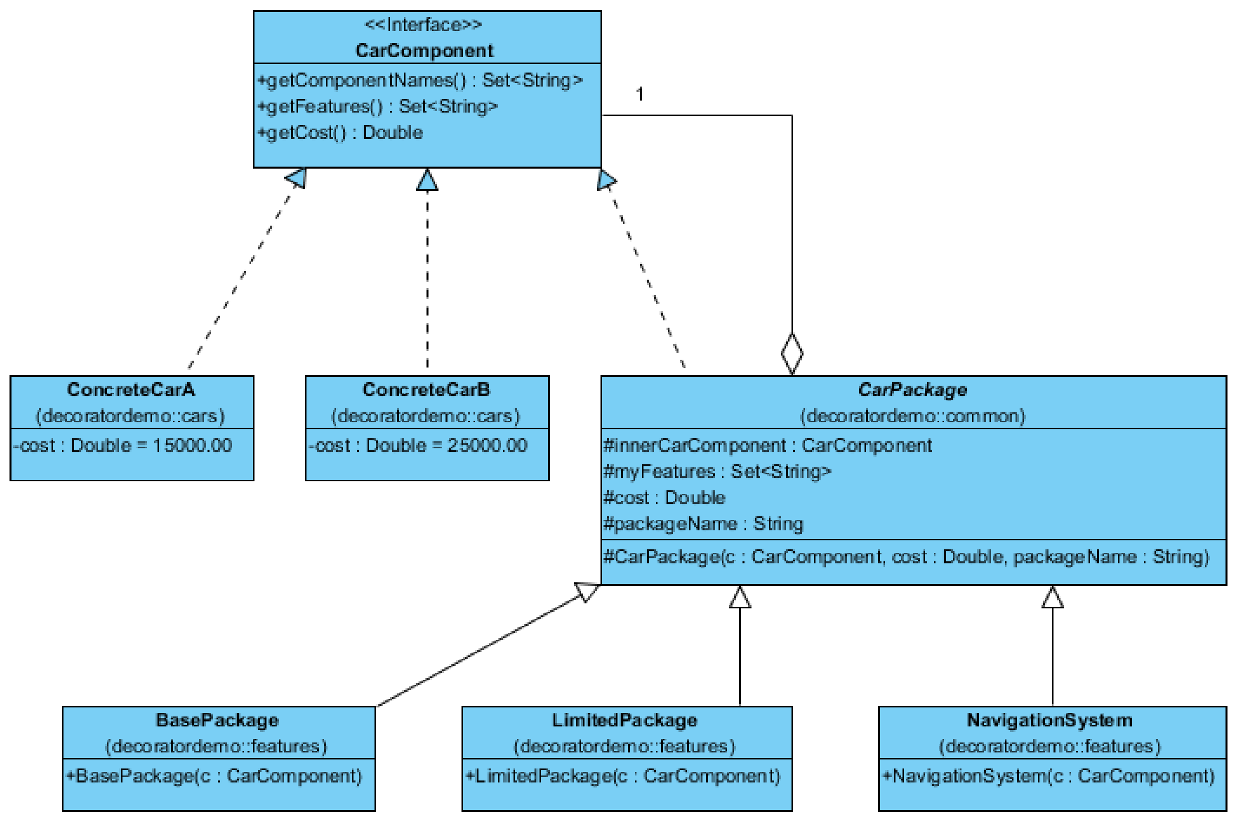
Chris Casola
2/20/2013
CS 4233

Decorator Pattern

The decorator pattern is an object-oriented design pattern that allows new behavior to be added to objects at runtime. It is a useful alternative to sub-classing, especially for cases where there are so many variations of an object that a class explosion would occur if sub-classing were used or where behavior needs to be added to specific objects rather than all objects of a certain class. The pattern is called a decorator because it works by wrapping an existing object inside a new object, with both objects implementing the same interface. Since both objects implement the same interface, once the original object has been “wrapped” by the decorator, all existing code will continue to work, however the decorator object will intercept all method calls and do some actions before asking the object inside the decorator to do its job. There can be multiple layers of decorators around an object, so decorators can wrap other decorators (Freeman, Robson, Bates, & Sierra, 2004).

Application of the decorator pattern can also be useful when adding new or special functionality to an existing system. Rather than modifying existing classes, which would violate the Open Close Principle, a decorator can be used to add special case behavior, without breaking or modifying existing code. The added functionality can simply be placed in a decorator, and then the class to be modified can simply be wrapped in the decorator (Kerievsky, 2004). A common situation where the decorator pattern would be applied in this way is when adding logging capabilities to a class. Using a decorator allows the developer to add code to write to the log without inserting this code just before or after methods in the existing class. Instead the logging code would be placed in a decorator, and when methods in the decorator are called, the log would be written to just before or after the inner, decorated class is called.

The decorator pattern relies on a common interface between the decorator and the class being decorated. The decorator can then contain a reference to the object being decorated. Existing code can then begin using a reference to the decorator in place of the reference to the original object. The class diagram below shows an example application of the decorator pattern using a car scenario.



<<Interface>>
CarComponent
+getComponentNames() : Set<String>
+getFeatures() : Set<String>
+getCost() : Double
1
ConcreteCarA
(decoratordemo::cars)
-cost : Double = 15000.00
ConcreteCarB
(decoratordemo::cars)
-cost : Double = 25000.00
CarPackage
(decoratordemo::common)
#innerCarComponent : CarComponent
#myFeatures : Set<String>
#cost : Double
#packageName : String
#CarPackage(c : CarComponent, cost : Double, packageName : String)
BasePackage
(decoratordemo::features)
+BasePackage(c : CarComponent)
LimitedPackage
(decoratordemo::features)
+LimitedPackage(c : CarComponent)
NavigationSystem
(decoratordemo::features)
+NavigationSystem(c : CarComponent)

The example above implements a system that allows packages (or groups of features) to be applied to cars. A similar system could be used in a car configurator or ordering form. The decorator pattern is used to apply, or wrap, packages around existing cars. Cars can be wrapped with more than one package (or decorator). The decorator is useful in this instance for calculating the total cost of a car and listing the features it offers. For example, when calling the getCost() method of ConcreteCarA a cost of $15000 will be returned. However, if a LimitedPackage decorator is applied to a ConcreteCarA, calling getCost() will cause the getCost() method in CarPackage to call the getCost() method in the decorated class (ConcreteCarA) which returns $15000, and then add the cost of itself so that a total greater than $15000 is ultimately returned.

The CarComponent interface defines the behavior that all concrete cars and CarPackages must provide. By using a common interface, concrete cars and CarPackages can be used in place of each other. In this example, the concrete CarPackage classes are decorators. Each instance of a concrete CarPackage maintains a reference to the class being decorated, so that calls to a decorator's public methods can delegate to the corresponding methods in the decorated class. While the decorator pattern does not require that concrete decorators (e.g. LimitedPackage and NavigationSystem) extend a common abstract class, using an abstract class removes some duplication of the basic code that handles delegation to the inner class being decorated. By employing a CarPackage abstract class, concrete decorators need only provide a constructor that sets the value of the fields inherited from parent abstract class (Christensen, 2010).

To use the classes in the diagram above one must simply instantiate a ConcreteCarA or ConcreteCarB and proceed to wrap them in decorators (or not). A car with no packages can be instantiated like this:

```
        CarComponent myCar = new ConcreteCarA();
```

Adding packages to the myCar object is simple. The LimitedPackage and NavigationSystem can be added like this:

```
        myCar = new LimitedPackage(new NavigationSystem(myCar));
```

After the line of code above is executed, myCar actually references a LimitedPackage, but since LimitedPackage implements the common CarComponent interface, it can continue to be treated as if it were a ConcreteCarA, or more generally a CarComponent.

Many real-world systems provide situations where the decorator pattern can be applied. The built-in Java IO API makes heavy use of the decorator pattern. Decorators are used to add functionality to the basic input and output streams. There are also decorators to make IO more efficient and to process data as it is read in or written out (Gardner & Manduchi, 2007). A commonly used class used when reading in characters is BufferedReader, which can decorate any class that implements the Stream interface. BufferedReader enhances the basic functionality of a stream by buffering input data. When users call readLine() on a BufferedReader, the BufferedReader does not always call readLine() on the underlying stream, rather it only asks for data from the underlying stream if the buffer is empty. In this way, BufferedReader prevents a large number of small IO requests from being sent to the operating system, resulting in many small disk reads.

A related situation where decorators would be useful is in a billing system. Suppose this billing system generates bills for a credit card company in a number of formats (printed on 8.5" x 11" paper, PDF, email, HTML) and a number of languages (English, Spanish, etc.). To allow for any combination of language and format, the decorator pattern can be applied. If the decorator pattern were not used, a separate class would need to be implemented for each combination of language and format, but with decorator, only one class is needed per format and per language. To create a PDF bill in Spanish, one would simply wrap the generic Bill in two decorators, SpanishDecorator and PDFDecorator. In this example, the order of the layering of decorators would likely be important. Language decorators should probably be beneath format decorators in the hierarchy. Another benefit to using the decorator pattern in this situation is that it makes supporting new language and formats simple. If the company determines a need to start generating bills in French, they can simply implement a FrenchDecorator and immediately generate bills in any format using the existing format decorators. Applying the decorator pattern not only reduces the amount of code that must be written or changed when adding new languages and formats, but it also makes the code more readable and easy to understand, primarily due to the drastically reduced number of classes.

The decision to choose the Decorator pattern may not always be clear. There are alternatives patterns that can be used in most situations. For example, the strategy pattern can often be used in many of the cases were decorator would apply. The patterns are similar in that they both allow the developer to move special-case behavior out of classes and allow a class to use different implementations or types of the special behavior. However, there are some important differences related to how Strategy and Decorator instances are actually applied to an object. With Decorator, a separate Decorator instance is required for each class that is being decorated. This is not necessarily required with a strategy pattern, especially if the strategy relies on static methods or is implemented using the Singleton pattern. The decorator pattern has an advantage in that the class being decorated does not need to be aware of its decorators, any number of decorators, as long as they implement the same interface as the class being decorated, can be applied. Classes making use of the strategy pattern must be aware of the different strategies that are available and must apply them correctly. Situations where it is feasible for a

Decorator to implement the common interface, and where the interface does not require a large number of public methods, are optimal for application of the decorator pattern (Kerievsky, 2004).

References

Christensen, H. B. (2010). *Flexible, Reliable Software: Using Patterns and Agile Development.* Boca Raton, FL: Chapman & Hall/CRC.

Freeman, E., Robson, E., Bates, B., & Sierra, K. (2004). *Head First Design Patterns.* Sebastapol, CA: O'Reilly Media.

Gardner, H., & Manduchi, G. (2007). *Design Patterns for e-Science.* New York: Springer.

Kerievsky, J. (2004). *Refactoring to Patterns.* Boston: Addison-Wesley Professional.