[**http://www.java67.com/2012/09/top-10-java-design-pattern-interview-question-answer.html**](http://www.java67.com/2012/09/top-10-java-design-pattern-interview-question-answer.html)

Design patterns are divided into three fundamental groups:

1.Behavioral

Chain of Responsibility Pattern

Command Pattern

Interpreter Pattern

Iterator Pattern

Mediator Pattern

Memento Pattern

Observer Pattern

State Pattern

Strategy Pattern

Template Pattern

Visitor Pattern

Null Object

2. Creational

[Factory Method Pattern](https://www.javatpoint.com/factory-method-design-pattern)

[Singleton Pattern](https://www.javatpoint.com/singleton-design-pattern-in-java)

[Prototype Pattern](https://www.javatpoint.com/prototype-design-pattern)

[Builder Pattern](https://www.javatpoint.com/builder-design-pattern)

[Object Pool Pattern](https://www.javatpoint.com/object-pool-pattern)

[Abstract Factory Pattern](https://www.javatpoint.com/abstract-factory-pattern)

3. Structural

1. [*Adapter:*](https://www.gofpatterns.com/structural-design-patterns/structural-patterns/adapter-pattern.php) Adapts one interface for a class into one that a client expects
2. *Bridge:* Decouple an abstraction from its implementation so that the two can vary independently
3. *Composite:* A tree structure of objects where every object has the same interface
4. *Decorator :* Add additional functionality to a class at runtime where subclassing would result in an exponential rise of new classes
5. *Facade:* Create a simplified interface of an existing interface to ease usage for common tasks
6. *Flyweight:* A high quantity of objects share a common properties object to save space
7. *Proxy:* A class functioning as an interface to another thing

**========== Strategy design pattern=====**

Strategy design pattern is one of the **behavioral design pattern**. Strategy pattern is used when we have multiple algorithm for a specific task and client decides the actual implementation to be used at runtime. One of the best example of strategy pattern is Collections.sort(ListArg,ValueComparator) method that takes Comparator parameter. Based on the different implementations of Comparator interfaces, the Objects are getting sorted in different ways

### What is decorator design pattern in Java?

### What is decorator pattern in Java? Can you give an example of Decorator pattern? Decorator pattern is another popular Java design pattern question which is common because of its heavy usage in java.io package. BufferedReader and BufferedWriter are a good example of decorator pattern in Java.

          Decorator design pattern is used to **enhance the functionality of a particular object at run- time** or dynamically.

          At the same time **other instance of same class will not be affected by this** so individual object gets the new behavior.

          Basically we wrap the original object through decorator object.

          Decorator design pattern is based on abstract classes and we derive concrete implementation from that classes,

**When to use Decorator pattern in Java**

          When sub classing is become impractical and we need large number of different possibilities to make independent object or we can say we have number of combination for an object.

          Secondly when we want to add functionality to individual object not to all object at run-time we use decorator design pattern.

### Sample Code of Decorator Design Pattern in Java

Here is complete Java program to demonstrate how you can implement decorator pattern in Java. You can use this sample code to add more features and create new classes. If you are using Eclipse IDE, just create a Java project, select that project in the package explorer and copy the code there, it will automatically create right packages and Java classes.

|  |  |
| --- | --- |
| **Sandwich.java**  **import** **java.math.BigDecimal**;  /\*\*  \* Base class for all types of Sandwich, cost method is abstract because  \* different sandwiches has different price.  \*  \* @author Javin Paul  \*/  **public** **abstract** **class** **Sandwich** {  **protected** String description = "Sandwich";    **public** String **getDescription**(){  **return** description;  }    **public** **abstract** BigDecimal **price**();  } | **WhiteBreadSandWich.java**  **import** **java.math.BigDecimal**;  /\*\*  \* A Concrete implementation of abstract Sandwich class, which represent a WhiteBread  \* Sandwich, whose price is 3.0$.  \*  \* @author Javin Paul  \*/  **public** **class** **WhiteBreadSandWich** **extends** Sandwich {    **public** **WhiteBreadSandWich**(String desc){  description = desc;  }    **@Override**  **public** BigDecimal **price**() {  **return** **new** **BigDecimal**("3.0");  }    } |
| **SandWichDecorator.java**  /\*\*  \* Base class for Decorators, this class inherit from Sandwich, so that  \* it can be of same type, which is required to pass decorators where  \* original object is expected. Later, this class will also come handy  \* to provide common functionalities to Decorators.  \*  \* @author  \*/  **public** **abstract** **class** **SandWichDecorator** **extends** Sandwich {    **@Override**  **public** **abstract** BigDecimal **price**();    } | **CheeseDecorator.java**  **import** **java.math.BigDecimal**;  /\*\*  \* A Decorator class, which adds cheese (new functionality) into Sandwich object.  \* This Decorator class modifies price() and getDescritption() method to implement  \* new behaviour.  \*  \* @author  \*/  **public** **class** **CheeseDecorator** **extends** SandWichDecorator{  Sandwich currentSandwich;    **public** **CheeseDecorator**(Sandwich sw){  currentSandwich = sw;  }    **@Override**  **public** String **getDescription**(){  **return** currentSandwich.getDescription() + ", Cheese";  }  **@Override**  **public** BigDecimal **price**() {  **return** currentSandwich.price().add(**new** BigDecimal("0.50"));  }    } |
| **SandwichMaker.java**  /\*\*  \* Test class to demonstrate How Decorator Pattern in Java work together. This class  \* first creates a Sandwich and decorates it with extra cheese. This is nice example  \* of how to provide new functionalities to an object at runtime using Decorator Pattern.  \*  \* @author Javain Paul  \*/  **public** **class** **SandwichMaker** {    **public** **static** **void** **main**(String args[]){    Sandwich mySandwich = **new** WhiteBreadSandWich("White bread Sandwich");  System.out.printf("Price of %s is $%.2f %n", mySandwich.getDescription(),  mySandwich.price());    //adding extra cheese using Decorator Patttern  mySandwich = **new** CheeseDecorator(mySandwich);  System.out.printf("Price of %s is $%.2f %n", mySandwich.getDescription(),  mySandwich.price());  }  }  **Output:**  Price of White bread Sandwich is $3.**00**  Price of White bread Sandwich, Cheese is $3.**50** | |

### Key things about Decorator Design Pattern

Now, we have seen an example of decorator pattern in Java, we can quickly summarize few important things which are worth remembering while implementing or applying decorator pattern or even to answers design pattern questions like *When to use Decorator design pattern in Java*.

1) The decorator must be of the same type of object, which they are decorating. This can be achieved either by implementing the interface of the object or by extending an abstract class of the original class.

2) Decorator is based on Composition, which means it needs an original object to decorate it. This is achieved by creating a constructor on decorator class which accepts a base type of original object. e.g. in this example constructor of CheeseDecorator accepts Sandwich object. Decorator pattern is also a good example of [Open Closed design principle](http://javarevisited.blogspot.com/2011/11/great-example-of-open-closed-design.html), which is one of the key principles from Uncle Bob's SOLID design principles.

3) Decorator class adds new functionality before or after delegating the task to the original object. In this example, the price of Decorator i.e. cheese is included after calculating the price of White Bread Sandwich.

4) Remember, Decorator design pattern only affects objects at runtime, it doesn't affect the class. You should use DECORATOR PATTERN when your intent is to add new functionality at runtime (i.e. a customer order, where you only know about order details, one it placed).

5) There is *one disadvantage of Decorator pattern* as well, it adds lots of small classes in the code base, remember the overwhelming number of classes in java.io package. Though, once you know that which classes are main classes, and which are decorators, you tend to get a better understanding of overall structure. UML diagrams certain helps in this case.

That's all on **Decorator design pattern in Java and Object oriented design**. I must say, this is one of the must know design patterns for a senior Java developers, it's general purpose and has lot's of use cases as well.

## Example of Builder Design pattern in Java RTS LOGGER iN CTS

We will use same example of creating Cake using Builder design pattern in Java. here we have [static nested builder class](http://javarevisited.blogspot.com/2011/11/static-keyword-method-variable-java.html) inside Cake which is used to create object.

**Guidelines for Builder design pattern in Java**

1) Make a static nested class called Builder inside the class whose object will be build by Builder. In this example its Cake.

2) Builder class will have exactly same set of fields as original class.

3) Builder class will expose method for adding ingredients e.g. sugar() in this example. each method will return same Builder object. Builder will be enriched with each method call.

4) Builder.build() method will copy all builder field values into actual class and return object of Item class.

5) Item class (class for which we are creating Builder) should have [private constructor](http://javarevisited.blogspot.sg/2012/03/private-in-java-why-should-you-always.html) to create its object from build() method and prevent outsider to access its constructor.

**public** **class** BuilderPatternExample {  
    
    **public** **static** **void** main(**String** args[]) {  
        
        *//Creating object using Builder pattern in java*  
        Cake whiteCake = **new**Cake.Builder().sugar(1).butter(0.5).  eggs(2).vanila(2).flour(1.5). bakingpowder(0.75).milk(0.5).build();  
        
        *//Cake is ready to eat :)*  
        **System**.out.println(whiteCake);  
    }  
}  
  
**class** Cake {  
  
    **private** **final** **double** sugar;   *//cup*  
    **private** **final** **double** butter;  *//cup*  
    **private** **final** **int** eggs;  
    **private** **final** **int** vanila;     *//spoon*  
    **private** **final** **double** flour;   *//cup*  
    **private** **final** **double** bakingpowder; *//spoon*  
    **private** **final** **double** milk;  *//cup*  
    **private** **final** **int** cherry;  
  
    **public** **static** **class** Builder {  
  
        **private** **double** sugar;   *//cup*  
        **private** **double** butter;  *//cup*  
        **private** **int** eggs;  
        **private** **int** vanila;     *//spoon*  
        **private** **double** flour;   *//cup*  
        **private** **double** bakingpowder; *//spoon*  
        **private** **double** milk;  *//cup*  
        **private** **int** cherry;  
  
        *//builder methods for setting property*  
        **public** Builder sugar(**double** cup){**this**.sugar = cup; **return** **this**; }  
        **public** Builder butter(**double** cup){**this**.butter = cup; **return** **this**; }  
        **public** Builder eggs(**int** number){**this**.eggs = number; **return** **this**; }  
        **public** Builder vanila(**int** spoon){**this**.vanila = spoon; **return** **this**; }  
        **public** Builder flour(**double** cup){**this**.flour = cup; **return** **this**; }  
        **public** Builder bakingpowder(**double** spoon){**this**.sugar = spoon; **return** **this**; }  
        **public** Builder milk(**double** cup){**this**.milk = cup; **return** **this**; }  
        **public** Builder cherry(**int** number){**this**.cherry = number; **return** **this**; }  
        
        
        *//return fully build object*  
        **public** Cake build() {  
            **return** **new** Cake(**this**);  
        }  
    }

*//private constructor to enforce object creation through builder*  
    **private** Cake(Builder builder) {  
        **this**.sugar = builder.sugar;  
        **this**.butter = builder.butter;  
        **this**.eggs = builder.eggs;  
        **this**.vanila = builder.vanila;  
        **this**.flour = builder.flour;  
        **this**.bakingpowder = builder.bakingpowder;  
        **this**.milk = builder.milk;  
        **this**.cherry = builder.cherry;         
    }  
  
    @**Override**  
    **public** **String** toString() {  
        **return** "Cake{" + "sugar=" + sugar + ", butter=" + butter + ", eggs=" + eggs + ", vanila=" + vanila + ", flour=" + flour + ", bakingpowder=" + bakingpowder + ", milk=" + milk + ", cherry=" + cherry + '}';  
  
    }   
    
}  
  
**Output:**  
Cake{sugar=0.75, butter=0.5, eggs=2, vanila=2, flour=1.5, bakingpowder=0.0, milk=0.5, cherry=0}

## Builder design pattern in Java – Pros and Cons

Live everything Builder pattern also has some disadvantages, but if you look at below, advantages clearly outnumber disadvantages of Builder design pattern. Any way here are few advantages and disadvantage of Builder design pattern for creating objects in Java.

**Advantages:**

1) more maintainable if number of fields required to create object is more than 4 or 5.

2) less error-prone as user will know what they are passing because of explicit method call.

3) more robust as only fully constructed object will be available to client.

**Disadvantages:**

1) verbose and code duplication as Builder needs to copy all fields from Original or Item class.

**When to use Builder Design pattern in Java**

Builder Design pattern is a creational pattern and should be used when number of parameter required in constructor is more than manageable usually 4 or at most 5

===============Design Pattern For AOP=========CITI Interview===

Aspect-oriented implementation of five well-known design patterns: Singleton, Observer, Command, Chain of Responsibility, and Proxy.

# Facade Pattern

## **Façade used in Java API** javax.servlet.http.HttpSession javax.servlet.http.HttpServletRequest javax.servlet.http.HttpServletResponse javax.faces.context.ExternalContext

**Benefits:**

* We can use the façade pattern to collate all the complex method calls and related code blocks and channelizes it through one single Façade class. In this way with respect to client there is only one single call. Even if we make changes to the subsystem packages / class and their logic there is no impact to the client call. In short this increases loose coupling.
* It makes easier to use and maintain creating a more structured environment and reduces dependencies between libraries or other packages.
* **Façade pattern** is one such pattern which provides a simplified interface to a set of interfaces within a system and thus it hides the complexities of the subsystem from the client.
* **When to use Façade Pattern?**
* Layering: Facade pattern can be used in JEE applications for creating a layer to abstract and unify the related interfaces in the application. Use of a facade will define an entry point to each subsystem level and thus make them communicate only through their facades; this can simplify the dependencies between them.
* Façade makes the API and libraries easier to use which is good for maintenance and readability. It can also collate and abstract various poorly designed APIs with a single simplified API.
* It also reduces dependencies of the external code on the inner working of the libraries and thus providing flexibility.

Facade Design Pattern structure

In the above structure for Façade pattern the Façade class insulates the subsystem from the client. The client only interacts with the Façade class without knowing about the subsystem classes.

*Example:*

Let’s take an example of Order processing online website. The client has placed an order without having knowledge of how internal classes are functioning. Once the order is placed the façade class layer calls the methods of the subsystems like Inventory for inventory check and Payment for processing of the payment. After processing it returns the control to the client class with the confirmation about the order being processed.

**Sequence Diagram:**

Facade Design Sequence Diagram

Stock-taking or "**inventory checking**" is the physical verification of the quantities and condition of items held in an **inventory** or warehouse.

Code Example:

Inventory.java -

public class Inventory {

public String checkInventory(String OrderId) {

return "Inventory checked";

}

}

Payment.java

public class Payment {

public String deductPayment(String orderID) {

return "Payment deducted successfully";

}

}

OrderFacade.java

public class OrderFacade {

private Payment pymt = new Payment();

private Inventory inventry = new Inventory();

public void placeOrder(String orderId) {

String step1 = inventry.checkInventory(orderId);

String step2 = pymt.deductPayment(orderId);

System.out

.println("Following steps completed:" + step1

+ " & " + step2);

}

}

Client.java

public class Client {

public static void main(String args[]){

OrderFacade orderFacade = new OrderFacade();

orderFacade.placeOrder("OR123456");

System.out.println("Order processing completed");

}

}

**Benefits:**

collect and combine (texts, information, or data).

"all the information obtained is being collated"

|  |  |
| --- | --- |
| *synonyms:* | collect, gather, accumulate, assemble; |

* We can use the façade pattern to collate all the complex method calls and related code blocks and channelizes it through one single Façade class. In this way with respect to client there is only one single call. Even if we make changes to the subsystem packages / class and their logic there is no impact to the client call. In short this increases loose coupling.
* It makes easier to use and maintain creating a more structured environment and reduces dependencies between libraries or other packages.

**Drawbacks/Consequences:**

* One of the drawback is that the subsystem methods are connected to the Façade layer. If the structure of the subsystem changes then it will require subsequent change to the Façade layer and client methods.

**Interesting points:**  
Façade pattern might be confused with mediator pattern. Mediator also abstracts the functionality of the subsystem in similar manner to façade. However there is a subtle difference between both these patterns. In case of Mediator pattern the subsystem is aware of the mediator, however in case of Façade the subsystem does not know anything about the façade. It’s a one way communication from Façade to subsystem.

**When to use Composite design Pattern in Java?**

SOM – Composite

Leaf- SOH, CASH-LEG, SEC-LEG, WRITE\_OFF\_AMMOUNT, Nostro Sweep (Cash message)

List<Command>

Command.execute();

Return: SOM messages

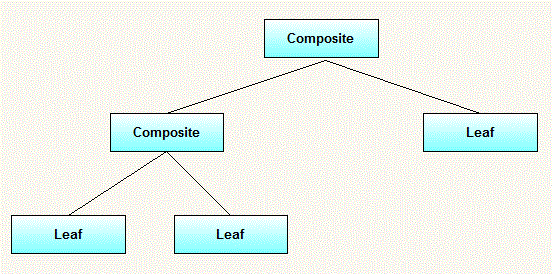
Composite design patten allows you to have a tree structure and ask each node in the tree structure to perform a task.You can take real life example of a organization.It have general managers and under general managers, there can be managers and  under managers there can be developers.Now you can set a tree structure and ask each node to perform common operation like getSalary().

As described by Gof:

"Compose objects into tree structure to represent part-whole hierarchies.Composite lets client treat individual objects and compositions of objects uniformly".

Composite design pattern treats each node in two ways-**Composite** or **leaf**.Composite means it can have other objects below it.leaf means it has no objects below it.

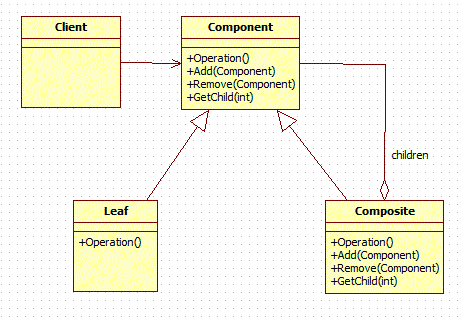
**Tree structure:**

[](http://3.bp.blogspot.com/-V9LlijmneWM/UGMwWPrgt3I/AAAAAAAAAac/vQp5-muqiDo/s1600/CompositeDesignPatternGenericHierarchy.gif)

### When to use it:

* you want to represent part-whole hierachies of objects.
* you want client to be able to ignore difference between compositions of objects and individual objects.Clients will treat all objects in the composite structure uniformly.

### UML Diagram for Composite design pattern:

[](http://2.bp.blogspot.com/-t88k0zLYMwA/UGMuUovjwzI/AAAAAAAAAaU/OA8Qoov2hZA/s1600/CompositeDesignPatternGeneric.gif)

### Elements:

* **Component**
  + declares interface for objects in composition.
  + implements deafault behaviour for the interface common to all classes as appropriate.
  + declares an interface for accessing and managing its child components.
* **Leaf**
  + represents leaf objects in the composition.A leaf has no children.
  + defines behaviour for primitive objects in the composition.
* **Composite**
  + defines behaviour for components having children.
  + stores child components.
  + implements child related operations in the component interface.
* **Client**
  + manipulates objects in the composition through the component interface.

### WorkFlow:

Client use the component class interface to interact with objects in the composition structure.if recipient is a leaf then request is handled directly.If recipient is a composite,then it usually forwards request to its child components,possibly performing additional operations before and after forwarding.

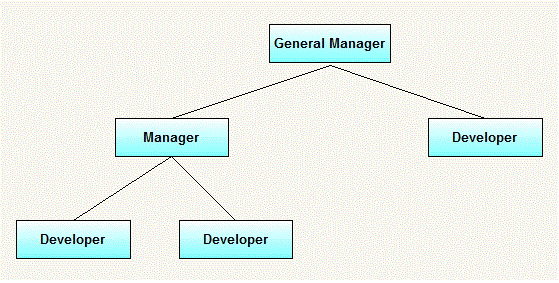
### Recursion:

What makes the Composite pattern one of the most beautiful is the power of recursion. I can explain this with the same organization example. You want to find the total salary paid to all employees of the organization. It is nothing but the salary of CEO + the salary paid to all the departments. What is the salary of a department? It is the salary of the department head + the salary of all projects. What is the total salary of a project? It is the salary of the project manager + the salary of all the project members. In short, the salary of  anything is the salary of self + the salary of all its sub groups.

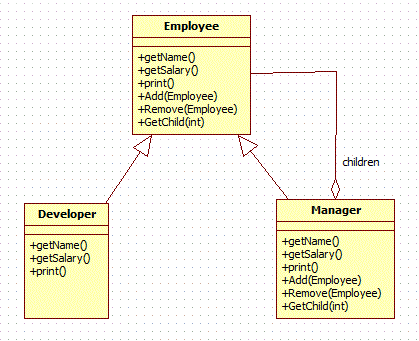
### Example:

In a small organization,there are 5 employees.At top position,there is 1 general manager.Under general manager,there are two employees,one is manager and other is developer and further manager has two developers working under him.We want to print name and salary of all employees from top to bottom.

**Tree structure for example:**

[](http://3.bp.blogspot.com/-OVvv9oMZI7k/UGMw1UvumOI/AAAAAAAAAak/EsBioEmjKSs/s1600/CompositeDesignPatternExampleHierarchy.gif)

**UML diagram for above example:**

[](http://4.bp.blogspot.com/-hxzn_ztVv5g/UGNJlLm1HPI/AAAAAAAAAbI/gkr2g8P-oP0/s1600/CompositeDesignPatternExample.gif)

Comparing from above generic elements.Our example consist of following elements.

* Manager(Composite)
* Developer(Leaf)
* Employee(Component)

#### Java code for all above classes:

First we will create component inrteface.It represents object in composition .It has all common operation that will be applicable to both manager and developer.

**Employee.java(Component) :**

package org.arpit.javapostsforlearning.designpatterns;

public interface Employee {

public void add(Employee employee);

public void remove(Employee employee);

public Employee getChild(int i);

public String getName();

public double getSalary();

public void print();

}

Now we will create manager(composite class).Key point here is that all common method delegates its operations to child objects.It has method to access and modify its children.

**Manager.java(Composite):**

package org.arpit.javapostsforlearning.designpatterns;

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

public class Manager implements Employee{

private String name;

private double salary;

public Manager(String name,double salary){

this.name = name;

this.salary = salary;

}

List<Employee> employees = new ArrayList<Employee>();

public void add(Employee employee) {

employees.add(employee);

}

public Employee getChild(int i) {

return employees.get(i);

}

public String getName() {

return name;

}

public double getSalary() {

return salary;

}

public void print() {

System.out.println("-------------");

System.out.println("Name ="+getName());

System.out.println("Salary ="+getSalary());

System.out.println("-------------");

Iterator<Employee> employeeIterator = employees.iterator();

while(employeeIterator.hasNext()){

Employee employee = employeeIterator.next();

employee.print();

}

}

public void remove(Employee employee) {

employees.remove(employee);

}

}

We will create developer class.This class is leaf node so all operations related to accessing children will be empty as it has no children.

**Developer.java(Leaf):**

package org.arpit.javapostsforlearning.designpatterns;

/\*\*

\* In this class,there are many methods which are not applicable to developer because

\* it is a leaf node.

\*/

public class Developer implements Employee{

private String name;

private double salary;

public Developer(String name,double salary){

this.name = name;

this.salary = salary;

}

public void add(Employee employee) {

//this is leaf node so this method is not applicable to this class.

}

public Employee getChild(int i) {

//this is leaf node so this method is not applicable to this class.

return null;

}

public String getName() {

return name;

}

public double getSalary() {

return salary;

}

public void print() {

System.out.println("-------------");

System.out.println("Name ="+getName());

System.out.println("Salary ="+getSalary());

System.out.println("-------------");

}

public void remove(Employee employee) {

//this is leaf node so this method is not applicable to this class.

}

}

**CompositeDesignPatternMain.java:**

package org.arpit.javapostsforlearning.designpatterns;

public class CompositeDesignPatternMain {

public static void main(String[] args) {

Employee emp1=new Developer("John", 10000);

Employee emp2=new Developer("David", 15000);

Employee manager1=new Manager("Daniel",25000);

manager1.add(emp1);

manager1.add(emp2);

Employee emp3=new Developer("Michael", 20000);

Manager generalManager=new Manager("Mark", 50000);

generalManager.add(emp3);

generalManager.add(manager1);

generalManager.print();

}

}

**Output:**

-------------

Name =Mark

Salary =50000.0

-------------

-------------

Name =Michael

Salary =20000.0

-------------

-------------

Name =Daniel

Salary =25000.0

-------------

-------------

Name =John

Salary =10000.0

-------------

-------------

Name =David

Salary =15000.0

-------------

#### Disadvantages:

* Once tree structure is defined,comosite design makes tree overly general.
* Leaf class have to create some methods which has to empty.

# Design Patterns - Prototype Pattern

Prototype pattern refers to creating duplicate object while keeping performance in mind. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

This pattern involves implementing a prototype interface which tells to create a clone of the current object. This pattern is used when creation of object directly is costly. For example, an object is to be created after a costly database operation. We can cache the object, returns its clone on next request and update the database as and when needed thus reducing database calls.

## Implementation

We're going to create an abstract class *Shape* and concrete classes extending the *Shape* class. A class *ShapeCache* is defined as a next step which stores shape objects in a *Hashtable* and returns their clone when requested.*PrototypPatternDemo*, our demo class will use *ShapeCache* class to get a *Shape* object.

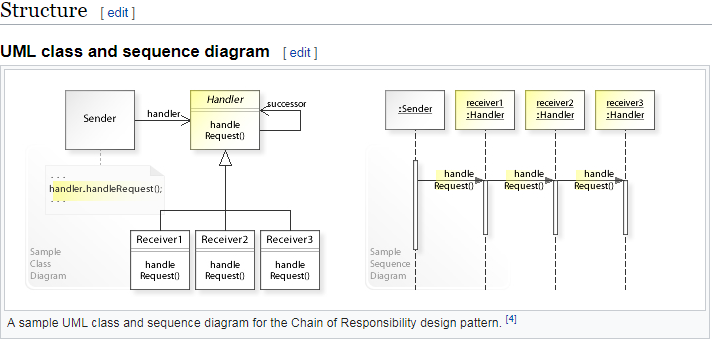
## 

**===Chain Of Responsibility design pattern=**

What solution does the Chain of Responsibility design pattern describe?

* Define a chain of receiver objects having the responsibility, depending on run-time conditions, to either handle a request or forward it to the next receiver on the chain (if any).

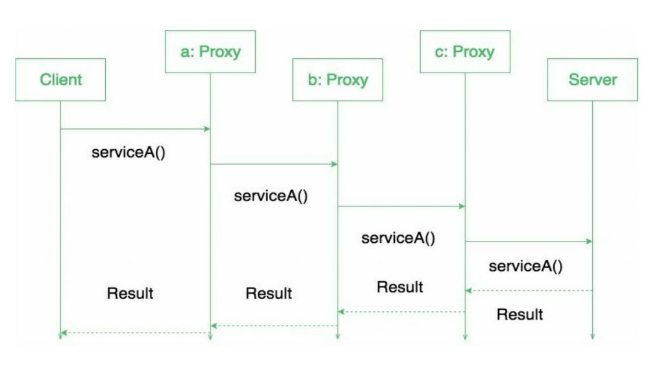
This enables to send a request to a chain of receivers without having to know which one handles the request. The request gets passed along the chain until a receiver handles the request. The sender of a request is no longer coupled to a particular receiver.



# =======Proxy Design Pattern(**Behavior**)==========

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**.

As in the decorator pattern, proxies can be chained together. The client, and each proxy, believes it is delegating messages to the real server:



**When to use this pattern?**

Proxy pattern is used when we need to create a wrapper to cover the main object’s complexity from the client.

* Proxy patterns provides the same interface from the original object but the decorator provides an enhanced interface. Decorator pattern adds additional behaviour at runtime.
* Proxy used in Java API: java.rmi.\*;

**Some Examples**

A very simple real life scenario is our college internet, which restricts few site access. The proxy first checks the host you are connecting to, if it is not part of restricted site list, then it connects to the real internet. This example is based on Protection proxies.

Lets see how it works :

|  |  |  |
| --- | --- | --- |
| **Interface of Internet** | |  | | --- | | package com.saket.demo.proxy;    public interface Internet  {      public void connectTo(String serverhost) throws Exception;  } | |
| **RealInternet.java** | package com.saket.demo.proxy;    public class RealInternet implements Internet  {      @Override      public void connectTo(String serverhost)      {          System.out.println("Connecting to "+ serverhost);      }  } |
| **ProxyInternet.java** | public class ProxyInternet implements Internet  {      private Internet internet = new RealInternet();      private static List<String> bannedSites;        static      {          bannedSites = new ArrayList<String>();          bannedSites.add("abc.com");          bannedSites.add("def.com");          bannedSites.add("ijk.com");          bannedSites.add("lnm.com");      }        @Override      public void connectTo(String serverhost) throws Exception      {          if(bannedSites.contains(serverhost.toLowerCase()))          {              throw new Exception("Access Denied");          }            internet.connectTo(serverhost);      }    } |
| **Client.java** | public class Client  {      public static void main (String[] args)      {          Internet internet = new ProxyInternet();          try          {              internet.connectTo("geeksforgeeks.org");              internet.connectTo("abc.com");          }          catch (Exception e)          {              System.out.println(e.getMessage());          }      }  } |

|  |
| --- |
|  |
| package com.saket.demo.proxy;    import java.util.ArrayList;  import java.util.List; |
| package com.saket.demo.proxy; |

As one of the site is mentioned in the banned sites, So  
Running the program will give the output :

Connecting to geeksforgeeks.org

Access Denied

**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.

I think total number of binary trees possible for n nodes is equal to Catalan number.  
C(n) = (2n)! / ((n+1)! \* n!)  
for n = 4, **14** binary trees are possible as given by catalan number.Nov 26, 2014