# **MULTI DESIGN PATTERN FOR ENHANCING SOFTWARE USBILITY:**

Adnan Ahmed Fahad Qureshi Talha jamal

K173615 k173925 k173877

1. **ABSTRACT:**

In order to maximize a software’s usability. More concentration should be given to user interfaces. New design patterns are required to provide alternative functionality in case something becomes unresponsive. User never likes handling errors no matter how experienced he/she is with handling errors. The idea is to amalgamate various user-interface design patterns in order to increase responsiveness, and ease of use of the software, in addition to making in learnable, thereby enhancing its usability. Some existing user interface design patterns are lazy registration, forgiving format, clear primary actions, now joining these ideas with some behavioral design patterns like strategy, structural patterns like bridge. We will be coming up with a new design pattern guaranteed to increase user experience and satisfaction.

1. **INTRODUCTION:**

The softwares used by big companies are designed at large scale meaning their code structure is very heavy and complex to maintain. Every now and then it gets bulkier so it needs to refactored. The explanation of programming refactoring is to improve the product framework all through the advancement measure by improving the inner construction and not the usefulness of the product. Advantages of this are expansion in the intelligibility of the product by diminishing the Intricacy of the system.Hence programming support cost has an enormous offer in all out cost of programming improvement [1]. It helps the more adaptable and reusable framework without changing the conduct of code; because of which programming practicality is expanded and inward engineering produced is more manageable.However, defects in the complex programming refactoring measure make support more inefficient.However, blemishes in the perplexing programming refactoring measure make support more wasteful [2].Author [3] named these shortcoming code smells.These are states of plan that impact the conservation of our system.Due to these, our framework turns out to be difficult to change i.e inflexible,and subsequently abandons approached [3]. Fundamentally these code smells don't prevent the framework from working or framework don't begins failing. Be that as it may, due to these framework advancement measure is affected and increment the likelihood of framework breaking down. On the off chance that a class having numerous duties and doing numerous works and is utilizing information from different classes is known as enormous class [4]. There is a standard of item arranged programming that the duties of classes ought to be similarly disseminated among all the more significant level classes[4]. "Overall plan design are reusable answer for a typical common issue or trouble happening in the product plan [6]". By and large, plan designs gives connections

**3)BLUEPRINT OF STUDY PAPER:**

This paper propose strategy to lessen the huge class code smell, by making the classes of indistinguishable kinds and proposing a plan example to address each type.Firstly, we present the fundamental explanation of our exploration paper, what was the issue furthermore, how might we tackle that. In the second area we will show the connected examination work on refactoring hostile to configuration design applications.Then in the third area we will characterize them in to various classifications in detail. And afterward at last in the forward segment, we will show the work by identifying the enormous classes in the genuine framework and afterward by applying configuration designs to refactor them.

1. **INSPIRATION:**

Quite possibly the most continuous happening terrible stench in the product framework is the huge class.Considering, Large class terrible stench as most hurtful bed smell this paper will zero in on a few of the fundamental reasons of the event of this code smell in terms of highlights and afterward we will order them in various types.The primary impediment of huge class is low attachment and high intricacy.

1. **OBJECTIVE:**

By applying various examples, this paper propose a viable strategy to lessen huge class code smell and making programming framework effective. Subsequently the two primary targets of this paper can be expressed as follows. Identifying various reasons for enormous class awful stench Suggesting distinctive plan examples to address each type of enormous class awful stench

1. **CHALLENGE:**

Our main idea is to bring the concept of usability to the developer side. To have a kind of framework for the developer so that he doesn’t need to worry about the concepts of usability. Our idea is to merge the existing design patterns from the three branches which are namely creational, structural and behavioral. And merge them for usability concepts now usability mainly consists of three principles which are:

1. Effectiveness.
2. Efficiency.
3. Accuracy.

Now we will be defining each of the above concepts in a little detail:

**EFFECTIVENESS:**

How much the system is easy to learn. The ease of preforming a task and how much the user is convenient with the way of performing a task and he/she is not finding any difficulty to use the system.

**EFFICIENCY:**

The resources taken by the system to perform a task should be minimum. So a task is performed efficiently and the user does not need to worry about the working load of the system.

**ACCURACY:**

There should be no difference in the service being provided and the service requested. Or there should be minimum difference.

Now we need to bring these concepts to the developer’s end so that when he is building the system he should be able to achieve accuracy, effectiveness and efficiency and have a complete framework to implement all these concepts. By framework we mean a design pattern. so this is our challenge. We are required to build a design pattern merged in order to provide usability at the developer’s end.

**METHODOLOGY:**

We want to increase usability of software for the developer and in the client code. For usability we have taken into account the aspects of accuracy, efficiency and effectiveness. And we tried to address each of these aspects with a design pattern each and in then we ended up combining three design patterns. We thought it would be more beneficial if we combined the three design patterns from the three types which are creational, structural and behavioral. So we took the liberty of combining one design pattern from each type. We chose the following design patterns:

|  |  |
| --- | --- |
| TYPE | DESIGN PATTERN |
| Creational | **Factory design pattern** |
| Structural | **Decorator design pattern** |
| Behavioral | **Strategy design pattern** |

Usability is a principle for user interface. It consists of concepts like accuracy, effectiveness and efficiency. Which are defined as:

**The effectiveness of the system:**

Is the system is easy to learn and capable of solving the problem.

**Efficiency:**

Little resources should be used to reach a desired goal.

**Accuracy:**

The desired task should only be reached and no useless data should be given.

Now how can we use these concepts from existing design patterns so that the developer don’t have to worry about reaching accuracy, efficiency and effectiveness such that the developer have a framework like design pattern to work with. Now we wanted to achieve these concepts using the design pattern from the gang of four 24 design patterns. So we decided to achieve each usability concept from one design pattern respectively. So we came up with the following mapping:

|  |  |
| --- | --- |
| USABILITY PRINCIPLE | DESIGN PATTERN |
| EFFICIENCY | **Factory design pattern** |
| EFFECTIVENESS | **Decorator design pattern** |
| ACCURACY | **Strategy design pattern** |

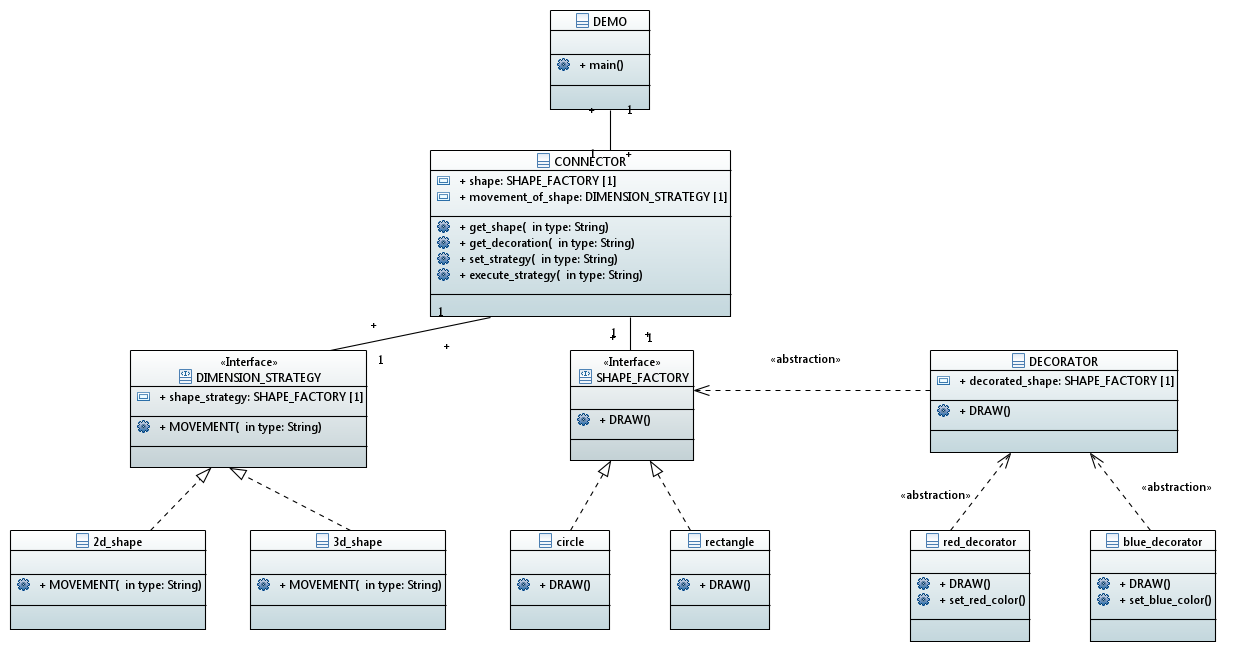
Now the factory pattern can be used for efficiency because it will return the correct object and no further use of resources for unnecessary stuff.

The decorator pattern is used for effectiveness as it makes the system simple. Because in decorator pattern when we need to add more functionality we just extend one more class for it and no change in the client code. The developer will pick it up right away what changes are required and be done with them. So the system becomes easy to learn.

The strategy pattern is used for accuracy as it will associate the exact strategy needed to associate with the object provided by the factory pattern. So it gives us accuracy.

**CLASS DIAGRAM:**

Following is class diagram for the amalgamated design patterns described above for achieving each goals of usability respectively.



There are three design patterns defined here connected by the CONNECTOR class the left side abstract class is and its child is used for strategy pattern , in the middle we have factory class and in the right we have abstract class for the decorator pattern. the demo class is the main class used for calling the functions and classes.

**EXAMPLE BEING FOLLOWED:**

The example being followed here is that there is a factory class for creating shape object like circle or rectangle , with that we need some decoration like fill the shape with some specific color and with this we also need a strategy to move the shapes like in 2 dimensional plane or in 3 dimensional plane. So instead of just shape we now also need a decoration and a strategy to go with the shape.

**SOLUTION:**

What we did was simple we connected the strategy and decorator pattern’s abstract parent class implement the factory class interface and over write its functions for the use of decorations and strategy. The child classes of decorator and strategy over write there parent’s abstract classes methods and provide decoration and strategy respectively to the shape object given by the factory class. Following is the code:

**CODE:**

**FACTORY DESIGN PATTERN:**

**SHAPE FACTORY INTERFACE:**

**package** ddr\_project;

**public** **interface** SHAPE\_FACTORY {

/\*\*

\*

\*/

**public** **void** DRAW() ;

};

**CONCRETE CLASSES OF SHAPES IMPLEMENTING FACTORY INTERFACE:**

**RECTANGLE CLASS:**

**package** ddr\_project;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\*/

**public** **class** rectangle **implements** SHAPE\_FACTORY

{

/\*\*

\*

\*/

**public** **void** DRAW() {

System.***out***.println("rectangle drawn");

}

};

**CIRCLE CLASS:**

**package** ddr\_project;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\*/

**public** **class** circle **implements** SHAPE\_FACTORY

{

/\*\*

\*

\*/

**public** **void** DRAW() {

System.***out***.println("CIRCLE OBJECT CREATED");

}

};

Now we bring in strategy pattern:

**STRATEGY PATTERN:**

**DIMENSION STRATEGY CLASS FOR PROVIDING 2D OR 3D DIMENSION:**

**package** ddr\_project;

**public** **interface** DIMENSION\_STRATEGY {

/\*\*

\*

\*/

**public** **void** MOVEMENT(String type) ;

};

In the above interface class we are defining a strategy function of movement which will be over written in the child class of 2D and 3D movement to define separate strategies.

**2D AND 3D CLASS EXTENDING DIMENSION STRATEGY INTERFACE:**

**twoD\_shape class:**

**package** ddr\_project;

**public** **class** twoD\_shape **implements** DIMENSION\_STRATEGY

{

/\*\*

\*

\*/

**public** **void** MOVEMENT(String type) {

System.***out***.println(type + " is moving in 2D");

}

/\*\*

\*

\*/

};

**threeD\_shape class:**

**package** ddr\_project;

**public** **class** threeD\_Shape **implements** DIMENSION\_STRATEGY

{

/\*\*

\*

\*/

**public** **void** MOVEMENT(String type) {

System.***out***.println(type + " shape is moving in 3d");

}

/\*\*

\*

\*/

};

The above classes are over writing the strategy interface to define specific strategies of 2 dimensional movement and 3 dimensional movement.

**DECORATOR PATTERN:**

**DECORATOR ABSTRACT CLASS IMPLEMENTING THE FACTORY SHAPE INTERFACE:**

**package** ddr\_project;

**public** **abstract** **class** DECORATOR **implements** SHAPE\_FACTORY{

**public** SHAPE\_FACTORY decorated\_shape;

/\*\*

\*

\*/

**public** DECORATOR(SHAPE\_FACTORY shape\_to\_decorate) {

**this**.decorated\_shape = shape\_to\_decorate;

}

**public** **void** DRAW() {

decorated\_shape.DRAW();

}

};

In the above class we are passing an object provided by the factory class and over writing the draw function and in that draw function we are describing the decorations needed. It is further extended by our red decoration and blue decoration classes

**RED DECORATION AND BLUE DECORATION EXTENDING THE PARENT DECORATOR ABSTRACT CLASS:**

**RED DECORATION CLASS:**

**package** ddr\_project;

**public** **class** red\_decorator **extends** DECORATOR{

**public** red\_decorator(SHAPE\_FACTORY shape\_to\_decorate) {

**super**(shape\_to\_decorate);

}

**public** **void** DRAW() {

set\_red\_color();

}

**public** **void** set\_red\_color() {

System.***out***.println("red color filled in");

}

};

**BLUE DECORATION CLASS:**

**package** ddr\_project;

**public** **class** blue\_decorator **extends** DECORATOR{

/\*\*

\*

\*/

**public** blue\_decorator(SHAPE\_FACTORY shape\_to\_decorate) {

**super**(shape\_to\_decorate);

}

**public** **void** DRAW() {

set\_blue\_color();

}

/\*\*

\*

\*/

**public** **void** set\_blue\_color() {

System.***out***.println("blue color filled in");

}

};

The above classes are over writing the decorator abstract class draw method so that the object passed to the decorator class from the shape factory class, is assigned the appropriate decoration which are set color here method.

**CONNECTOR CLASS:**

**package** ddr\_project;

**public** **class** CONNECTOR {

/\*\*

\*

\*/

**public** SHAPE\_FACTORY shape;

/\*\*

\*

\*/

**public** DIMENSION\_STRATEGY movement\_of\_shape;

/\*\*

\*

\*/

**public** SHAPE\_FACTORY get\_shape(String type) {

**if**(type.equals("circle")) {

shape = **new** circle();

**return** shape;

}**else** {

shape = **new** rectangle();

**return** shape;

}

}

/\*\*

\*

\*/

**public** **void** set\_strategy(DIMENSION\_STRATEGY ds) {

**this**.movement\_of\_shape = ds;

}

**public** **void** execute\_strategy(String type) {

**this**.movement\_of\_shape.MOVEMENT(type);

}

/\*\*

\*

\*/

**public** **void** get\_docaration(String type,SHAPE\_FACTORY shape) {

**if**(type=="red\_decoration") {

shape = **new** red\_decorator(**this**.shape);

shape.DRAW();

}

**else** **if**(type=="blue\_decoration") {

shape = **new** blue\_decorator(**this**.shape);

shape.DRAW();

}

}

};

The connector class is just connecting all the patterns together and calling methods from exact classes so that the functionality can be provided to the main client code and we don’t need to write further logic of how to access stuff in the client code. You can see in the connector class there are get shape is implementing the factory design pattern returning the object and the set strategy and execute strategy is implementing the strategy pattern giving you an object associated with your shape to give the strategy whether it is 2D or 3D. the get decoration is implementing the decorator class in which we are passing the shape object to the decorator class so that it can be appropriately decorated as defined in the client code with respect to the functions provided in the system

**CLIENT CODE (MAIN):**

**package** ddr\_project;

**public** **class** DEMO {

/\*\*

\*

\*/

**public** **static** **void** main(String args[]) {

SHAPE\_FACTORY s1;

CONNECTOR c1 = **new** CONNECTOR();

s1 = c1.get\_shape("circle");

s1.DRAW();

c1.set\_strategy(**new** twoD\_shape());

c1.execute\_strategy("circle");

c1.get\_docaration("red\_decoration",s1);

System.***out***.println("\n");

SHAPE\_FACTORY s2;

CONNECTOR c2 = **new** CONNECTOR();

s2 = c2.get\_shape("rectangle");

s2.DRAW();

c2.set\_strategy(**new** threeD\_Shape());

c2.execute\_strategy("rectangle");

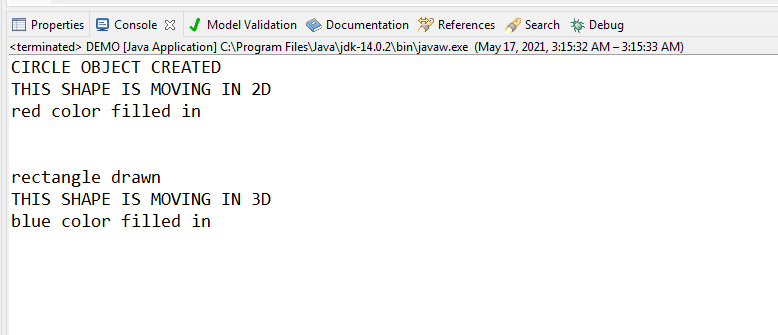
c2.get\_docaration("blue\_decoration",s2);

}

};

Now we can see that in the client code we can easily just define one connector class object (one for each shape object) and have it access all functionality. The get shape provides you shape object and the set and execute strategy and get decoration are providing you strategy and decoration for your created object as you are passing the created object in these functions. And everything is taken care of inside the classes.

**OUTPUT:**



**CONCLUSION:**