static filtering

1. Put @JsonIgnore annotation in bean property
2. Put @JsonIgnoreProperties(value={field1,field2}

Dynamic filtering – In we want field to show in one scenario and hide in another scenario That static filtering will not work. We need to implement dynamic filtering.