**JAVA DESIGN PATTERNS**

**Abstract:** “Pattern” as the name suggests, means series of events occurring in a definite order. The patterns can be found in Java and J2ee technologies also. Many a times, we find that there is a particular way of tackling a problem. This way is easy and has been used many times successfully by a number of people earlier also. This method becomes a pattern.

Learning the design patterns is a multiple step process:  
1.Acceptance  
2.Recognition  
3. Internalization

**Patterns Defined:** The patterns can be defined in many ways. You can find the definitions of patterns in many good books.

***“Design patterns are recurring solutions to design problems.”***

**Patterns:** According to commonly known practices, there are 23 design patterns in Java. These patterns are grouped under three heads:  
1. Creational Patterns  
2. Structural Patterns  
3. Behavioral Patterns

**Creational Patterns**

All the creational patterns define the best possible way in which an object can be instantiated. These describes the best way to CREATE object instances. Now everyone knows the object instance in Java can be created using a new operator.  
  
**Book book = new Book ();**

So, what’s the great stuff? Well, that’s true. The new Operator creates the instance of an object, but this is hard-coding. As I have said earlier, creating good software is difficult and so, hard coding is the last thing one should do. Also, at times the very nature of the object which is created can change according to the nature of the program. In such scenarios, we can make use of patterns to give this a more general and flexible approach.   
  
There are five types of Creational Patterns.  
1. [Factory Pattern](http://www.allapplabs.com/java_design_patterns/factory_pattern.htm)  
2. [Abstract Factory Pattern](http://www.allapplabs.com/java_design_patterns/abstract_factory_pattern.htm)  
3. [Builder Pattern](http://www.allapplabs.com/java_design_patterns/builder_pattern.htm)  
4. [Prototype Pattern](http://www.allapplabs.com/java_design_patterns/prototype_pattern.htm)  
5. [Singleton Pattern](http://www.allapplabs.com/java_design_patterns/singleton_pattern.htm)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Creational Patterns - Factory Pattern**  Factory of what? Of classes. In simple words, if we have a super class and n sub-classes, and based on data provided, we have to return the object of one of the sub-classes, we use a factory pattern.  Let’s take an example to understand this pattern.  **Example:** Let’s suppose an application asks for entering the name and sex of a person. If the sex is Male (M), it displays welcome message saying Hello Mr. <Name> and if the sex is Female (F), it displays message saying Hello Ms <Name>.  The skeleton of the code can be given here.   |  |  | | --- | --- | | public class Person { | | |  | // name string public String name; // gender : M or F private String gender;  public String getName() { return name; }  public String getGender() { return gender; } | | }// End of class | |   This is a simple class Person having methods for name and gender. Now, we will have two sub-classes, Male and Female which will print the welcome message on the screen.   |  |  | | --- | --- | | public class Male extends Person { | | |  | public Male(String fullName) { System.out.println("Hello Mr. "+fullName); } | | }// End of class | |   Also, the class Female   |  |  | | --- | --- | | public class Female extends Person { | | |  | public Female(String fullNname) { System.out.println("Hello Ms. "+fullNname); } | | }// End of class | |   Now, we have to create a client, or a SalutationFactory which will return the welcome message depending on the data provided.   |  |  | | --- | --- | | public class SalutationFactory { | | |  | public static void main(String args[]) { SalutationFactory factory = new SalutationFactory(); factory.getPerson(args[0], args[1]); }  public Person getPerson(String name, String gender) { if (gender.equals("M")) return new Male(name); else if(gender.equals("F")) return new Female(name); else return null; } | | }// End of class | |   This class accepts two arguments from the system at runtime and prints the names.  Running the program:  After compiling and running the code on my computer with the arguments Prashant and M:  java Prashant M  The result returned is: “Hello Mr. Prashant”.  When to use a Factory Pattern? The Factory patterns can be used in following cases: 1. When a class does not know which class of objects it must create. 2. A class specifies its sub-classes to specify which objects to create. 3. In programmer’s language (very raw form), you can use factory pattern where you have to create an object of any one of sub-classes depending on the data provided. | |  | | --- | |  | |

**Creational Patterns - Abstract Factory Pattern**

This pattern is one level of abstraction higher than factory pattern. This means that the abstract factory returns the factory of classes. Like Factory pattern returned one of the several sub-classes, this returns such factory which later will return one of the sub-classes.  
  
Let’s understand this pattern with the help of an example.

Suppose we need to get the specification of various parts of a computer based on which work the computer will be used for.  
  
The different parts of computer are, say Monitor, RAM and Processor. The different types of computers are PC, Workstation and Server.   
  
So, here we have an abstract base class Computer.

|  |  |
| --- | --- |
| package creational.abstractfactory;  public abstract class Computer { | |
|  | /\*\* \* Abstract method, returns the Parts ideal for \* Server \* @return Parts \*/ public abstract Parts getRAM();  /\*\* \* Abstract method, returns the Parts ideal for \* Workstation \* @return Parts \*/ public abstract Parts getProcessor();  /\*\* \* Abstract method, returns the Parts ideal for \* PC \* @return Parts \*/ public abstract Parts getMonitor(); |
| }// End of class | |

This class, as you can see, has three methods all returning different parts of computer. They all return a method called Parts. The specification of Parts will be different for different types of computers. Let’s have a look at the class Parts.

|  |  |
| --- | --- |
| package creational.abstractfactory;  public class Parts { | |
|  | /\*\* \* specification of Part of Computer, String \*/ public String specification;  /\*\* \* Constructor sets the name of OS \* @param specification of Part of Computer \*/ public Parts(String specification) { this.specification = specification; }  /\*\* \* Returns the name of the part of Computer \* \* @return specification of Part of Computer, String \*/ public String getSpecification() { return specification; } |
| }// End of class | |

And now lets go to the sub-classes of Computer. They are PC, Workstation and Server.

|  |  |
| --- | --- |
| package creational.abstractfactory;  public class PC extends Computer { | |
|  | /\*\* \* Method over-ridden from Computer, returns the Parts ideal for \* Server \* @return Parts \*/ public Parts getRAM() { return new Parts("512 MB"); }  /\*\* \* Method over-ridden from Computer, returns the Parts ideal for \* Workstation \* @return Parts \*/ public Parts getProcessor() { return new Parts("Celeron"); }  /\*\* \* Method over-ridden from Computer, returns the Parts ideal for \* PC \* @return Parts \*/ public Parts getMonitor() { return new Parts("15 inches"); } |
| }// End of class | |

|  |  |
| --- | --- |
| package creational.abstractfactory;  public class Workstation extends Computer { | |
|  | /\*\* \* Method over-ridden from Computer, returns the Parts ideal for \* Server \* @return Parts \*/ public Parts getRAM() { return new Parts("1 GB"); }  /\*\* \* Method over-ridden from Computer, returns the Parts ideal for \* Workstation \* @return Parts \*/ public Parts getProcessor() { return new Parts("Intel P 3"); }  /\*\* \* Method over-ridden from Computer, returns the Parts ideal for \* PC \* @return Parts \*/ public Parts getMonitor() { return new Parts("19 inches"); } |
| }// End of class | |

|  |  |
| --- | --- |
| package creational.abstractfactory;  public class Server extends Computer{ | |
|  | /\*\* \* Method over-ridden from Computer, returns the Parts ideal for \* Server \* @return Parts \*/ public Parts getRAM() { return new Parts("4 GB"); }  /\*\* \* Method over-ridden from Computer, returns the Parts ideal for \* Workstation \* @return Parts \*/ public Parts getProcessor() { return new Parts("Intel P 4"); }  /\*\* \* Method over-ridden from Computer, returns the Parts ideal for \* PC \* @return Parts \*/ public Parts getMonitor() { return new Parts("17 inches"); } |
| }// End of class | |

Now let’s have a look at the Abstract factory which returns a factory “Computer”. We call the class ComputerType.

|  |  |  |
| --- | --- | --- |
| package creational.abstractfactory;  /\*\* \* This is the computer abstract factory which returns one \* of the three types of computers. \* \*/ public class ComputerType { | | |
|  | private Computer comp;  public static void main(String[] args) { | |
|  |  | ComputerType type = new ComputerType();  Computer computer = type.getComputer("Server"); System.out.println("Monitor: "+computer.getMonitor().getSpecification()); System.out.println("RAM: "+computer.getRAM().getSpecification()); System.out.println("Processor: "+computer.getProcessor().getSpecification()); |
|  | } |  |
|  |  | /\*\* \* Returns a computer for a type \* \* @param computerType String, PC / Workstation / Server \* @return Computer \*/ |
|  | public Computer getComputer(String computerType) { | |
|  |  | if (computerType.equals("PC")) comp = new PC(); else if(computerType.equals("Workstation")) comp = new Workstation(); else if(computerType.equals("Server")) comp = new Server();  return comp; |
|  | } |  |
| }// End of class | | |

Running this class gives the output as this:

Monitor: 17 inches  
RAM: 4 GB  
Processor: Intel P 4.

When to use Abstract Factory Pattern?  
One of the main advantages of Abstract Factory Pattern is that it isolates the concrete classes that are generated. The names of actual implementing classes are not needed to be known at the client side. Because of the isolation, you can change the implementation from one factory to another.

**Creational Patterns - Singleton Pattern**

This is one of the most commonly used patterns. There are some instances in the application where we have to use just one instance of a particular class. Let’s take up an example to understand this.

A very simple example is say Logger, suppose we need to implement the logger and log it to some file according to date time. In this case, we cannot have more than one instances of Logger in the application otherwise the file in which we need to log will be created with every instance.

We use Singleton pattern for this and instantiate the logger when the first request hits or when the server is started.

|  |  |
| --- | --- |
| package creational.singleton;  import org.apache.log4j.Priority;  import java.text.SimpleDateFormat; import java.util.GregorianCalendar; import java.util.Properties; import java.io.InputStream; import java.io.FileOutputStream; import java.io.PrintStream; import java.io.IOException;  public class Logger { | |
|  | private String fileName; private Properties properties; private Priority priority;  /\*\* \* Private constructor \*/ private Logger() { logger = this; }  /\*\* \* Level of logging, error or information etc \* \* @return level, int \*/ public int getRegisteredLevel() { int i = 0; try { InputStream inputstream = getClass().getResourceAsStream("Logger.properties"); properties.load(inputstream); inputstream.close(); i = Integer.parseInt(properties.getProperty("\*\*logger.registeredlevel\*\*")); if(i < 0 || i > 3) i = 0; } catch(Exception exception) { System.out.println("Logger: Failed in the getRegisteredLevel method"); exception.printStackTrace(); } return i; } /\*\* \* One file will be made daily. So, putting date time in file \* name. \* \* @param gc GregorianCalendar \* @return String, name of file \*/ private String getFileName(GregorianCalendar gc) { SimpleDateFormat dateFormat1 = new SimpleDateFormat("dd-MMM-yyyy"); String dateString = dateFormat1.format(gc.getTime()); String fileName = "C:\\prashant\\patterns\\log\\PatternsExceptionLog-" + dateString + ".txt"; return fileName; }  /\*\* \* A mechanism to log message to the file. \* \* @param p Priority \* @param message String \*/ public void logMsg(Priority p, String message) { try { GregorianCalendar gc = new GregorianCalendar(); String fileName = getFileName(gc); FileOutputStream fos = new FileOutputStream(fileName, true); PrintStream ps = new PrintStream(fos); SimpleDateFormat dateFormat2 = new SimpleDateFormat("EEE, MMM d, yyyy 'at' hh:mm:ss a"); ps.println("<"+dateFormat2.format(gc.getTime())+">["+message+"]"); ps.close(); } catch (IOException ie) { ie.printStackTrace(); } } /\*\* \* this method initialises the logger, creates an object \*/ public static void initialize() { logger = new Logger(); }  // singleton - pattern private static Logger logger; public static Logger getLogger() { return logger; } |
| }// End of class | |

**Difference between static class and static method approaches:**  
One question which a lot of people have been asking me. What’s the difference between a singleton class and a static class? The answer is static class is one approach to make a class “Singleton”.

We can create a class and declare it as “final” with all the methods “static”. In this case, you can’t create any instance of class and can call the static methods directly.

Example:

final class Logger {  
//a static class implementation of Singleton pattern  
static public void logMessage(String s) {  
System.out.println(s);  
}  
}// End of class

//==============================  
public class StaticLogger {  
public static void main(String args[]) {  
Logger.logMessage("This is SINGLETON");  
}  
}// End of class

The advantage of this static approach is that it’s easier to use. The disadvantage of course is that if in future you do not want the class to be static anymore, you will have to do a lot of recoding.

**Creational Patterns - Builder Pattern**

Builder, as the name suggests builds complex objects from simple ones step-by-step. It separates the construction of complex objects from their representation.

Let’s take a non-software example for this. Say, we have to plan for a children meal at a fast food restaurant. What is it comprised of? Well, a burger, a cold drink, a medium fries and a toy.

This is common to all the fast food restaurants and all the children meals. Here, what is important? Every time a children’s meal is ordered, the service boy will take a burger, a fries, a cold drink and a toy. Now suppose, there are 3 types of burgers available. Vegetable, Fish and Chicken, 2 types of cold drinks available. Cola and Orange and 2 types of toys available, a car and a doll.

So, the order might be a combination of one of these, but the process will be the same. One burger, one cold drink, one fries and one toy. All these items are placed in a paper bag and is given to the customer.

Now let’s see how we can apply software to this above mentioned example.

The whole thing is called a children meal. Each of the four burger, cold drink, fries and toy are items in the meal. So, we can say, there is an interface Item having two methods, pack() and price().

Let’s take a closer look at the interface Item:  
  
**Item.java**

|  |  |
| --- | --- |
| package creational.builder;  public interface Item { | |
|  | /\*\* \* pack is the method, as every item will be packed \* in a different way. \* E.g.:- The burger will be packed as wrapped in a paper \* The cold drink will be given in a glass \* The medium fries will be packed in a card box and \* The toy will be put in the bag straight. \* The class Packing is an interface for different types of \* for different Items. \*/  public Packing pack();  /\*\* \* price is the method as all the items \* burger, cold drink, fries will have a price. \* The toy will not have any direct price, it will \* be given free with the meal. \* \* The total price of the meal will be the combined \* price of the three items. \* \* @return price, int in rupees. \*/  public int price(); |
| }// End of class | |

So, we must now have a class defined for each of the items, as burger, toy, cold drink and fries. All these will implement the Item interface.

Lets start with Burger:  
  
**Burger.java**

|  |  |
| --- | --- |
| package creational.builder; /\*\* \* The class remains abstract as price method will be implemented \* according to type of burger. \* @see price() \*  \*/ public abstract class Burger implements Item { | |
|  | /\*\* \* A burger is packed in a wrapper. Its wrapped \* in the paper and is served. The class Wrapper is  \* sub-class of Packing interface. \* @return new Wrapper for every burger served. \*/ public Packing pack() { return new Wrapper(); }  /\*\* \* This method remains abstract and cannot be \* given an implementation as the real implementation \* will lie with the type of burger. \* \* E.g.:- Veg Burger will have a different price from \* a fish burger. \* \* @return price, int. \*/ public abstract int price(); |
| }// End of class | |

The class Burger can be further extended to VegBurger, FishBurger, ChickenBurger etc. These classes will each implement the price() method and return a price for each type of burger. I, in this example have given the implementation for VegBurger class.   
  
**VegBurger.java**

|  |  |
| --- | --- |
| package creational.builder;  /\*\* \* The implementation of price method. \*/ public class VegBurger extends Burger { | |
|  | /\*\* \* This is the method implementation from \* the super class Burger. \* \* @return price of a veg burger in rupees. \*/ public int price() { return 39; } |
| }// End of class | |

Let’s concentrate on other items now. I, here for explanation purpose will give another item Fries.  
  
**Fries.java**

|  |  |
| --- | --- |
| package creational.builder;  /\*\* \* Implements the Item interface. \*/ public class Fries implements Item { | |
|  | /\*\* \* Packing in which fries are served. \* \* @return new Packing for every fries. \* Envelop is a packing in which fries are given \*/ public Packing pack() { return new Envelop(); }  /\*\* \* Price of the medium fries. \* \* @return int , price of medium fries in rupees \*/ public int price() { return 25; } |
| }// End of class | |

Now, let’s see the Builder class, MealBuilder. This class is the one which serves the Children’s meal.  
  
**MealBuilder.java**

|  |  |
| --- | --- |
| package creational.builder;  /\*\* \* Main builder class which builds the entire meal \* for the customers \*/ public class MealBuilder { | |
|  | public Packing additems() { Item[] items = {new VegBurger(), new Fries(), new Cola(), new Doll()}  return new MealBox().addItems(items); }  public int calculatePrice() { int totalPrice = new VegBurger().price() + new Cola().price() + new Fries().price() + new Doll().price();  return totalPrice; } |
| }// End of class | |

This class gives the total meal and also presents the total price. Here, we have abstracted the price calculation and meal package building activity from the presentation, which is a meal box. The Builder pattern hides the internal details of how the product is built.

Each builder is independent of others. This improves modularity and makes the building of other builders easy.

Because, each builder builds the final product step by step, we have more control on the final product.

**Structural Patterns**

Structural Patterns describe how objects and classes can be combined to form larger structures. The difference between class patterns and object patterns is that class patterns describe abstraction with the help of inheritance and how it can be used to provide more useful program interface. Object patterns, on other hand, describe how objects can be associated and composed to form larger, more complex structures.

There are seven structural patterns described. They are as follows:

Patterns.  
1. [Adapter Pattern](http://www.allapplabs.com/java_design_patterns/adapter_pattern.htm)  
2. [Bridge Pattern](http://www.allapplabs.com/java_design_patterns/bridge_pattern.htm)  
3. [Composite Pattern](http://www.allapplabs.com/java_design_patterns/composite_pattern.htm)  
4. [Decorator Pattern](http://www.allapplabs.com/java_design_patterns/decorator_pattern.htm)  
5. [Facade Pattern](http://www.allapplabs.com/java_design_patterns/facade_pattern.htm)  
6. [Flyweight Pattern](http://www.allapplabs.com/java_design_patterns/flyweight_pattern.htm)  
7. [Proxy Pattern](http://www.allapplabs.com/java_design_patterns/proxy_pattern.htm)  
  
I will describe each one of these in details

**Structural Patterns - Adapter Pattern**

The Adapter pattern is used so that two unrelated interfaces can work together. The joining between them is called an Adapter. This is something like we convert interface of one class into interface expected by the client. We do that using an Adapter.   
  
Let’s try and understand this with the help of an example. Again, I will like to take a general example. We all have electric sockets in our houses of different sizes and shapes. I will take an example of a socket of 15 Ampere. This is a bigger socket and the other one which is smaller is of 5 Ampere. A 15 Amp plug cannot fit into a 5 Amp socket. Here, we will use an Adapter. The adapter can be called a connector here. The connector connects both of these and gives output to the client plug which is of 5 Amp.  
  
The Adapter is something like this. It will be having the plug of suitable for 15 Amp and a socket suitable for a 5 Amp plug. So, that the 5 Amp plug which here is the client can fit in and also the server which here is the 15 Amp socket can give the output.  
  
Let’s try and convert the same example into a software program. How do we do this? Let’s try and understand the problem once more. We have a 5 Amp plug and want a 5 Amp socket so that it can work. We DO NOT have a 5 Amp socket, what we have is a 15 Amp socket in which the 5 Amp plug cannot fit. The problem is how to cater to the client without changing the plug or socket.  
  
The Adapter Pattern can be implemented in two ways, by Inheritance and by Composition.  
  
Here is the example of Adapter by Inheritance:  
  
Let’s say there is a socket interface.  
  
**Socket.java**

|  |  |
| --- | --- |
| package structural.adapter.inheritance; /\*\* \* The socket class has a specs for 15 AMP. \*/ public interface Socket { | |
|  | /\*\* \* This method is used to match the input to be \* given to the Plug \* \* @return Output of the Plug (Client) \*/ public String getOutput(); |
| }// End of interface | |

And there is a class Plug which wants the input of 5 AMP. This is the client.  
  
**Plug.java**

|  |  |
| --- | --- |
| package structural.adapter.inheritance; /\*\* \* The input for the plug is 5 AMP. which is a \* mismatch for a 15 AMP socket. \* \* The Plug is the client. We need to cater to \* the requirements of the Plug. \*/ public class Plug { | |
|  | private String specification = "5 AMP";  public String getInput() { return specification; } |
| }// End of class | |

Finally, there will be an adapter class. This will inherit the socket and give output for Plug.  
  
**ConnectorAdapter.java**

|  |  |
| --- | --- |
| package structural.adapter.inheritance; /\*\* \* ConnectorAdapter has is the connector between \* the socket and plug so as to make the interface \* of one system to suit the client. \*/ public class ConnectorAdapter implements Socket { | |
|  | /\*\* \* Method coming from the interface \* Socket which we have to make to \* fit the client plug \* \* @return Desired output of 5 AMP \*/ public String getOutput() { Plug plug = new Plug(); String output = plug.getInput(); return output; } |
| }// End of class | |

This class implements the getOutput() method of Socket and sets it to fit the client output.

Similarly, let’s consider the Association and Composition of objects by which Adapter can be implemented.

The class Socket gives the 15 AMP output.

**Socket.java**

|  |  |
| --- | --- |
| package structural.adapter.composition; /\*\* \* Class socket giving the 15 AMP output. \*/ public class Socket { | |
|  | /\*\* \* Output of 15AMP returned. \* \* @return Value of output from socket \*/ public String getOutput() { return "15 AMP"; } |
| }// End of class | |

There is an interface Plug.java which has a method getInput(). This is the client and we need to adapt the output for this input which is 5 AMP.

**Plug.java**

|  |  |
| --- | --- |
| package structural.adapter.composition; /\*\* \* The input for the plug is 5 AMP. which is a \* mismatch for a 15 AMP socket. \* \* The Plug is the client. We need to cater to \* the requirements of the Plug. \*/ public interface Plug { | |
|  | public String getInput(); |
| }// End of class | |

Plug5AMP is the implementation of Plug which requires 5 AMP of input.

**Plug5AMP.java**

|  |  |
| --- | --- |
| package structural.adapter.composition;  public class Plug5AMP implements Plug { | |
|  | /\*\* \* Get the input of client i.e. Plug \* \* @return 5 AMP \*/ public String getInput() { return "5 AMP"; } |
| }// End of class | |

The Adapter here takes output from the Socket. If the output is what is needed, it gives it to the Plug else, it overrides the value and returns the adapter output.  
 **ConnectorAdapter.java**

|  |  |
| --- | --- |
| package structural.adapter.composition; /\*\* \* Using composition \*/ public class ConnectorAdapter { | |
|  | Plug5AMP plug5;  public ConnectorAdapter(Plug5AMP plug) { this.plug5 = plug; }  public static void main(String[] args) { // Taking output from the Socket Socket socket = new Socket(); String outputFromSocket = socket.getOutput();  // Giving away input to the Plug ConnectorAdapter adapter = new ConnectorAdapter(new Plug5AMP()); String inputToPlug = adapter.getAdapterOutput(outputFromSocket); System.out.println("New output by adapter is: "+inputToPlug); }  public String getAdapterOutput(String outputFromScoket) { /\* \* if output is same, return \*/ if (outputFromScoket.equals(plug5.getInput())) { return outputFromScoket; } /\* \* Else, override the value by adapterOutput \*/ else { String adapterOutput = plug5.getInput(); return adapterOutput; } |
| }// End of class | |

This is how the Adapter pattern works. When one interface cannot be changed and has to be suited to the again cannot-be-changed client, an adapter is used so that both the interfaces can work together.

**Structural Patterns - Bridge Pattern**

The Bridge Pattern is used to separate out the interface from its implementation. Doing this gives the flexibility so that both can vary independently.

The best example for this is like the electric equipments you have at home and their switches. For e.g., the switch of the fan. The switch is the interface and the actual implementation is the Running of the fan once its switched-on. Still, both the switch and the fan are independent of each other. Another switch can be plugged in for the fan and this switch can be connected to light bulb.

Let’s see how we can convert this into a software program. Switch is the interface having two functions, switchOn() and switchOff().

Here is the sample code for Switch.  
  
**Switch.java**

|  |  |
| --- | --- |
| package structural.bridge; /\*\* \* Just two methods. on and off. \*/ public interface Switch { | |
|  | // Two positions of switch. public void switchOn(); public void switchOff(); |
| }// End of interface | |

This switch can be implemented by various devices in house, as Fan, Light Bulb etc. Here is the sample code for that.  
  
**Fan.java**

|  |  |
| --- | --- |
| package structural.bridge; /\*\* \* Implement the switch for Fan \*/ public class Fan implements Switch { | |
|  | // Two positions of switch. public void switchOn() { System.out.println("FAN Switched ON"); }  public void switchOff() { System.out.println("FAN Switched OFF"); } |
| }// End of class | |

And implementation as Bulb.  
  
**Bulb.java**

|  |  |
| --- | --- |
| package structural.bridge; /\*\* \* Implement the switch for Fan \*/ public class Bulb implements Switch { | |
|  | // Two positions of switch. public void switchOn() { System.out.println("BULB Switched ON"); }  public void switchOff() { System.out.println("BULB Switched OFF"); } |
| }// End of class | |

Here, we can see, that the interface Switch can be implemented in different ways. Here, we can easily use Switch as an interface as it has only two functions, on and off. But, there may arise a case where some other function be added to it, like change() (change the switch). In this case, the interface will change and so, the implementations will also changed, for such cases, you should use the Switch as abstract class. This decision should be made earlier to implementation whether the interface should be interface or abstract class.

Structural Patterns - Composite Pattern

In developing applications, we come across components which are individual objects and also can be collection of objects. Composite pattern can represent both the conditions. In this pattern, you can develop tree structures for representing part-whole hierarchies.

TThe most common example in this pattern is of a company’s employee hierarchy. We here will also take the same example.

The employees of a company are at various positions. Now, say in a hierarchy, the manager has subordinates; also the Project Leader has subordinates, i.e. employees reporting to him/her. The developer has no subordinates.

So, let’s have a look at the class Employee: This is a simple class with getters and setters for attributes as name, salary and subordinates.

Employee.java

package structural.composite;

import java.util.Vector;

public class Employee {

private String name;

private double salary;

private Vector subordinates;

public Vector getSubordinates() {

return subordinates;

}

public void setSubordinates(Vector subordinates) {

this.subordinates = subordinates;

}

// constructor

public Employee(String name, double sal) {

setName(name);

setSalary(sal);

subordinates = new Vector();

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public double getSalary() {

return salary;

}

public void setSalary(double salary) {

this.salary = salary;

}

public void add(Employee e) {

subordinates.addElement(e);

}

public void remove(Employee e) {

subordinates.remove(e);

}

}// End of interface

Next we, fill up the tree. You can make a class to access the class Employee and try filling up the tree like this:

/\*\*

\* This will add employess to the tree. The boss, is PM

\* and has subordinates.

\*/

private void addEmployeesToTree() {

CFO = new Employee("CFO", 30000);

Employee headFinance1 = new Employee("Head Finance. North Zone", 20000);

Employee headFinance2 = new Employee("Head Finance. West Zone", 22000);

Employee accountant1 = new Employee("Accountant1", 10000);

Employee accountant2 = new Employee("Accountant2", 9000);

Employee accountant3 = new Employee("Accountant3", 11000);

Employee accountant4 = new Employee("Accountant4", 12000);

CFO.add(headFinance1);

CFO.add(headFinance2);

headFinance1.add(accountant1);

headFinance1.add(accountant4);

headFinance2.add(accountant2);

headFinance2.add(accountant3);

}// End of class

Once we have filled the tree up, now we can get the tree for any employee and find out whether that employee has subordinates with the following condition.

Vector subOrdinates = emp.getSubordinates();

if (subOrdinates.size() != 0)

getTree(subOrdinates);

else

System.out.println("No Subordinates for the Employee: "+emp.getName());

Thus the Composite pattern allows you to create a tree like structure for simple and complex objects so they appear the same to the client.

Behavioral Patterns

Behavioral patterns are those which are concerned with interactions between the objects. The interactions between the objects should be such that they are talking to each other and still are loosely coupled. The loose coupling is the key to n-tier architectures. In this, the implementation and the client should be loosely coupled in order to avoid hard-coding and dependencies.

The behavioral patterns are:

Patterns.

1. Chain of Resposibility Pattern

2. Command Pattern

3. Interpreter Pattern

4. Iterator Pattern

5. Mediator Pattern

6. Momento Pattern

7. Observer Pattern

8. State Pattern

9. Strategy Pattern

10. Template Pattern

11. Visitor Pattern

I will describe each one of these in details

**Behavioral Patterns - Chain of Responsibility Pattern**

The chain of responsibility pattern is based on the same principle as written above. It decouples the sender of the request to the receiver. The only link between sender and the receiver is the request which is sent. Based on the request data sent, the receiver is picked. This is called “data-driven”. In most of the behavioral patterns, the data-driven concepts are used to have a loose coupling.  
  
The responsibility of handling the request data is given to any of the members of the “chain”. If the first link of the chain cannot handle the responsibility, it passes the request data to the next level in the chain, i.e. to the next link. For readers, familiar with concepts of Java, this is similar to what happens in an Exception Hierarchy. Suppose the code written throws an ArrayIndexOutOfBoundsException. Now, this exception is because of some bug in coding and so, it gets caught at the correct level. Suppose, we have an application specific exception in the catch block. This will not be caught by that. It will find for an Exception class and will be caught by that as both the application specific exceptions and the ArrayIndexOutOfBoundsException are sub-classes of the class Exception.  
  
Once get caught by the exception, which is the base class, it will then not look for any other exception. This is precisely the reason why, we get an “Exception is unreachable” message when we try to add a catch block with the exception below the parent exception catch block.  
  
So, in short, the request rises in hierarchy till some object takes responsibility to handle this request.   
  
It’s the same mechanism used for multi-level filtration. Suppose, we have a multi level filter and gravel of different sizes and shapes. We need to filter this gravel of different sizes to approx size categories. We will put the gravel on the multi-level filtration unit, with the filter of maximum size at the top and then the sizes descending. The gravel with the maximum sizes will stay on the first one and rest will pass, again this cycle will repeat until, the finest of the gravel is filtered and is collected in the sill below the filters. Each of the filters will have the sizes of gravel which cannot pass through it. And hence, we will have approx similar sizes of gravels grouped.  
  
Let’s apply the same example in the form of code.  
  
First, let’s talk about the request data. In this case, it is the gravel. We call it Matter. It has size and quantity. Now, the size determines whether it matches the size of filter or not and the quantity tells how much matter is below the size.  
  
**Matter.java**

|  |  |
| --- | --- |
| package bahavioral.chainofresponsibility;   public class Matter { | |
|  | // size of matter private int size; // quantity private int quantity;  /\*\* \* returns the size \*/ public int getSize() { return size;  }  /\*\* \* sets the size \*/ public void setSize(int size) { this.size = size;  }  /\*\* \* returns the quantity \*/ public int getQuantity() { return quantity;  }  /\*\* \* sets the quantity \*/ public void setQuantity(int quantity) { this.quantity = quantity;  } |
| }// End of class | |

The next thing is now the base class. This base class in our case is Sill. Nothing escapes the Sill. All the matter is collected in the sill. Everything which cannot be filtered gets collected in the Sill. Like all the requests which cannot be handled at a lower level rise to higher level and are handled at the highest level.  
  
**Sill.java**

|  |  |
| --- | --- |
| package bahavioral.chainofresponsibility;  /\*\* \* Sill.java \*  \* This is the base class, you can say, which collects everything  \* and nothing passes this. Whatever matter is remaining, and is  \* still not filtered, is collected here. \*/ public class Sill { | |
|  | /\*\* \* Method collect. \* Everything is collected here. \*/ public void collect(Matter gravel) {  } |
| }// End of class | |

And of course, the next class will be the chain. In the example, I have just created one single class called Filter1. This class extends from the Sill. And the chain grows on. Every class like Filter2, Filter3 etc extends from Filter1, Filter2 and so on.  
  
**Filter1.java**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| package bahavioral.chainofresponsibility;  /\*\* \* This is a filter. This filters out the gravel and \* passes the rest to the next level. \*/ public class Filter1 extends Sill { | | | | |
|  | private int size;  public Filter1(int size) { this.size = size; }  /\*\* \* over-ridden method from base class \*/ public void collect(Matter gravel) { | | | |
|  |  | // for the entire quantity of matter for(int i = 0; i < gravel.getQuantity(); i++) { | | |
|  |  |  | // if gravel size is less than size of filter,  // the gravel will pass to the next level. | |
|  |  |  |  | if(gravel.getSize() < size) { super.collect(gravel); } else { //collect here. that means, only matter with less  // size will pass... } |
|  |  | } |  |  |
|  | } |  |  |  |
| }// End of class | | | | |

This is how, this pattern works. Based on principles of decoupling, the pattern is totally data-driven. The famous example is the Exception hierarchy.  
  
The other advantage is distribution of responsibilities. There can be such a scenario when none of the objects in the chain can handle the request. In this case, the chain will discard the request. The basic object can also be an interface depending on needs.

**Behavioral Patterns - Command Pattern**

This is another of the data-driven pattern. The client invokes a particular module using a command. The client passes a request, this request gets propagated as a command. The command request maps to particular modules. According to the command, a module is invoked.   
  
This pattern is different from the Chain of Responsibility in a way that, in the earlier one, the request passes through each of the classes before finding an object that can take the responsibility. The command pattern however finds the particular object according to the command and invokes only that one.  
  
It’s like there is a server having a lot of services to be given, and on Demand (or on command), it caters to that service for that particular client.  
  
A classic example of this is a restaurant. A customer goes to restaurant and orders the food according to his/her choice. The waiter/ waitress takes the order (command, in this case) and hands it to the cook in the kitchen. The cook can make several types of food and so, he/she prepares the ordered item and hands it over to the waiter/waitress who in turn serves to the customer.  
  
Let’s have a look at this example with Java code.  
  
First thing is the Order. The order is made of command which the customer gives the waiter.  
  
**Order.java**

|  |  |
| --- | --- |
| package bahavioral.command;  /\*\* \* Order.java \* This is the command. The customer orders and \* hands it to the waiter. \*/ public class Order { | |
|  | private String command;  public Order(String command) { this.command = command;  } |
| }// End of class | |

The other thing is the waiter who takes the order and forwards it to the cook.  
  
**Waiter.java**

|  |  |
| --- | --- |
| package bahavioral.command;  /\*\* \* A waiter is associated with multiple customers and multiple orders \*/ public class Waiter { | |
|  | public Food takeOrder(Customer cust, Order order) {  Cook cook = new Cook(); Food food = cook.prepareOrder(order, this); return food; } |
| }// End of class | |

The waiter calls the prepareFood method of the cook who in turn cooks.  
 **Cook.java**

|  |  |
| --- | --- |
| package bahavioral.command;  public class Cook { | |
|  | public Food prepareOrder(Order order, Waiter waiter) { Food food = getCookedFood(order); return food; }  public Food getCookedFood(Order order) { Food food = new Food(order); return food; } |
| }// End of class | |

Now, here, the waiter takes command and wraps it in an order, the order is associated to a particular customer. For, the cook, the order is associated to a cook and also Food is associated to the Order.

The order is an object which depends on the command. The food item will change as soon as the command changes. This is loose-coupling between the client and the implementation.

Behavioral Patterns - Interpreter Pattern

The Interpreter Pattern defines a grammatical representation for a language and an interpreter to interpret the grammar. The best example you can get for this is Java itself which is an interpreted language. It converts the code written in English to a byte code format so as to make possible for all the operating systems to understand it. This quality of it makes it platform independent.

The development of languages can be done when you find different cases but, somewhat similar, it is advantageous to use a simple language which can be interpreted by the system and can deal with all these cases at the same time.

To make this interpreter clearer, let’s take an example. The “musical notes” is an “Interpreted Language”. The musicians read the notes, interpret them according to “Sa, Re, Ga, Ma…” or “Do, Re, Me… ” etc and play the instruments, what we get in output is musical sound waves. Think of a program which can take the Sa, Re, Ga, Ma etc and produce the sounds for the frequencies.

For Sa, the frequency is 256 Hz, similarly, for Re, it is 288Hz and for Ga, it is 320 Hz etc etc…

In this, case, we need these values set somewhere so, that when the system encounters any one of these messages, we can just send the related frequency to the instrument playing the frequency.

We can have it at one of the two places, one is a constants file, “token=value” and the other one being in a properties file. The properties file can give us more flexibility to change it later if required.

This is how a properties file will look like:

MusicalNotes.properties

# Musical Notes Properties file

# This denotes the frequencies of musical notes in Hz

Sa=256

Re=288

Ga=320

Here are the other classes used for this system:

NotesInterpreter.java

package bahavioral.interpreter;

public class NotesInterpreter {

private Note note;

/\*\*

\* This method gets the note from the keys pressed.

\* Them, this sets it at a global level.

\*/

public void getNoteFromKeys(Note note) {

Frequency freq = getFrequency(note);

sendNote(freq);

}

/\*\*

\* This method gets the frequency for the note.

\* Say, if the note is “Sa”, it will return 256.

\*/

private Frequency getFrequency(Note note) {

// Get the frequency from properties

// file using ResourceBundle

// and return it.

return freq;

}

/\*\*

\* This method forwards the frequency to the

\* sound producer which is some electronic instrument which

\* plays the sound.

\*/

private void sendNote(Frequency freq) {

NotesProducer producer = new NotesProducer();

producer.playSound(freq);

}

}// End of class

NotesProducer.java

package bahavioral.interpreter;

public class NotesProducer {

private Frequency freq;

public NotesProducer() {

this.freq = freq;

}

/\*\*

\* This method produces the sound wave of the

\* frequency it gets.

\*/

public void playSound(Frequency freq) {

}

}// End of class

This is how an interpreter pattern works in its most simple implementation. If you are using interpreter pattern, you need checks for grammatical mistakes etc. This can make it very complex. Also, care should be taken to make the interpreter as flexible as possible, so that the implementation can be changed at later stages without having tight coupling.

Other advantage of Interpreter is that you can have more than one interpreter for the same output and create the object of interpreter based on the input. E.g. “Sa” or “Do” can also be implemented as “Download” activity in some other language. In this case, you can use same input and different outputs by getting the proper interpreter from the InterpreterFactory.

This is not a very common pattern.

public class SimpleSingleton {

private static SimpleSingleton INSTANCE = new SimpleSingleton();

//Marking default constructor private

//to avoid direct instantiation.

private SimpleSingleton() {

}

//Get instance for class SimpleSingleton

public static SimpleSingleton getInstance() {

return INSTANCE;

}

}

public class SimpleSingleton {

private SimpleSingleton singleInstance = null;

//Marking default constructor private

//to avoid direct instantiation.

private SimpleSingleton() {

}

//Get instance for class SimpleSingleton

public static SimpleSingleton getInstance() {

if(null == singleInstance) {

singleInstance = new SimpleSingleton();

}

return singleInstance;

}

}

public static synchronized SimpleSingleton getInstance() { }

Builder Pattern

Last updated Nov 21, 2008.

Many times you may want to build a different object depending on the type of input you receive. For example, the book Design Patterns Java Workbook defines a reservation system in which a reservation builder either builds a user a reservation or a counter offer, depending on whether or not minimum criteria is met — if the minimum criteria is met then the user is returned a fully functional reservation object, but if the minimum criteria is not met then a new counter offer object is created that contains the application’s proposal for a valid reservation for the user. The book, Java Design Patterns: A Tutorial, defines a wealth management application that changes its display based upon the number of inputs that is has. For example, if the user is looking at bonds, in which there are three or fewer options, then it displays those options in a set of checkboxes, but if the user is looking at stocks, in which there are more than three options, then it displays those options in a list control.

Both of these examples make use of the builder design pattern. The builder design pattern delegates the construction of an object to a builder class that, depending on the input, constructs an object on your application’s behalf. The object is assembled by a “director” that executes methods on a specific builder type. For example, in the wealth management application, the wealth builder class (the director) obtains the appropriate multiple selection builder object from a factory and then executes methods on that builder object to construct its user interface element. Figure 1 shows a class diagram, taken from the Builder Pattern page on WikiPedia, that illustrates the relationships between the various objects.

Figure 1. Builder Pattern Class Diagram

The Builder class is abstract and defines the methods that concrete builder instances need to implement. There is then one or more concrete builder implementations that understand how to build application Product classes. Finally, the Director is passed a concrete builder implementation to use in constructing Products — the application interacts with the Director.

Following is an automotive example. In this example Mechanic (director) creates cars using one of two car builders: a performance builder or an economy builder. The example that follows then creates both a Porsche and a Civic. Listing 1 shows the source code for the Car object.

Listing 1. Car.java

public class Car {

private String engine;

private String tires;

private String exhaust;

public void setEngine( String engine ) {

this.engine = engine;

}

public void setTires( String tires ) {

this.tires = tires;

}

public void setExhaust( String exhaust ) {

this.exhaust = exhaust;

}

}Listing 2 shows the source code for the abstract CarBuilder class, which will serve as a base class from which concrete CarBuilder instances are created.

Listing 2. CarBuilder.java

public abstract class CarBuilder {

protected Car car;

public Car getCar() {

return car;

}

public void createNewCar() {

car = new Car();

}

public abstract void buildEngine();

public abstract void buildTires();

public abstract void buildExhaust();

}The first CarBuilder example is the FastAndFuriousCarBuilder class that builds highly modified cars. Your petrol average may be poor, but the car will certainly be fun to drive! Listing 3 shows the source code for the FastAndFuriousCarBuilder class.

Listing 3. FastAndFuriousCarBuilder.java

public class FastAndFuriousCarBuilder extends CarBuilder {

public void buildEngine() {

car.setEngine( "Twin Turbo with NOS" );

}

public void buildTires() {

car.setTires( "19 inch low profile racing tires" );

}

public void buildExhaust() {

car.setExhaust( "Deafening!" );

}

}While u enjoy fast cars, the rise in petrol prices forces to buy a Indica! For those of you that like economy cars, the next builder is the EconomyCarBuilder, shown in listing 4.

Listing 4. EconomyCarBuilder.java

public class EconomyCarBuilder extends CarBuilder {

public void buildEngine() {

car.setEngine( "4 cylinder" );

}

public void buildTires() {

car.setTires( "15 inch with plenty of air" );

}

public void buildExhaust() {

car.setExhaust( "Quiet and efficient" );

}

}Now that we have car builders, we need a skilled professional that know how to use these car builders to create working cars. That is the role of the Mechanic (the director in builder pattern terms), shown in listing 5.

Listing 5. Mechanic.java

public class Mechanic {

private CarBuilder carBuilder;

public void setCarBuilder( CarBuilder carBuilder ) {

this.carBuilder = carBuilder;

}

public Car getCar() {

return carBuilder.getCar();

}

public void constructCar() {

carBuilder.createNewCar();

carBuilder.buildEngine();

carBuilder.buildTires();

carBuilder.buildExhaust();

}

}The Mechanic provides a setCarBuilder() method that an application invokes to tell it what type of cars to build. The application then invokes the constructCar() method, followed by a call to getCar() to retrieve the newly constructed car. Listing 6 pulls all of this together to build a Porshe and a Civic.

Listing 6. CarExample.java

public class CarExample {

public static void main( String[] args ) {

Mechanic mechanic = new Mechanic();

CarBuilder performanceBuilder = new FastAndFuriousCarBuilder();

CarBuilder economyBuilder = new EconomyCarBuilder();

mechanic.setCarBuilder( performanceBuilder );

mechanic.constructCar();

Car porsche = mechanic.getCar();

mechanic.setCarBuilder( economyBuilder );

mechanic.constructCar();

Car civic = mechanic.getCar();

}

}The builder pattern has many uses, but the primary criterion that should be evaluated when choosing the builder pattern is identifying the need for different object types based upon some input. You may be tempted to build a robust product class that can adapt itself to its input, but those decision trees detract from the purpose of the object itself: representing its underlying object. For example, the Car class is interested in representing a car, not in understanding how cars are built, so why should it host logic for building cars? Although this section provided a rudimentary example, notice that the car only hosts information about its engine, tires, and exhaust — the logic for configuring a car’s options are delegated to one of the concrete car builders.

In short, if your application needs similar, but distinct, object types, then the builder pattern is a good choice.