Decorator Pattern by Example

Venkat Subramaniam
venkats@agiledeveloper.com
http://www.agiledeveloper.com/download.aspx

Abstract

In this article we present an example that illustrates the use of the Decorator pattern. The objective is to illustrate the flexibility provided by this very elegant pattern. The example is presented in Java; however any .NET developer must be able to grasp the concept from it.

An Evaluation Application

Let's consider a program where there are some rules that will be used in evaluating an application submitted to a University. Say the Registrar object, in our program, is responsible for carrying out the specific evaluation by using the rules. To start with say we only have been given the GPAEval as the criteria to be applied.

So, here is one possible code we have for this:

The Registrar class is shown first:

```
public class Registrar
{
    public boolean evaluate(Application theApp, GPAEval criteria)
    {
        //...do what ever you have to here

        // when all else done

        boolean success = criteria.evaluate(theApp);

        // ... do what ever else you have to

        return success;
    }
}
```

The evaluate method does whatever it has to. In addition, it calls evaluate on the GPAEval object. The GPAEval object encapsulates and isolates the evaluation based on the GPA requirements.

The GPAEval object is shown below:

```
public class GPAEval
{
    public boolean evaluate(Application theApp)
    {
        // Code to do actual work like if (theApp.getGPA() > 3) result = true;

        System.out.println("GPAEval.evaluate called");
        return true;
    }
}
```

The Application class itself does not have a whole lot (and will not in this example) as shown below:

```
public class Application
{

}
```

Let's look at a test code that will use all this:

```
public class TestCode
{
    // Sample code to run our little app

    public static void main(String[] args)
    {
        Application anApp = new Application();
        Registrar reg = new Registrar();

        GPAEval aGPAEval = new GPAEval();

        reg.evaluate(anApp, aGPAEval);
    }
}
```

Running this program produces the following result:

```
GPAEval.evaluate called
```

Let's Extend this now

Now that we got this working, let's say we are asked to, in some but not all cases, evaluate GRE scores in addition to GPA scores. How can we realize this without breaking the Registrar class?

One possibility is to derive from the GPAEval class as shown below:

```
public class GREEval extends GPAEval
{
    public boolean evaluate(Application theApp)
    {
        if(super.evaluate(theApp))
        {
             // Code to do actual work like if (theApp.getGRE() > ...) result = true;
            System.out.println("GREEval.evaluate called");
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

The modified TestCode is shown below:

```
public class TestCode
{
    // Sample code to run our little app

    public static void main(String[] args)
    {
        Application anApp = new Application();
        Registrar reg = new Registrar();

        System.out.println("Running first eval");
        GPAEval aGPAEval = new GPAEval();

        reg.evaluate(anApp, aGPAEval);

        //Evaluate GRE and GPA
        System.out.println("Running second eval");
        GREEval aGREEval = new GREEval();
        reg.evaluate(anApp, aGREEval);
    }
}
```

And the output is:

```
Running first eval
GPAEval.evaluate called
Running second eval
GPAEval.evaluate called
GREEval.evaluate called
```

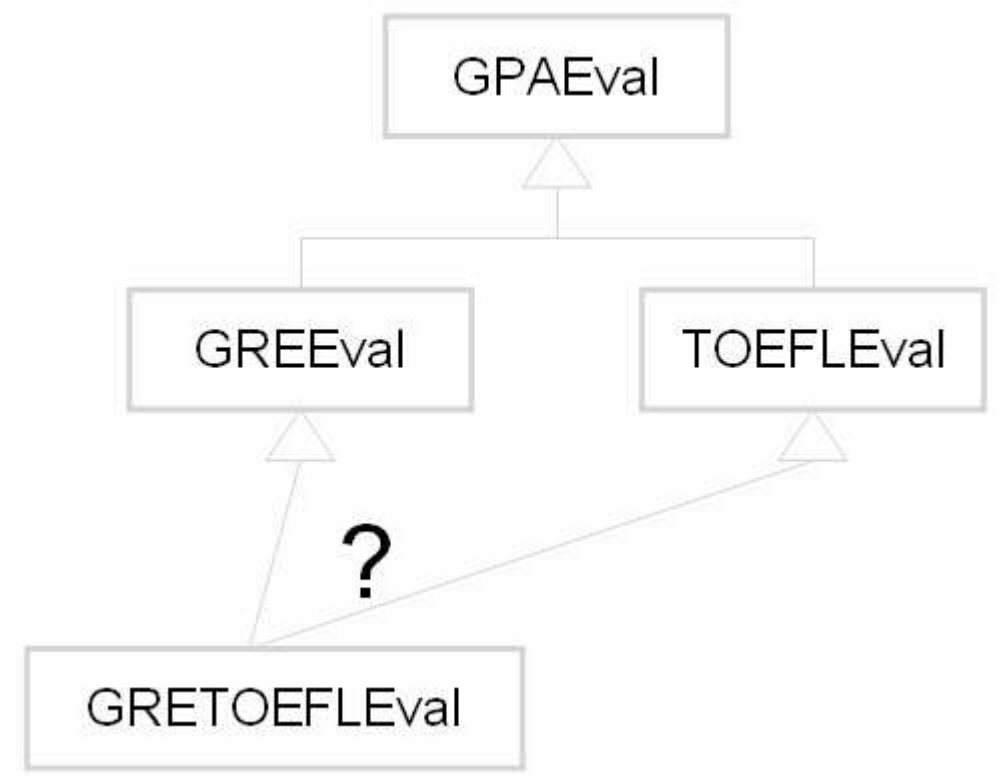
OCP Compliance

That's great so far. We kept the code pretty much OCP[1] compliant so far. We were able to accommodate the change in requirements by adding new module of code and not changing any existing code (not considering change to the TestCode, of course). Now, say we are asked to evaluate the applicant based on TOEFL and GPA. How can we do

that? We can write another class TOEFLEval which inherits from GPAEval and that should take care of it right?

Growing Pain

What if we are asked to evaluate an application on GPA, GRE and TOEFL scores? Again, not all application may have to be evaluated based on all these criteria. Only a few may need a combination of these criteria. Going down the path we have taken so far, should we write another class that derives from both GREEval and TOEFLEval? I hear you saying "you can't do that in Java (or .NET)." Should I inherit the class from GREEval and may be contain TOEFLEval in it? Then I can delegate the call to the base to evaluate GRE and delegate the call to the contained TOEFLEval object to evaluate TOEFL scores right?



Do we need to create more classes?

The above approach while may appear to be OCP compliant, leads to class proliferation. Instead of spending the time writing more classes, we can instead spend the time using the objects of those classes. Let's see how we can do that.

What's Decorator?

Decorator[2] is a pattern that shows us how to solve problems like this. Another way to understand decorator is to understand chaining. The criteria objects can be chained to achieve extensibility and agility (I have to use that word somewhere!).

Let's say I wake up one morning of an important meeting and look in the mirror and say "oh Venkat, you do not look good." What should I do, should I find some one else

instead to go to the meeting? No. I may shower, wear a nice shirt and pant, may be wear a tie, a tie pin, etc. Some one may decide to wear a makeup, ear rings, nose rings, tongue rings, etc! In other words, we decorate the object with other objects. That is the kind of idea we will follow here.

Decorator in Action

Let's modify the classes and come up with a different hierarchy. We will first create an abstract class named EvaluationCriteria as shown below:

```
public abstract class EvaluationCriteria
{
    public abstract boolean evaluate(Application theApp);
}
```

We will modify the Registrar to use this class instead:

```
public class Registrar
{
    public boolean evaluate(Application theApp, EvaluationCriteria criteria)
    {
        //...do what ever you have to here

        // when all else done

        boolean success = criteria.evaluate(theApp);

        // ... do what ever else you have to

        return success;
    }
}
```

Now, the GPAEval is derived from EvaluationCriteria as shown below:

```
public class GPAEval extends EvaluationCriteria
{
    public boolean evaluate(Application theApp)
    {
        // Code to do actual work like if (theApp.getGPA() > 3) result = true;

        System.out.println("GPAEval.evaluate called");
        return true;
    }
}
```

Here comes the trick. We will create a class called CriteriaLink as shown below:

```
public abstract class CriteriaLink extends EvaluationCriteria
{
    private EvaluationCriteria next;

    public CriteriaLink(EvaluationCriteria theNext)
    {
        next = theNext;
    }

    public boolean evaluate(Application theApp)
    {
        if(next != null)
            return next.evaluate(theApp);
        else
            return true;
    }
}
```

What's this class doing? It is an abstract class that maintains a link to the next EvaluationCriteria, thus forming a linked list. When evaluate is called on it, it simply forwards the request to the next object in the chain. It says, "what ever the other object says, I will go with it." (Like what I say when my wife is around!)

Now, let's see how we would write the GREEval and TOEFLEval classes:

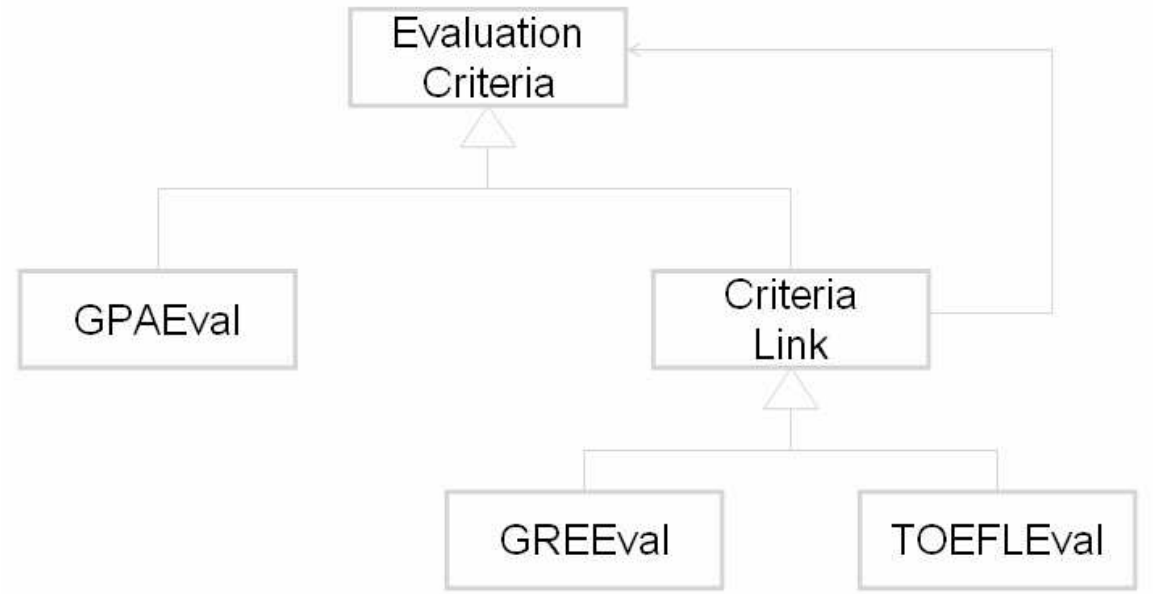
```
public class GREEval extends CriteriaLink
{
    public GREEval(EvaluationCriteria theNext)
    {
        super(theNext);
    }

    public boolean evaluate(Application theApp)
    {
        if(super.evaluate(theApp))
        {
            // Code to do actual work like if (theApp.getGRE() > ...) result = true;
            System.out.println("GREEval.evaluate called");
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

```
public class TOEFLEval extends CriteriaLink
{
    public TOEFLEval(EvaluationCriteria theNext)
    {
        super(theNext);
    }

    public boolean evaluate(Application theApp)
    {
         if(super.evaluate(theApp))
        {
             // Code to do actual work like if (theApp.getTOEFL() > ...) result = true;
            System.out.println("TOEFLEval.evaluate called");
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

What have we done so far? Let's visualize the code using the UML notation:



We have the Criteria link forming the chain. Such fundamental or leaf criteria like GPAEval fall in the bottom of the chain. Those additional criteria like GREEval are in the upper level of the chain. How would we use this now? Let's look at the TestCode to see that:

```
public class TestCode
{
    // Sample code to run our little app

    public static void main(String[] args)
    {
        Application anApp = new Application();
        Registrar reg = new Registrar();

        System.out.println("Running first eval");
        GPAEval aGPAEval = new GPAEval();

        reg.evaluate(anApp, aGPAEval);

        //Evaluate GRE and GPA
        System.out.println("Running second eval");
        GREEval aGREEval = new GREEval(new GPAEval());
        reg.evaluate(anApp, aGREEval);

        //Evaluate TOEFL and GPA
        System.out.println("Running third eval");
        TOEFLEval aTOEFLEval = new TOEFLEval(new GPAEval());
        reg.evaluate(anApp, aTOEFLEval);

        //Evaluate TOEFL, GRE and GPA
        System.out.println("Running fourth eval");
        EvaluationCriteria criteria = new TOEFLEval(aGREEval); // Chaining of criteria
        reg.evaluate(anApp, criteria);
    }
}
```

Notice how the criteria are being chained. If I want to say evaluate TOEFL, GRE and GPA, I could write it as

GPAEval aGPAEval = new GPAEval();
GREEval aGREEval = new GREEval(aGPAEval);
EvaluationCriteria criteria = new TOEFLEval(aGREEval);
reg.evaluate(criteria);

or I may also write (at the expense of readability):

EvaluationCriteria criteria = new TOEFLEval(new GREEval(new GPAEval()));
reg.evaluate(criteria);

If I want some other combination of criteria, I can readily write that without worrying about how to subclass one criteria from the other. The output of the program is shown here:

```
Running first eval
GPAEval.evaluate called
Running second eval
GPAEval.evaluate called
GREEval.evaluate called
Running third eval
GPAEval.evaluate called
TOEFLEval.evaluate called
Running fourth eval
GPAEval.evaluate called
GREEval.evaluate called
TOEFLEval.evaluate called
```

Consequences of using Decorator

One argument I have heard from people against Decorator is that it may be slower. There is no significant performance issue in terms of execution speed. Instead of calling the method on the base class, you end up calling the method on the object next in the chain. One significant difference is the number of objects you end up using. In the case of using inheritance alone, you end up with one object (assuming multiple implementation inheritance that is). In this case, you have objects chained together. From within the evaluate method of the Registrar object you have no way of knowing how many objects you have got. The Registrar can't make any decision based on the type of the object it thinks it received. This last point actually may be an advantage as trying to find that may be a violation of Liskov's Substitution Principle[1] any ways.

Conclusion

In this article we have shown, through an example, the use of Decorator pattern. This is a pattern that provides quite a bit of flexibility. It eliminates the needs for sub-classing. It is a pattern that is pretty effective when the object being used should appear to change its behavior.

References

1. Robert C. Martin, *Agile Software Development: Principles, Practices and Pattern*, Prentice Hall. Refer to "The Open Closed Principle" and "Liskov's Substitution Principle."
2. Erick Gamma, et. al., *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston, MA, 1994.