Chapter 20

Decorator

20.1 The Problem

Alphatown contacts us with a new requirement. They have a single pay station that often overflows with 5 cent coins and would like to find out if there is any pattern in this odd phenomenon. Thus, they would like the pay station to maintain a log file of all coins entered into each pay station, recording coin type and a time stamp. The log file of the pay station could then be examined to detect a pattern.

One entry of the log file could look like:

```
5 cents  : 08:35
10 cents : 08:35
25 cents : 08:47
...
```

20.2 Composing a Solution

I first identify the behavior that must vary. This is clearly associated with the coin entry and thus the responsibility *Accept payment*. Thus one plausible path is to factor out payment acceptance using the strategy pattern and then provide two different concrete payment accept strategies: the standard one and one that in addition makes entries in the log file.

There is a twist here, though. We are actually not required to *vary* the behavior rather we are required to provide *additional* behavior over and above the standard one. This allows me to proceed by another path.

Figuratively speaking I could get my job done by hiring a person to stand in front of the pay station. This person would mediate all handling of the pay station for the customer. He would accept each coin from the customer, note coin type and time in his log book, and then of course put the coin into the pay station. He would push the buy button when requested, take the receipt and hand it over to the customer. This way we would get the intended log file without changing the pay station software at all. Thus, the hired person essentially has the same interface as the pay station but adds functionality to one of the actions, namely inserting coins.

Talking about software, I can *compose* the required behavior by putting *an intermediate object with the same interface* in front of the pay station object. All requests are ultimately passed on to the pay station object, but additional processing can be made in the intermediary. The client does not know that an intermediate object is used because it responds to exactly the same interface and protocol. A sequence diagram for the add payment scenario will then look like in Figure 20.1.

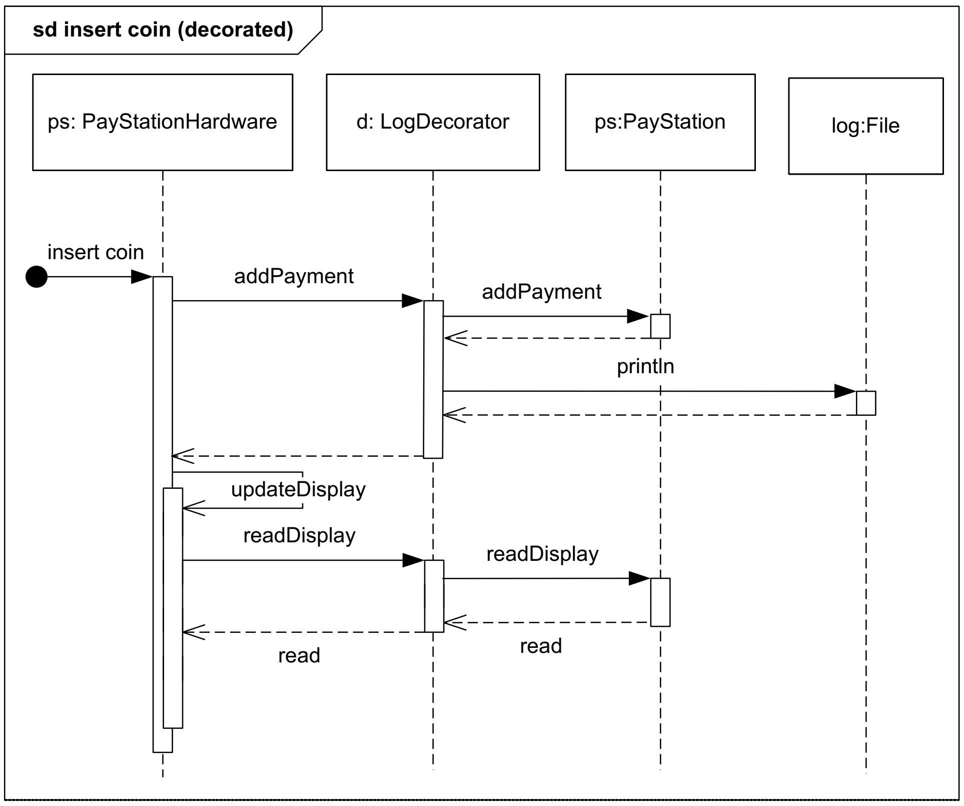


Figure 20.1: Decorating with logging payments.

This is the essence of the DECORATOR pattern: I have *decorated* the pay station's behavior with additional behavior, the logging of coins. The graphical user interface (or hardware) object cannot tell that it does not operate a "normal" pay station as the decorator has exactly the same interface. The decorator itself delegates any request to the decorated object. A fine example of the ① *program to an interface* and ② *favor object composition* principles at work. The decorator code itself is simple, basically just delegation code.

Listing: chapter/decorator/src/paystation/domain/LogDecorator.java

```
package paystation.domain;
import java.util.Date;
```

```
/** A  PayStation decorator that logs coin entries.
*/
public class LogDecorator implements PayStation {
  private PayStation paystation;
  public LogDecorator( PayStation ps ) {
    paystation = ps;
  }
  public void addPayment( int coinValue )
          throws IllegalCoinException {
    System.out.println( ""+coinValue+" cents: "+new Date() );
    paystation.addPayment( coinValue );
  }
  public int readDisplay() { return paystation.readDisplay(); }
  public Receipt buy() { return paystation.buy(); }
  public void cancel() { paystation.cancel(); }
}
```

Exercise 20.1: Draw the UML class diagram for the structure of the logging decorator for the pay station.

Exercise 20.2: Discuss whether I could have used a polymorphic technique by subclassing instead of using a decorator.

☞ Study the source code provided in folder *chapter/decorator* on the web site.

20.3 The Decorator Pattern

The key aspect of DECORATOR (design pattern box 20.1, page 17) is composition, the ② *favor object composition* principle. Its intent is

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Rephrasing this, decorator allows you to dynamically add responsibilities to a role. The underlying object is unaffected and in this way a decorator can help keeping the number of responsibilities of a role low—I can always decorate them later. For instance, I would definitely not feel comfortable about adding a "Log time of coin entry" to the pay station role, it is not a responsibility that belongs naturally there. The decorator helps me out of this dilemma. In addition, decorators enjoy several benefits and a few liabilities as outlined below.

First, any object implementing the interface may be decorated. Alternative implementations of the PayStation interface may just as easily be decorated with the LogDecorator. Contrast this to the inheritance based solution: if I had subclassed the Alphatown class to add coin logging then it had become specific for the Alphatown variant. The decorator solution is general.

Second, decorators can easily be **chained** so one decorator decorates another decorator and so forth until the final, base, object. For instance, I could define other pay station decorators that reject 10 cent coins, or counts the number of cancel operations called. I could then combine all behaviors with a declaration like

```
Paystation ps =
  new Reject10CentDecorator(
   new LogDecorator(
    new CountCancelDecorator(
     new PayStation(
      new BetaTownFactory()
  ))));
```

Third, the coupling between the decorator and the decorated component is dynamic which means that you can "rewire" it at run-time. As an example, you can add and remove the logging behavior dynamically to a pay station instance by code like this (both payStation and decoratee are of type PayStation):

```
if (payStation == decoratee) {
  // enable the logging by decorating the component
  decoratee = payStation; // but remember the component
  payStation = new LogDecorator(payStation);
} else {
  // remove logging by making payStation point to
  // the component object once again
  payStation = decoratee;
}
```

Note that the decorated object keeps its state; it knows the amount entered in the current transaction and what to display; and is unaffected by any decorators. Again, this is not possible in an inheritance based design.

> ☞ Argue why state cannot be kept when adding/removing behavior if an inheritance based design was adopted.

The strength of the decorator: the dynamic and compositional nature; is also its liability. In a system relying heavily on decorators, the behavior of objects is assembled at run-time and this assembly process may be distributed in the code. Each fraction of the resulting behavior is defined in separate, small, classes where most methods are simply delegations. Thus *analyzability* suffers because there is no localized place in the code where "the algorithm is written". The result is that the system is hard to debug and learn.

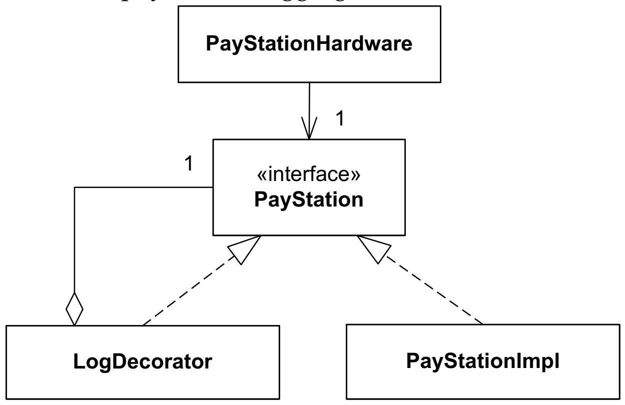
In the pay station case above I simply made all the delegations directly in the LogDecorator class. Of course this will lead to a lot of redundant code if several different pay station decorators are needed. In that case, it is better to define an (abstract) class that defines all the delegation and only provide additional behavior in its subclasses, as shown in the description in pattern box 20.1.

One example of a decorator is the JScrollPane class from the `javax.swing` package. A JScrollPane is a subclass of Component, the basic graphical user interface unit in the Java graphical user interface library. When you construct a scroll pane it takes a Component instance as parameter. Thus the component is decorated with scroll bars to allow panning of the underlying component. The Java Collection Framework also contains *wrapper implementations* of the common types of collections which are decorators.

20.4 Selected Solutions

Discussion of Exercise 20.1:

The class diagram for the pay station logging decorator looks like this.



Discussion of Exercise 20.2:

You could subclass the implementation class for Alphatown's pay station and override the addAmount method. In this method you could add the coin logging behavior.

20.5 Review Questions

Describe the DECORATOR pattern. What problem does it solve? What is its structure and what is the protocol? What roles and responsibilities are defined? What are the benefits and liabilities?

Explain how decorators can be *chained*. How and why does this work on the code level? What are the consequences?

Explain why and how you can add and remove logging behavior at run-time while still retaining the pay station object's state (like amount earned, payment entered in this buy session, etc.).

20.6 Further Exercises

Exercise 20.3:

Walk in my footsteps. Introduce the DECORATOR that can wrap a pay station and log all coins entered into the machine.

Exercise 20.4:

Make a decorator that adds behavior to the pay station such that it refuses to accept more than 10 coins of value 5 cents since the last buy operation. Combine it with the previous decorator to make a decorator chain.

Exercise 20.5:

The Alphatown municipality decides that in the time interval between 08:00 PM in the evening and 07:00 AM in the morning parking is free. Thus during this time coins are simply returned (that is, acts as if cancel was pressed).

Sketch a design that handles this requirement using a decorator.

Exercise 20.6:

Describe the roles for the design resulting from exercise 20.3 and/or 20.4 by a role diagram.

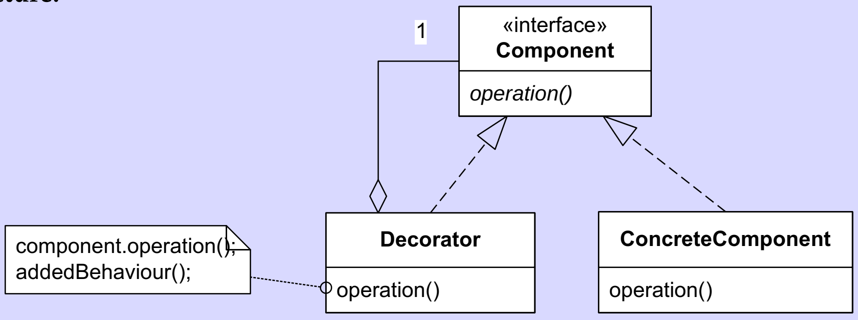
[20.1] Design Pattern: Decorator

Intent Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Problem You want to add responsibilities and behavior to individual objects without modifying its class.

Solution You create a decorator class that responds to the same interface. The decorator forwards all requests to the decorated object but may provide additional behavior to certain requests.

Structure:



Roles **Component** defines the interface of some abstraction while **ConcreteComponents** are implementations of it. **Decorator** delegates all method calls to the decorated **Component** object and adds additional behaviour.

Cost - Benefit Decorators allow *adding or removing responsibilities at run-time* to objects. They also allow *incrementally adding responsibilities* in your development process and thus help to keep the *number of responsibilities of decorated components low*. Decorators can provide *complex behavior by chaining* decorators after one another. A liability is that you end up with *lots of little objects* that all look alike, this can make understanding decorator chains difficult. The delegation code for each method in the decorator is a bit *tedious to write.*