Chapter 12- Decorator Design Pattern

Contents

12.1 Introduction

The decorator pattern is used to extend the functionality of an object without changing the physical structure of the object [2]. The extension is either in terms of elaborating on the state of the object or in terms of behaviour. A combination of both state and behaviour is also possible which implies that these extensions can be stacked on the object and rather than using subclassing, which is a compile-time solution, the extensions can be applied at runtime.

In the sections that follow, an overview of the structure of the decorator pattern will be given along with an explanation of how the decorator can be applied. The two examples presented will illustrate how the decorator can be implemented. The first example will decorate the composite tree developed in Lecture note 14, while the second example will show how decorations of a till slip can change how the till slip is structured.

12.2 Decorator Pattern

12.2.1 Identification

Name	Classification	Strategy
Decorator	Structural	Delegation (Object)
Intent		
Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. ([1]:175)		

12.2.2 Structure

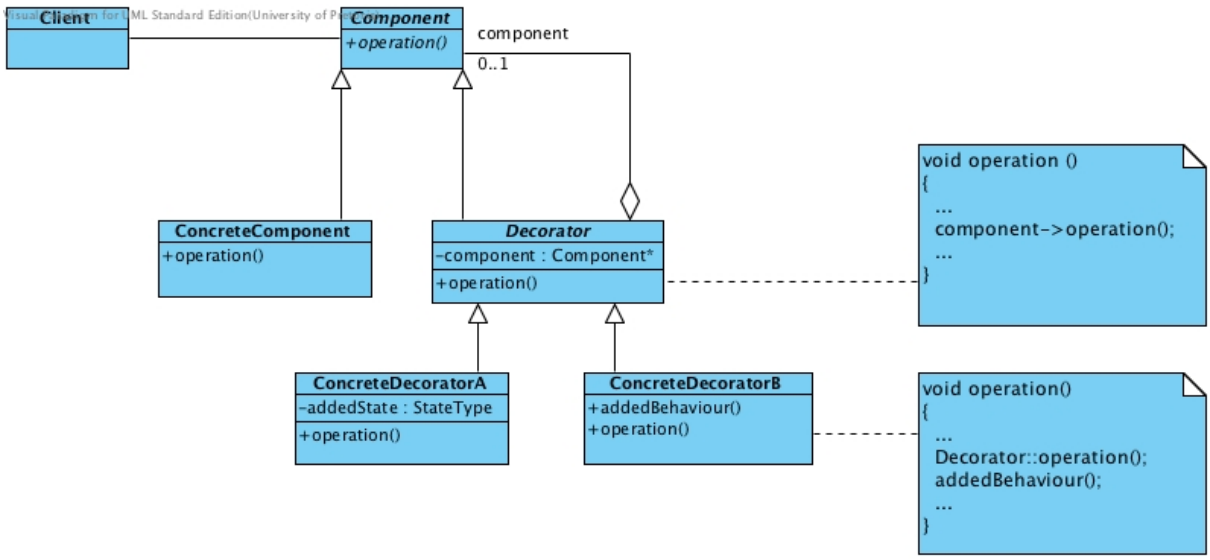

Figure 1: The structure of the Decorator Pattern

12.2.3 Participants

Component

- interface for objects that can have responsibilities dynamically added to them

ConcreteComponent

- the object to which the additional responsibilities can be attached

Decorator

- defines a reference to a Component-type object

ConcreteDecoratorA

- adds state-based responsibilities to the component

ConcreteDecoratorB

- adds behavioural-based responsibilities to the component

12.3 Decorator Explained

The structure of the Decorator is similar to the Composite. The main differences are the number of components related to; and the specialisations the composite and decorator may have. The composite comprises of multiple components, while decorators may or may not comprise of a component. The composite class is defined as a concrete class, while the decorator class is abstract and concrete decorator participants specialise the decorator. The second difference ensures that the composite builds a tree data structure, while the decorator only a list data structure.

As with the composite, it is the concrete components that are to be decorated and there may be multiple of these. A single concrete component object may also have more than decorator instance applied to it. Figure 2 provides a few combinations of decorators that may be applied to the concrete component class.

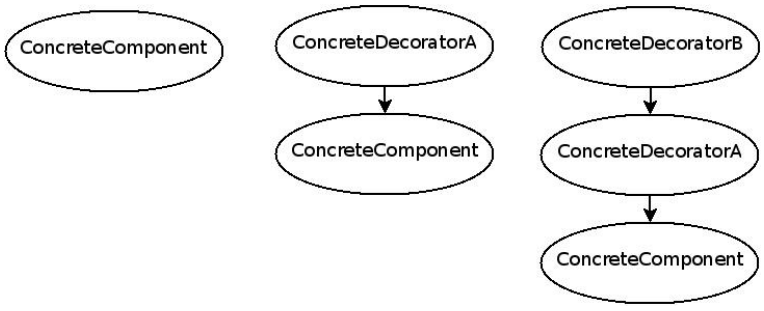

Figure 2: Examples of list structures of objects created by the Decorator

It is conceivable that a list may be decorated with the same concrete decorator more than once, it is however not always practical. The order of the application of the concrete

decorator should also be independent of one another and the net effect should be the same. The reason being that decorated objects should behave as if defined as a single large object with all the additional responsibility embedded in it.

12.3.1 Code improvements achieved

The advantage of applying the decorator design pattern is that objects of the concrete component provide the basic functionality expected of such a component. Any additional responsibility, be it state-based or behavioural-based, can be seen as adding value to the object, but is not embedded in the object. This design separates the concerns of required functionality and "nice to haves".

12.3.2 Implementation Issues

Two types of concrete decorators are defined, those that add state-based responsibilities and those that add behavioural-based responsibilities. It is easier to implement the state-based concrete decorators than it is to implement the behaviour-based responsibility version. It is also conceivable that both these types of responsibilities are included in a single concrete decorator class.

The same issues, as with Composite, arise when dealing with anonymous references.

12.3.3 Related Patterns

Adapter
: Changes the interface to an object while the Decorator only changes responsibilities.

Composite
: A Decorator can be seen as a Composite with only one component that has added responsibility.

Strategy
: The Strategy pattern changes the inner workings of an object while the Decorator changes the looks.

12.4 Example

12.4.1 Tree

To decorate the `BaseNode` of the tree example presented in Chapter 11, the decorator pattern is applied to the `Tree` and `BaseNode` classes as shown in Figure 3. Notice that destructors have been added to the hierarchy in order to ensure that the decorator clears the memory when the first object in the list goes out of scope. The destructors for the classes `Tree`, `BaseNode`, `BehaviourDecorator` and `StateDecorator` are all defined as virtual and an implementation with no statements is provided. The destructor of the `Decorator` class deletes the instance referred to by the attribute `component`.

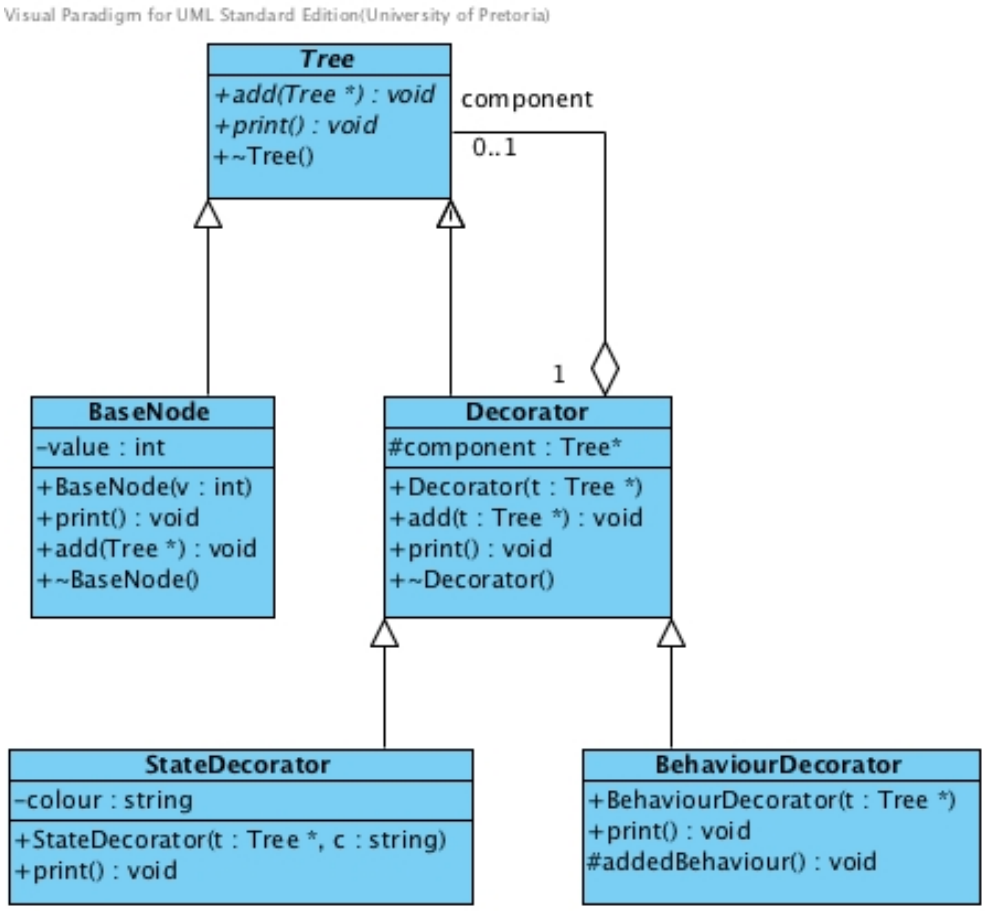

Figure 3: Decorating the Tree: showing only the decorator pattern

Both the `print` functions defined in the concrete decorator participants make calls to the parent `print` function to ensure that all chained prints are executed. Sample implementations of the `print` functions are given.

```
void Decorator::print()
{
    component->print();
}

void StateDecorator::print()
{
    cout << "!" << colour << "-";
    Decorator::print();
    cout << "!";
}

void BehaviourDecorator::print()
{
    addedBehaviour();
    Decorator::print();
}
```

Figure 4 shows how the Composite and Decorator design patterns can be used together. It is now possible to decorate the composite participant, `IntermediateNode`, as well.

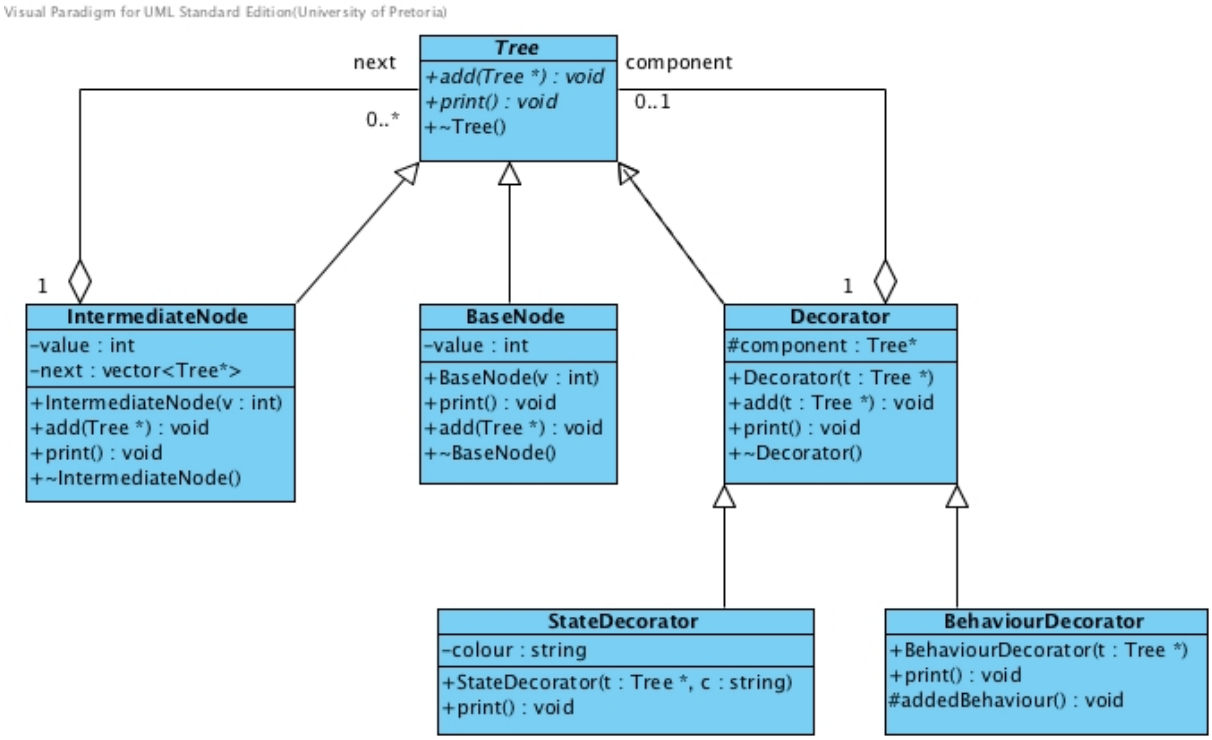

Figure 4: Decorating the Composite Tree

12.4.2 SalesTicket

This example illustrates how the decorator can be applied to change the "look" of a till slip (also referred to as a sales ticket) and customise it for a particular situation. A typical till slip has a header section where the name of the shop is printed, a body where a list of purchases are given and a footer with some friendly message or information. The basic functionality of the till slip is to provide the customer with the items listed in the body of the slip. The shop name displayed in the header and the greeting printed in the footer are "nice to have" and provide an individual identity for the till slip. These added responsibilities can easily be included by decorating the till slip with a header and a footer that is customisable for the particular shop.

Figure 5 presents the UML class diagram for the description of the till slip given above. The class `SalesTicket` represents the *ConcreteComponent* participant of the design pattern. `SomeClass` represents the *Decorator participant* and the classes `Header1`, `Header2`, `Footer1` and `Footer2` the *ConcreteDecorator participant*. *SalesOrder* represents the client for the design pattern.

Understanding how the pattern works can be tricky and therefore some coding aspects of the pattern are highlighted, specifically how `printTicket` is implemented for the participating classes. The `printTicket` of the `SalesTicket` class prints the body of the till slip. `SomeClass` first checks whether the `Component` has been decorated before it called the relevant `printTicket` function for linked component. The functions for both the *Header*

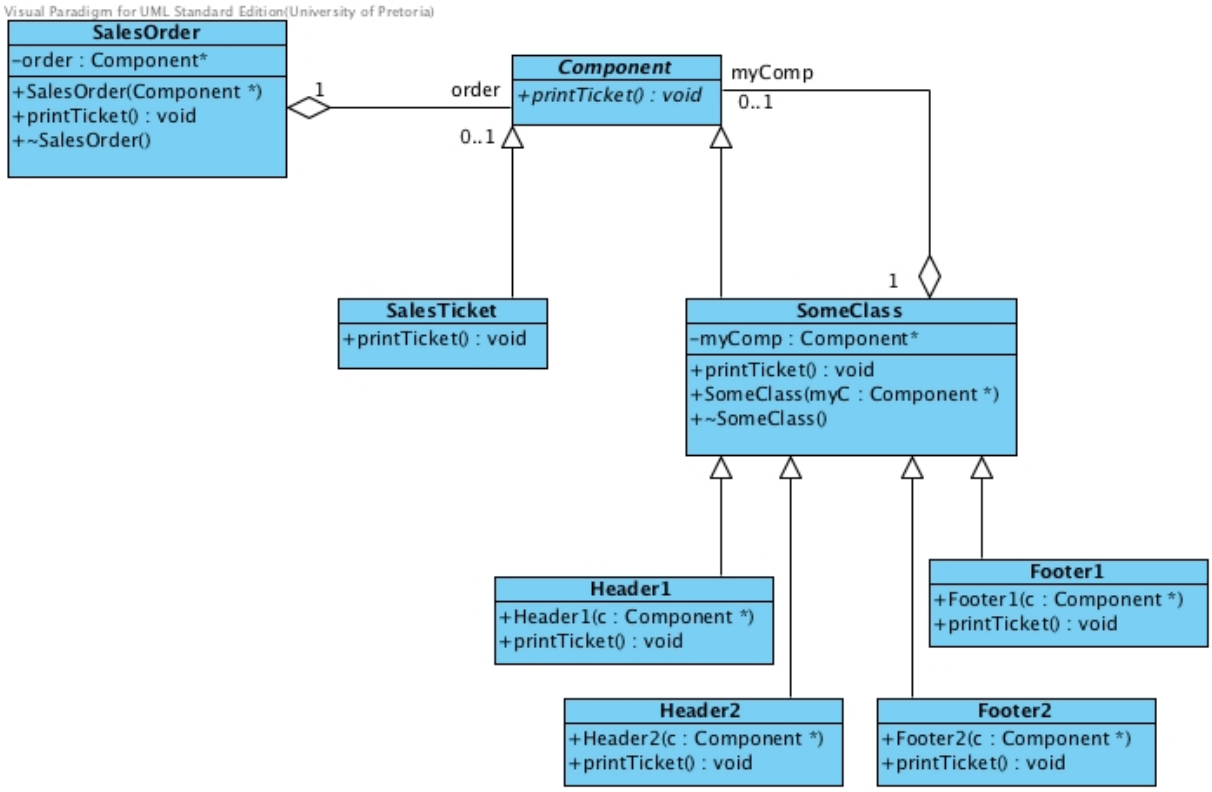

Figure 5: Printing sales tickets with the decorator

classes must first print their message before passing the printing on to the next component. The *Footer* classes do this in reverse to ensure that the relevant text is displayed at the bottom of the till slip.

```
void SalesTicket::printTicket()
{
    cout<<"Cash Sale Ticket"<<endl;
    cout<<"List of items purchased"<<endl;
    cout<<"Item"<<'\t'<<"Quantity"<<'\t'<<"Price"<<endl;
    // print the items out
    cout<<"TOTAL:"<<endl;
}

void SomeClass::printTicket()
{
    if (myComp)
        myComp->printTicket();
}

void Header1::printTicket()
{
    cout<<"Welcome to the Crazy Zone"<<endl;
    SomeClass::printTicket();
}
```

```
void Footer1::printTicket()
{
    SomeClass::printTicket();
    cout << "It was a pleasure doing" << " business with you"<<endl;
}
```

12.5 Exercises

1. Is it possible for the Composite design pattern to be restricted to build a list data structure? Explain.

2. For the combination of the Decorator and Composite given in Figure 4, identify the participants of both patterns.

3. In the sales ticket example given in Figure 5 it is possible to construct the concrete classes using the default constructor. Doing so will cause memory problems within the classes. Explain how you would go about to fix the problem.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass, 1995.

[2] Wikipedia. Decorator pattern, 2012. URL `http://en.wikipedia.org/wiki/Decorator_pattern`. [Online; accessed 27 August 2012].