Haaris Infotech

JAVA DESIGN PATTERNS

Patterns
Creational Patterns
Factory Pattern
Abstract Factory Pattern
Singleton Pattern
Builder Pattern
Prototype Pattern
Structural Patterns
Adapter Pattern
Bridge Pattern
Composite Pattern
Decorator Pattern
Facade Pattern
Flyweight Pattern
Proxy Pattern
Behavioral Patterns
Chain of Responsibility Pattern
Command Pattern
Interpreter Pattern
Iterator Pattern
Mediator Pattern
Momento Pattern
Observer Pattern
State Pattern
Strategy Pattern
Template Pattern
Visitor Pattern

Creational Patterns

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 Hit and M:

java Hit M

The result returned is: “Hello Mr. Hit”.

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:\\hit\\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.

Creational Patterns - Prototype Pattern

The prototype means making a clone. This implies cloning of an object to avoid creation. If the cost of creating a new object is large and creation is resource intensive, we clone the object. We use the interface Cloneable and call its method clone() to clone the object.

Again let's try and understand this with the help of a non-software example. I am stressing on general examples throughout this write-up because I find them easy to understand and easy to accept as we all read and see them in day-to-day activity. The example here we will take will be of a plant cell. This example is a bit different from the actual cloning in a way that cloning involves making a copy of the original one. Here, we break the cell in two and make two copies and the original one does not exist. But, this example will serve the purpose. Let's say the Mitotic Cell Division in plant cells.
Let's take a class PlantCell having a method split(). The plant cell will implement the interface Cloneable.

Following is the sample code for class PlantCell.

PlantCell.java

```
package creational.builder;

/**
* Shows the Mitotic cell division in the plant cell.
*/
public class PlantCell implements Cloneable {
        public Object split() {
                try {
                        return super.clone();
                        }catch(Exception e) {
```

```
                    System.out.println("Exception occured: "+e.getMessage());
                    return null;
                    }
                }
}// End of class
```

Now let's see, how this split method works for PlantCell class. We will make another class CellDivision and access this method.

CellDivision.java

```
package creational.prototype;
/**
* Shows how to use the clone.
*/
public class CellDivision {
        public static void main(String[] args) {
        PlantCell cell = new PlantCell();

        // create a clone
        PlantCell newPlantCell = (PlantCell)cell.split();
        }
}// End of class
```

One thing is you cannot use the clone as it is. You need to instantiate the clone before using it. This can be a performance drawback. This also gives sufficient access to the data and methods of the class. This means the data access methods have to be added to the prototype once it has been cloned.

Structural Patterns

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.

Structural Patterns - Decorator Pattern

The decorator pattern helps to add behavior or responsibilities to an object. This is also called "Wrapper". Suppose we have some 6 objects and 2 of them need a special behavior, we can do this with the help of a decorator.

Java Design Patterns suggest that Decorators should be abstract classes and the concrete implementation should be derived from them.

The decorator pattern can be use wherever there is a need to add some functionality to the object or group of objects. Let's take an example of a Christmas tree. There is a need to decorate a Christmas tree. Now we have many branches which need to be decorated in different ways.

Let's have a look at the basic Decorator class.

Decorator.java

```
package structural.decorator;

public abstract class Decorator {
        /*
        * The method places each decorative item
        * on the tree.
        */
        public abstract void place(Branch branch);
}// End of class
```

This class has just one method place(). This method places different types of items on the branches of the tree.

The class ChristmasTree is very simple and has just one method which returns a branch.

ChristmasTree.java

```
package structural.decorator;

public class ChristmasTree {
        private Branch branch;

        public Branch getBranch() {
        return branch;
        }
}// End of class
```

Now we can decorate the branches in three different ways, one is by putting coloured balls on them, by putting coloured ruffles on them and also by putting stars on them.

Let's have a look at the implementation of these three different types of decorators.

BallDecorator.java

```
package structural.decorator;

/**
* Decorates the branch of the tree with
* coloured balls.
*/
public class BallDecorator extends Decorator {
        // Default Constructor
        public BallDecorator(ChristmasTree tree) {
        Branch branch = tree.getBranch();
        place(branch);
        }

        /*
        * The method places each decorative item
        * on the tree.
        */
        public void place(Branch branch) {
        branch.put("ball");
        }
```

}// End of class

Similarly, we can make StarDecorator and RufflesDecorator. Now, we will see how to use the decorator. Its simple, we just are needed to pass the instance of ChristmasTree class to a decorator.

StarDecorator decorator = new StarDecorator(new ChristmasTree());

This way the decorator will be instantiated and a branch of the Christmas tree will be decorated.

This is a very abstract example and could not be implemented in terms of code fully. But, then, as I have said, I want you to understand the patterns rather than giving you concrete examples. Once the pattern is internalized, you can think of some good software examples yourself.
Decorators, Composites and Adapters
The decorator and adapter patterns are similar. Adapters also seem to decorate the classes. The intent of using adapter is to convert the interface of one or more classes to suit the interface of the client program. In case of decorator, the intent is to add behavior and functionality to some of the objects, not all the objects or adding different functionalities to each of the objects.

In case of composite objects, the client program treats the objects similarly, whether it is a simple or complex object (nodes).

The decorator pattern provides functionality to objects in a more flexible way rather than inheriting from them.

There is however disadvantage of using decorator. The disadvantage is that the code maintenance can be a problem as it provides the system with a lot of similar looking small objects (each decorator).

Structural Patterns - Facade Pattern

Facade as the name suggests means the face of the building. The people walking past the road can only see this glass face of the building. They do not know anything about it, the wiring, the pipes and other complexities. The face hides all the complexities of the building and displays a friendly face.

This is how facade pattern is used. It hides the complexities of the system and provides an interface to the client from where the client can access the system. In Java, the interface JDBC can be called a facade. We as users or clients create connection using the "java.sql.Connection" interface, the implementation of

which we are not concerned about. The implementation is left to the vendor of driver.

Let's try and understand the facade pattern better using a simple example. Let's consider a store. This store has a store keeper. In the storage, there are a lot of things stored e.g. packing material, raw material and finished goods.

You, as client want access to different goods. You do not know where the different materials are stored. You just have access to store keeper who knows his store well. Whatever you want, you tell the store keeper and he takes it out of store and hands it over to you on showing him the credentials. Here, the store keeper acts as the facade, as he hides the complexities of the system Store.

Let us see how the Store example works.

Store.java

```
package structural.facade;

public interface Store {
        public Goods getGoods();
}// End of interface
```

The store can very well be an interface. This only returns Goods. The goods are of three types as discussed earlier in this document. RawMaterialGoods, FinishedGoods and PackagingMaterialsGoods. All these classes can implement the Goods interface.

Similarly, the stores are of three types and can implement the Store interface. Let's have a look at the code for one of the stores.

FinishedGoodsStore.java

```
package structural.facade;

public class FinishedGoodsStore implements Store {
        public Goods getGoods() {
        FinishedGoods finishedGoods = new FinishedGoods();
        return finishedGoods;
        }
}// End of class
```

Now let's consider the facade StoreKeeper.

StoreKeeper.java

```
package structural.facade;

public class StoreKeeper {
        /**
        * The raw materials are asked for and
        * are returned
        *
        * @return raw materials
        */
        public RawMaterialGoods getRawMaterialGoods() {
        RawMaterialStore store = new RawMaterialStore();
        RawMaterialGoods rawMaterialGoods =
        (RawMaterialGoods)store.getGoods();
        return rawMaterialGoods;
        }


        /**
        * The packaging materials are asked for and
        * are returned
        *
        * @return packaging materials
        */
        public PackingMaterialGoods getPackingMaterialGoods() {
        PackingMaterialStore store = new PackingMaterialStore();
        PackingMaterialGoods packingMaterialGoods =
        (PackingMaterialGoods)store.getGoods();
        return packingMaterialGoods;
        }


        /**
        * The finished goods are asked for and
        * are returned
        *
        * @return finished goods
        */
        public FinishedGoods getFinishedGoods() {
        FinishedGoodsStore store = new FinishedGoodsStore();
```

```
        FinishedGoods finishedGoods = (FinishedGoods)store.getGoods();
        return finishedGoods;
        }
}// End of class
```

This is clear that the complex implementation will be done by StoreKeeper himself. The client will just access the StoreKeeper and ask for either finished goods, packaging material or raw material.

How will the client program access this façade? Here is a simple code.

Client.java

```
package structural.facade;

public class Client {
        /**
        * to get raw materials
        */
        public static void main(String[] args) {
        StoreKeeper keeper = new StoreKeeper();
        RawMaterialGoods rawMaterialGoods =
        keeper.getRawMaterialGoods();
        }
}// End of class
```

In this way the implementation is left to the façade. The client is given just one interface and can access only that. This hides all the complexities.

There is another way of implementing this. We can have just one method in our StoreKeeper class getGoods(String goodsType).

Another version of StoreKeeper method is here.

StoreKeeper.java

```
package structural.facade;

public class StoreKeeper {
        /**
        * The common method
        *
```

```
        * @return Goods
        */
        public Goods getGoods(String goodsType) {

        if (goodsType.equals("Packaging")) {
        PackingMaterialStore store = new PackingMaterialStore();
        PackingMaterialGoods packingMaterialGoods =
        (PackingMaterialGoods)store.getGoods();
        return packingMaterialGoods;
        }
        else if (goodsType.equals("Finished")) {
        FinishedGoodsStore store = new FinishedGoodsStore();
        FinishedGoods finishedGoods = (FinishedGoods)store.getGoods();
        return finishedGoods;
        }
        else {
        RawMaterialStore store = new RawMaterialStore();
        RawMaterialGoods rawMaterialGoods =
        (RawMaterialGoods)store.getGoods();
        return rawMaterialGoods;
        }
}// End of class
```

The client program can now create an object of StoreKeeper class and call method getGoods() passing as parameter the type of goods required. This can be done as follows.

new StoreKeeper().getGoods("RawMaterials");

In this case, the type-casting ill be needed on client side to narrow down Goods to RawMaterialsGoods.

All in all, the Façade pattern hides the complexities of system from the client and provides a simpler interface. Looking from other side, the facade also provides the implementation to be changed without affecting the client code.

Structural Patterns - Flyweight Pattern

The pattern here states about a mechanism by which you can avoid creating a large number of object instances to represent the entire system.

To decide if some part of your program is a candidate for using Flyweights, consider whether it is possible to remove some data from the class and make it

extrinsic. If this makes it possible to reduce greatly the number of different class instances your program needs to maintain, this might be a case where Flyweights will help.

The typical example you can see on this in every book will be of folders. The folder with name of each of the company employee on it, so, the attributes of class Folder are: 'Selected' , 'Not Selected' and the third one is 'employeeName'. With this methodology, we will have to create 2000 folder class instances for each of the employees. This can be costly, so we can create just two class instances with attributes 'selected' and 'not selected' and set the employee's name by a method like:

setNameOnFolder(String name);

This way, the instances of class folder will be shared and you will not have to create multiple instances for each employee.

I was going through this pattern and was trying to find the best suited non-software example. Then, I remembered the talks I had with one of my cousin's who used to work in a grinding wheel manufacturing company. I am a Chemical Engineer and so, remember the names of chemical compounds. He was telling me that the grinding wheels are used for metal cutting across the industry. Basically the main ingredients for these grinding wheels are Aluminum Oxide (Al2O3) and Silicon Carbide (SiC). These compounds are used in form of grains. For those who remember Chemistry from schools, and for others, just follow the example.

His company manufactures nearly 25000 types of grinding wheels. Now, there is another technicality in this and that is bonding.

There are two types of bondings used to bond the material i.e. Aluminum Oxide and Silicon Carbide together. One is Glass bonding – this is like, the wheel is heated to 1300 degree C and the silicon turns into glass to hold the two materials together. The second type of bonding is Resin bonding, this is when some resins help in holding the materials together. The wheels are in different sizes, have different ratio of materials mixed and have any of these two types of bondings. This decides the strength of the wheel. In all, creating 25,000 types of combinations is a pretty complex looking scenario.

If we consider this from software point of view, we can see that each wheel is of a different type and so, we need to make 25000 classes for taking care of each of the wheel. This of course will be very resource intensive. So, how to avoid this?

Well, we already know a few things and we can see a common pattern in each of the wheels. The common things are as follows:

1. Materials of use – They are always Aluminum Oxide and Silicon Carbide.
2. Each wheel has a bonding.
3. Each wheel has a size.

We can follow one thing, the combination above mentioned three ingredients can give us a large number of instances so, why not take a scenario where only one of these is used in the constructor and rest be passed as method parameters.

Let's have a look at the code. For every flyweight, there is a factory providing the object instance. Now, naturally wheels are made in a factory so, there is GrindingWheelFactory class which returns the type of wheel needed.

GrindingWheelFactory.java

```
package structural.flyweight;

/**
* This factory of wheel is accessed by the client. Everytime,
* the client wants a wheel, the software is accessed through this
* Factory class. The specifications are passed through the
* method parameters and a new wheel is returned.
*
* User: hit.satarkar
* Date: Apr 8, 2004
* Time: 5:06:13 PM
*/
public class GrindingWheelFactory {
        /**
        * The method takes the input parameters and return the appropriate
        * wheel.
        *
        * @return An instance of grinding wheel
        */
        public GrindingWheel getWheel(boolean isGlassBonded) {

        return new GrindingWheel(isGlassBonded);
        }
}// End of interface
```

This class is very important. Let's have a closer look at the class. It returns a wheel based only on the bonding. As we know that bondings are only of two types, and so, at any point in time, there will be two instances which are negligible as compared to 25000.

The other important class here is of course GrindingWheel. This gets constructed depending on the parameters passed to the method getWheel() of class GrindingWheelFactory. Let's have a look at the class GrindingWheel.

GrindingWheel.java

```
package structural.flyweight;

/**
* The wheel is formed for different ratio of alumina
* and silicon carbide, for a different bonding and for
* a different size, which depends on the diameter of
* the wheel.
*
* User: hit.satarkar
* Date: Apr 8, 2004
* Time: 5:13:23 PM
*/
public class GrindingWheel {
        private int ratioAlumina;
        private int diameter;
        private boolean isGlassBonded;

        /**
        * Default Constructor
        *
        */
        public GrindingWheel(boolean isGlassBonded) {
        this. isGlassBonded = isGlassBonded;
        }
        .
        .
        .
        .
}// End of class
```

This class can have other methods getters and setters for diameter, and ratioAlumina, on which the complete wheel is dependent.

In each of the instances of the wheels, we can pass the values of ratio of alumina to silicon carbide as method parameters and also the sizes which can lead to a great number of combinations.

Hence, we can see that by using the flyweight pattern, we can reduce the instances of the class.

Structural Patterns - Proxy Pattern

The proxy pattern is used when you need to represent a complex with a simpler one. If creation of object is expensive, its creation can be postponed till the very need arises and till then, a simple object can represent it. This simple object is called the "Proxy" for the complex object

The cases can be innumerable why we can use the proxy. Let's take a scenario. Say, we want to attach an image with the email. Now, suppose this email has to be sent to 1 lakh consumers in a campaign. Attaching the image and sending along with the email will be a very heavy operation. What we can do instead is, send the image as a link to one of the sevlet. The place holder of the image will be sent. Once the email reaches the consumer, the image place holder will call the servlet and load the image at run time straight from the server.

Let's try and understand this pattern with the help of a non-software example as we have tried to do throughout this article.

Let' say we need to withdraw money to make some purchase. The way we will do it is, go to an ATM and get the money, or purchase straight with a cheque. In old days when ATMs and cheques were not available, what used to be the way??? Well, get your passbook, go to bank, get withdrawal form there, stand in a queue and withdraw money. Then go to the shop where you want to make the purchase. In this way, we can say that ATM or cheque in modern times act as proxies to the Bank.

Let's look at the code now.

Bank will define the entire method described above. There are references of classes like You (as the person who wants to withdraw money), also Account, as persons account. These are dummy classes and can be considered of fulfilling the responsibilities as described.

Bank.java

```
package structural.proxy;

/**
* Bank.java
* The class acts as the main object for which
* the proxy has to be created. As described, going
* to bank for withdrawal is very costly time wise.
*/
public class Bank {
        private int numberInQueue;
        /**
        * Method getMoneyForPurchase
        * This method is responsible for the entire banking
        * operation described in the write-up
        */
        public double getMoneyForPurchase(double amountNeeded) {

        // get object for person
        You you = new You("Hit");
        // get obj for account
        Account account = new Account();
        // get person's account number
        String accountNumber = you.getAccountNumber();
        // passbook got.
        boolean gotPassbook = you.getPassbook();

        // get number in queue
        int number = getNumberInQueue();

        // the number will decrease every few mins
        while (number != 0) {
        number--;
        }

        // now when the number = 0, check if balance is sufficient
        boolean isBalanceSufficient = account.checkBalance(accountNumber,
        amountNeeded);

        if(isBalanceSufficient)
        return amountNeeded;
        else
        return 0;
        }
```

```
        /**
        * returns the number in the queue
        */
        private int getNumberInQueue() {
        return numberInQueue;
        }
}// End of class
```

Also, the second class is ATMProxy. This also defines the way the transaction can be handled for withdrawal of money.

ATMProxy.java

```
package structural.proxy;

public class ATMProxy {
        /**
        * Method getMoneyForPurchase
        * This method is responsible for the entire banking
        * operation described in the write-up
        */
        public double getMoneyForPurchase(double amountNeeded) {

        // get obj of You to get card
        You you = new You("Hit");
        // get obj for account
        Account account = new Account();

        boolean isBalanceAvailable = false;
        // if card there, go ahead
        if(you.getCard()) {
        isBalanceAvailable = account.checkBalance(you.getAccountNumber(),
        amountNeeded);
        }

        if(isBalanceAvailable)
        return amountNeeded;
        else
        return 0;
        }
}// End of class
```

Here, we can also create another proxy called ChequeProxy. I am not creating it here as the message I wanted to send across has been conveyed with the help of one proxy only. We can see here that creation of object of Bank is very costly, effort and time wise, and so, we can as well use a proxy called ATM to get the result. ATM can internally communicate with the Bank object. So, ATM here is also acting like a façade.

This might and might not happen. It can happen that we at a later stage have to create the same heavy object. We just want to postpone the creation of object to the last minute so that the application is not loaded by resources utilized for creating the object.

Behavioural Patterns

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.

Haaris Infotech

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.

Behavioral Patterns - Mediator Pattern

The mediator pattern deals with the complexity which comes in the coding when number of classes increase. I will explain this. When we begin with development, we have a few classes and these classes interact with each other producing results. Now, consider slowly, the logic becomes more complex and functionality increases. Then what happens? We add more classes and they still interact with each other but it gets really difficult to maintain this code now. Mediator pattern takes care of this problem. It makes the code more maintainable. It promotes loose-coupling of classes such that only one class (Mediator) has the knowledge of all the classes, rest of the classes have their responsibilities and they only interact with the Mediator.

A very common example can be airplanes interacting with the control tower and not among themselves. The control tower knows exactly, where each of the

airplanes is, and guides them whereas the airplanes have their own responsibilities of landing and takeoff. Another popular example is Stock exchange. In old days when there were no stock markets, the individual brokers used to buy or sell commodities among themselves. They used to face huge risks, of defaulting of counterparty, limited information (as, only limited deals in limited areas were possible), limited geographical reach, price variance (everyone could quote whatever price they wanted) and many more.

So, the concept of stock exchange came into play. For ex: BSE or NSE in India and NYSE, NASDAQ etc in the US. The stock exchange started acting like a mediator and the traders did not need to know other traders and services provided by them to make a deal. The traders have their own responsibilities as buyers and sellers and it was stock exchange's responsibility to match their deals and get the settlement done. Many of the above mentioned risks were mitigated by this. But, there were some standardization procedures which came into picture because of this. All the traders who wanted to deal on stock exchange had to follow these standardization procedures.

Let's look at the code part:

```
/**
* StockExchange – this is the mediator class
*/
public class StockExchange {
    public static void doTransaction (String typeOfTransaction, int quantity, Scrip
    scrip, Trader trader) {
        Transaction transaction = new Transaction(typeOfTransaction, quantity,
        scrip, trader);
        // try and match the current transaction
        // with the ones saved in DB and find out
        // whether a counter transaction is there or
        // are there many such transactions which could
        // fulfill requirement of this transaction.
        matchTransaction(transaction)
    }
    public static getPrice (Scrip scrip) {
        // try and match this transaction with all
        // the saved ones. If they match till whatever extent
        // trade for that. Then save, with status Traded for
        // number of shares traded and save the rest as New.
    }
}// End of class
```

```
/**
* Trader1 – this trader wants to sell 100 shares of company XYZ
*/

public class Trader1 {
    public void doTransaction (String typeOfTransaction, int quantity) {
        int expectedPrice = 320;
        Scrip scrip = new Scrip("XYZ");
        int price = StockExchange.getPrice(scrip);

        if(typeOfTransaction.equals("SELL")){
            if(price >= expectedPrice){
            StockExchange.doTransaction("SELL", 100, scrip, trader1);
            }
        }else if(typeOfTransaction.equals("BUY")){
            if(price <= expectedPrice){
            StockExchange.doTransaction("BUY", 100, scrip, trader1);
            }
        }
    }
}// End of class



/**
* Trader2 – this trader wants to buyl 100 shares of company XYZ
*/

public class Trader2 {
    public void doTransaction (String typeOfTransaction, int quantity) {
        int expectedPrice = 320;
        Scrip scrip = new Scrip("XYZ");
        int price = StockExchange.getPrice(scrip);

        if(typeOfTransaction.equals("SELL")){
            if(price >= expectedPrice){
            StockExchange.doTransaction("SELL", 100, scrip, trader2);
            }
```

```
        }else if(typeOfTransaction.equals("BUY")){
            if(price <= expectedPrice){
            StockExchange.doTransaction("BUY", 100, scrip, trader2);
            }
        }
    }
}// End of class
```

This is simple illustration of how we can use a mediator. Here are the main features of a mediator:
- Responsibilities to individual objects.
- Mediator is the only smart delegating object.
- Transparency for other objects.
- If more objects join in, only place of registration is Mediator, other objects do not need to know anything about the new object.
- The code becomes very maintainable.

On hind side, this brings standardization, which might be cumbersome. Also, there might be a slight loss in efficiency.

Behavioral Patterns - Iterator Pattern

The Iterator pattern is one, which allows you to navigate through a collection of data using a common interface without knowing about the underlying implementation.

Iterator should be implemented as an interface. This allows the user to implement it anyway its easier for him/her to return data.

We use iterators quite frequently in everyday life. For example, remote control of TV. Any remote control we use, either at home/hotel or at a friend's place, we just pick up the TV remote control and start pressing Up and Down or Forward and Back keys to iterate through the channels.

What sort of interface can Iterator be in case of Remote Controls?

```
/**
* Iterator interface has method declarations for iterating through
* TV channels. All remote controls implement Iterator.
*/
public interface Iterator {
    public Channel nextChannel(int currentChannel);
```

```
    public Channel prevChannel(int currentChannel);
}// End of interface
```

The channel iterator is common for all the remote controls. It's like a specification implemented by all the remote control manufacturing companies.

```
/**
* ChannelSurfer is a part of remote control which implements the Iterator
* interface. This class overrides the nextChannel and prevChannel methods.
*/

public ChannelSurfer implements Iterator {
    /**
    * nextChannel – method which takes the current channel number
    * and returns the next channel.
    */
    public Channel nextChannel (int currentChannel) {
        Channel channel = new Channel(currentChannel+1);
        return channel;
     }
    /**
    * prevChannel – method which takes the current channel number
    * and returns the previous channel.
    */
    public Channel prevChannel (int currentChannel) {
            Channel channel = new Channel(currentChannel-1);
            return channel;
    }
}// End of class


/**
* RemoteControl class is the actual remote control and it behaves and makes
* use of ChannelSurfer.
*/
public class RemoteControl {
    private ChannelSurfer surfer;
    private Settings settings;
```

```
    public RemoteControl() {
        surfer = new ChannelSurfer();
        settings = new Settings();
    }
    /**
    * getProgram returns the program for that channel.
    *
    */
    public getProgram(ChannelSurfer surfer) {
        return new Program(surfer.nextChannel());
    }
}// End of class
```

We all know that every channel is associated to a program and it's basically the program and not the channel number which a user wants to see. And so, the implementation which returns a program for channels surfed.

This tells us that we can apply some logic before returning the elements through iterator. We can set rules. The Iterator here, can also be programmed to return the 'programs' straight away rather than returning the channels.

The common Java iterator is Enumeration which has implicit hasMoreElements()
and nextElement() methods.

The benefits of Iterator are about their strength to provide a common interface for iterating through collections without bothering about underlying implementation.