Research Journal of Applied Sciences, Engineering and Technology 13(5): 416-421, 2016
DOI:10.19026/rjaset.13.2961
ISSN: 2040-7459; e-ISSN: 2040-7467

Submitted: April 5, 2016 Accepted: June 7, 2016 Published: September 05, 2016

Research Article

A Design Pattern Approach to Improve the Structure and Implementation of the Decorator Design Pattern

[1]Bilal Hussein and [2]Aref Mehanna
[1]Institute of Technology, Lebanese University, Saida,
[2]Faculty of Economics and Business Administration, Lebanese University, Aaley, Lebanon

Abstract: Reusability is a technique used to unify the abstract representation of real world entities. In object-oriented design and programming, inheritance mechanism plays an important role in software reusability. The Decorator Design Pattern (DDP), invented by GoF, was an alternative solution to the inheritance mechanism. It uses the concept of decorating objects instead of inheritance. The aim of this study is to present the DDP, showing its advantages and disadvantages and offer a new innovative approach, called Decorator Pattern Approach (DPA), in order to improve the structure and implementation of DDP. The main objective of DPA is to provide a way to separate, dynamically, the decorating objects from the objects to decorate.

Keywords: Decorator,design pattern, Decorator Pattern Approach (DPA), inheritance, Object Oriented Design (OOD), reusability

INTRODUCTION

The system's design with the object approach is to model the entities of the real world by an abstract representation often described by graphical notations such as classes and associations in UML (Unified Modeling Language). The class diagram is one of UML diagrams whom we can model with the static structure of a system in the form of classes (attributes and methods) and associations (Grady *et al*., 1998). For recurring problems, designers do not build the same class diagrams. This is for several reasons, first the real world entities are not seen and described in the same way. Second, designers do not have the same level of expertise. For this reason, developers prefer to develop software from existing classes. This concept is called reusability. Reusability is an important characteristic of high quality classes (Goyal and Gupta, 2014). Reusability brings the following benefits: development cost is reduced; reliability is increased, less time to market and Low cost maintenance (Alvaro *et al*., 2006; Narwal, 2012). In object-oriented design and programming (OODP), inheritance mechanism plays an important role in software reusability. This enables to create a new class from an existing class by adding new responsibilities. However, reusability by inheritance requires a multiplication of subclasses to describe real entities and especially when the objects of these entities do not share many features. Moreover, the reusability by inheritance is limited to the classes declared non-final (i.e., the classes declared final cannot be sub-classed). Furthermore, reusability by inheritance provides a very strong coupling between existing and new classes. The Decorator Design Pattern (DDP) invented by GoF (Gamma *et al*., 1995), was an alternative solution to the reusability by inheritance. It allows adding new responsibilities to one or more objects dynamically without affecting other objects (Gamma *et al*., 1995). The aim of this study is to present the DDP, showing its advantages and disadvantages and offer a new innovative approach, called Decorator Pattern Approach (DPA), in order to improve the structure and implementation of DDP. The main objective of DPA is to provide a way to separate, dynamically, the decorating objects from the objects to decorate.

MATERIALS AND METHODS

When and where this study was conduct: Lebanese University.

In this section, we present the reusability and inheritance concepts. We show the Decorator design pattern structure and implementation and an example of use.

Reusability and inheritance: To simplify the time devoted to the coding phase in a software life cycle, it is

Corresponding Author: Bilal Hussein, Institute of Technology, Lebanese University, Saida, Lebanon, Tel.: 9613215145

interesting to be able to reuse an existing code. In this context, the object-oriented allows encapsulating a code in a structure called class. The class includes two aspects: static and dynamic. The static aspect is described by a set of attributes and the dynamic aspect by a set of operations.

The class reuse is done in two different ways: either to use the class without changing the existing code or use it by adding certain functionalities (Inheritance). Inheritance is a main feature of OODP paradigm. It is used to encapsulate a set of closely related functionality in a structured hierarchy where common functionality is added in one class and more specialized functionality of that class is added in other classes (Varsha and Shweta, 2013).

For example, suppose that 'A' is an already compiled class and 'B' a class that wants to use it. Then, the class 'B' can instantiate objects from the class 'A' and call its methods. Now, if we want to add new functionality to the class 'A' before 'A' is used by the class 'B', we have to create a class 'C' derived from the class 'A' and add to it the desired functionaly. Finally, the class 'B' can instantiate objects from the class 'C'.

Limits and difficulties of inheritance: Sometimes we want to add new functionality to individual objects, not to an entire class. One way to do this is with inheritance. Inheriting a border from another class puts a border around every subclass instance. This is inflexible, however, because the choice of border is made statically, a client can't control how and when to decorate the component with a border (Gamma *et al*., 1995).

The extension by sub classing is impractical. Sometimes a large number of independent extensions is possible and would produce an explosion of subclasses to support every combination (Gamma *et al*., 1995).

Moreover, you cannot reuse a single method of a class without inheriting that class's other methods as well as its data members.

Another disadvantage of inheritance is the difficulty to implement the multiple inheritance mechanism in some programming languages such as Java. Multiple inheritance mechanism allows a single class to inherit the characteristics from several super classes. This concept can produce many consequences at polymorphism level. A well-known drawback in Java is its limitation in implementing multiple inheritances which is considered by many researchers a fundamental concept in object oriented (Albalooshi, 2015).

In their book, Gang of Four (GoF) (Gamma *et al*., 1995) have provided a set of design patterns including one called Decorator whose goal is to overcome the difficulties and limitations of the inheritance.

Decorator design pattern: Design patterns are a proven way to build flexible software architectures (Pavlič *et al*., 2014). Decorator pattern is one of these patterns invented by GoF (Gang of Four). Design patterns are conceptual solutions built using class diagrams. The purpose, of these patterns, is to provide designers and developers the conceptual solutions of recurring problems without needing to rebuild applications from the beginning. Indeed design patterns cover most of the problems associated with the design and implementation of large and complex software systems (Debboub and Meslati, 2013).

Design patterns are grouped into three categories: Creational, structural and behavioral. Every pattern and according to its interest, is attached to one of these categories. For instance, we list the following patterns: Abstract Factory, Singleton (creational), Adapter, Decorator (structural), Observer, visitor (behavioral), etc.

For a better understanding of the design pattern, GoF contributors use a consistent format including some sections such as: Intent, motivation, structure, implementation (Gamma *et al*., 1995).

Intent: what does the design pattern do?

Motivation: A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem.

Structure: A graphical representation of the classes in the pattern using an object modeling notations.

Implementation: What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

Decorator pattern basic concepts:

Intent: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality (Gamma *et al*., 1995).

Motivation: Sometimes we want to add functionality to individual objects, not to an entire class. Decorator uses an object that modifies behavior of, or adds features to, another object.

Structure: Figure 1 illustrates the structure of the Decorator design pattern.

Component can be declared as an interface or an abstract class. The class *ConcreteComponent* implements (or extends) *Component*. *Decorator* is an abstract class and it implements *Component.* It contains an attribute called *component* that holds an instance of type *Component*. This instance is decorated using decorating objects of the classes *ConcreteDecoratorA* and *ConcreteDecoratorB*. These classes derive from *Decorator*.

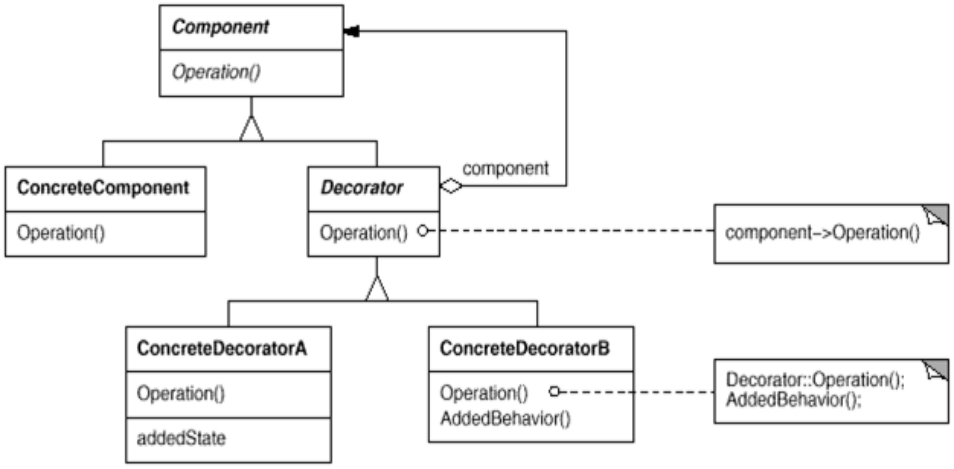

Fig. 1: Decorator design pattern structure

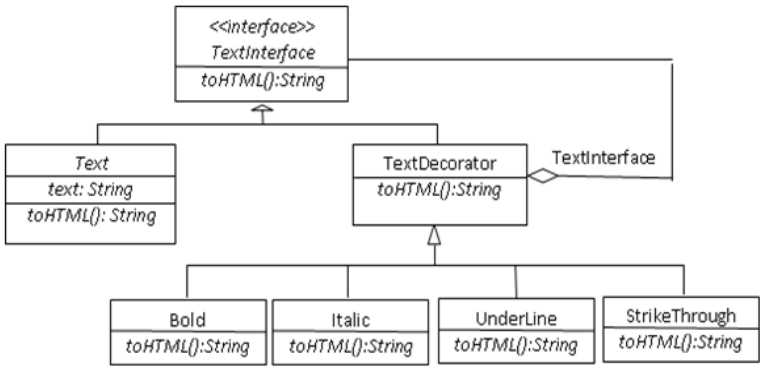

Fig. 2: Decorator of text objects

Example: Suppose we want to format, dynamically, a text object by adding to it some of the font characteristics such as Bold, Italic, Underline, Strickethrough, Subscript, etc. The decorator design pattern structure for this example is illustrated in the Fig. 2:

Implementation:

```
public interface TextInterface{
public String toHTML();
}
public class Text implements TextInterface{
private String text;
public Text(String text){this.text = text;}
public String toHTML(){return this.text;}
}
public abstract class TextDecorator implements TextInterface{
privateTextInterfaceaText;
publicTextDecorator(TexteInterfaceaText){
this.aText = aText;
}
public String toHTML(){
returnaText.toHTML();
}
}
public class Bold extends TextDecorator{
public Bold(TextInterfaceaText){
super(aText);
}
public String toHTML(){
return "<B>" + super.toHTML() + "</B>";
}
}
public class Italic extends TextDecorator {
public Italic(TextInterfaceaText){
super(aText);
}
public String toHTML(){
return "<I>" + super.toHTML() + "</I>";
}
}
public class Strike extends TextDecorator {
public Strike(TextInterfaceaText){
super(aText);
}
public String toHTML(){
return "<S>" + super.toHTML() + "</S>";
}
}
```

```
public class UnderLine extends TextDecorator {
publicUnderLine(TextInterfaceaText){
super(aText);
}
public String toHTML(){
return "<U>" + super.toHTML() + "</U>";
}
}
```

Now, suppose we need to format the text "Welcome to Lebanon" in Bold, Italic and UnderLine. The implementation will be as follow:

TextInterface text = new Bold (new Italic (new UnderLine (new Text (″Welcome to Lebanon″)));
String s = text.toHTML();

RESULTS AND DISCUSSION

In this section, we present our DPA approach in order to overcome some weaknesses found in the use of the original DDP.

Our discussion starts from the following question: what can we do if we want to reformat the same text 'Welcome to Lebanon' in Bold and strikethrough only? The implementation should be changed as:

TextInterface text = new Bold (newStrikeThrough (new Text(″Welcome to Lebanon″)));
String s = text.toHTML();

Here, there are several points to discuss:

- What happened to the old decorating objects, Bold, Italic and Underline that are used to decorate our text object? Are they lost?

According to the Decorator pattern structure, we can say that this structure is recursive. Each decorating object has, in its private attribute 'TextInterface', the reference of another decorating object and so forth. Note that the constructor of the Decorator class assigns to the decorating object attribute the reference of another decorating object. We conclude that objects, Bold, Italic and UnderLine are lost. However, the re-decoration of a same object requires the creation of new decorating objects. Finally, the number of lost objects increases as many times as there are re-decorating operation.

Based on this weakness, we list some questions in order to improve the structure and implementation of Decorator pattern:

- How can we reduce the number of lost decorating objects?
- Can we use the same decorating object to make a re-decoration?
- Can we, with the same decorating object, decorate many different objects?

Our DPA approach provides an answer to these questions. It presents a new motivation, structure and implementation of DDP.

New motivation: Our DPA approach consists in separating the decorating objects from objects to decorate. This means they do not share the same interface or abstract class.

New structure: The following structure (Fig. 3) illustrates the new improved structure of DPA.

Decoratee is the class of objects to decorate. *Decoratee* has an association with *Decorator* interface which allows *Decoratee* to declare an attribute of type *Decorator*.

The class composite *Decorator* contains all concrete decorator elements that can be used to decorate the *Decoratee*.

The above structure shows that *Decoratee* objects are separated from decorator elements. This allows dynamically adding and removing decorator elements. Also, *Decorator* can be used to decorate another type of *Decoratee*. And, we don't need to enclose objects in other objects, simply you have to add or remove decorator elements from the composite. This flexibility allows programmers at run time, not to use the

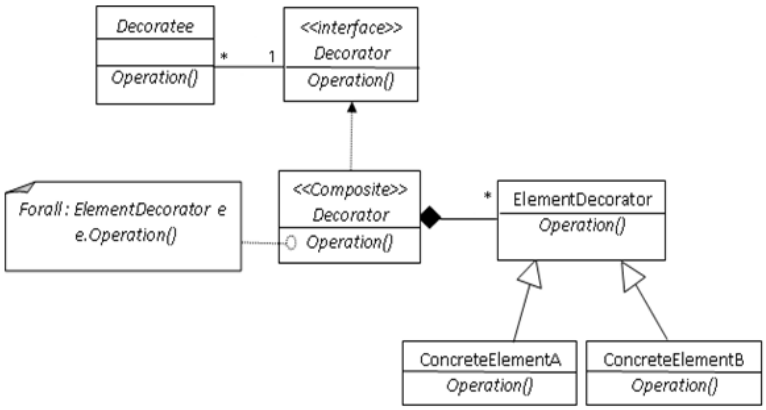

Fig. 3: Structure of a new decorator pattern

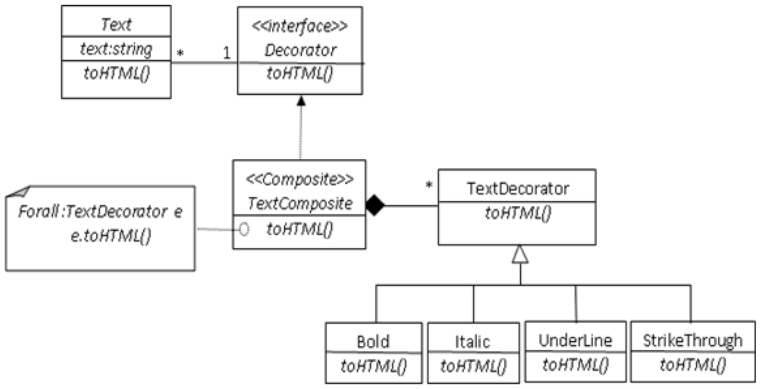

Fig. 4: Decorator of text objects

reference assignments such as those used in the standard Decorator pattern.

By returning to our previous example, the structure according to our approach will be (Fig. 4):

New implementation:

```
importjava.util.ArrayList;
importjava.util.Collection;
interface Decorator{
public String toHTML(String st);
public void add(TextDecorator d);
public void remove(TextDecorator d);
}
interfaceTextDecorator{
public String toHTML(String st);
}
class Text{
Decorator myDecorator=new TextComposite();
String text;
public Text(String text){this.text = text;}
public String toHTML(){return myDecorator.toHTML(this.text);}
}
classTextComposite implements Decorator{
Collection <TextDecorator> decorators = new ArrayList<TextDecorator>();
public String toHTML(String st){
for (TextDecorator e: decorators)
st=e.toHTML(st);
returnst;
}
public void add(TextDecorator d){
decorators.add(d);
}
public void remove(TextDecorator d){
decorators.remove(d);
}
}
class Bold implements TextDecorator{
public String toHTML(String st){
return "<B>" + st + "</B>";
}
}
class Italic implements TextDecorator {
public String toHTML(String st){
return "<I>" + st + "</I>";
}
}
class Strike implements TextDecorator {
public String toHTML(String st){
return "<S>" + st + "</S>";
}
}
classUnderLine implements TextDecorator {
public String toHTML(String st){
return "<U>" + st + "</U>";
}
}
public class Client{
public static void main(String [] args){
Text text1=new Text("Welcome to Lebanon");
TextDecorator B=new Bold();
TextDecorator U=new UnderLine();
TextDecorator S=new Strike();
// Decorating object text1 with bold, underline // and strike
text1.myDecorator.add(B);
text1.myDecorator.add(U);
text1.myDecorator.add(S);
System.out.println(t.toHTML());
// Re-decoration text1 by Bold and strike
// requires only removing of the decorating
// object underline (U).
text1.myDecorator.remove(U);
System.out.println(t.toHTML());
// Same objects underline (U) and strike (S)
// are used to decorate another text (text2).
text2.myDecorator.add(U);
text2.myDecorator.add(S);
System.out.println(t1.toHTML());
}
}
```

If we pass a simple look to our example, we see that decorating objects (B, U and S) have been used, dynamically, to decorate and re-decorate the text

objects without affecting or losing any decorating object. Finally, we were able to limit the number of decorating objects and use them for a decoration and re-decoration of multiple objects as many times as we want.

CONCLUSION

Design patterns in object oriented design and programming are efficient ways to improve the performance of the reusability mechanism. Reusability is achieved using inheritance. Decorator pattern is one of these design patterns that provides a very powerful alternative to inheritance. By using the Decorator pattern, we can avoid some difficulties coming from the use of inheritance and especially the situation that produces an explosion of subclasses to support every combination. That is why we selected the Decorator pattern in order to give it some flexibility during implementation and using. We suggested in this study a new innovative approach called DPA (Design Pattern Approach) which purpose is to improve the structure and implementation of the Decorator design pattern. With DPA approach, we were able to limit the number of used decorating objects. Also we are able to re-use them at any time in order to perform decoration and re-decoration operations to every object without affecting old decorated objects.

REFERENCES

Albalooshi, F., 2015. Software design concerns associated with simulating multiple inheritance in java for implementation purposes. Brit. J. Math. Comput. Sci., 6(5): 435-443.

Alvaro, A., E.S. De Almeida and S.L. Meira, 2006. A software component quality model: A preliminary evaluation. Proceeding of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06). Cavtat, Dubrovnik, pp: 28-37.

Debboub, S. and D. Meslati, 2013. Quantitative and qualitative evaluation of AspectJ, JBoss AOP and CaesarJ, using Gang-of-Four design patterns. Int. J. Softw. Eng. Appl., 7(6): 157-174.

Gamma, E., R. Helm, R. Johnson and J. Vlissides, 1995. Design Patterns: Elements of Resuable Object-Oriented Software. Pearson Education Ltd., England.

Goyal, N. and E.D. Gupta, 2014. Reusability calculation of object oriented software model by analyzing CK metric. Int. J. Adv. Res. Comput. Eng. Technol., 3(7): 2466-2470.

Grady, B., R. James and J. Ivar, 1998. The Unified Modeling Language User Guide. 1st Edn., Addison Wesley.

Narwal, A., 2012. Empirical evaluation of metrics for component based software systems. Int. J. Latest Res. Sci. Technol., 1(4): 373-378.

Pavlič, L., V. Podgorelec and M. Heričko, 2014. A question-based design pattern advisement approach. Comput. Sci. Inf. Syst., 11(2): 645-664.

Varsha, M. and Y. Shweta, 2013. Reusability evaluation of object oriented inheritance and interface code. Eng. Univ., Sci. Res. Manage., 5(2).