Avoid Excessive Subclassing with the Decorator Design Pattern

by Barry A. Burd and Michael P. Redlich

Introduction

All developers apply design patterns of one kind or another. Some developers do this consciously; other apply patterns without even knowing it. Used appropriately, design patterns help you develop software that's robust and easy to maintain. But design patterns aren't panaceas. If you apply a design pattern in the wrong context, you can do more harm than good. The best advice is to learn and understand design patterns before applying them to your software projects.

This article introduces the Decorator design pattern, one of the 23 design patterns defined in the legendary 1995 book *Design Patterns – Elements of Reusable Object-Oriented Software*. The authors of this book, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, are known affectionately as the "Gang of Four." (So popular is this book that the Gang of Four has its own acronym – GoF.)

Let's start by reviewing the concept of a design pattern.

Design Patterns

The GoF book states:

"Design patterns are recurring solutions to software design problems you find again and again in real-world application development. Patterns are about the design and interaction of objects, as well as providing a communication platform concerning elegant, reusable solutions to commonly encountered programming challenges."

It's important to note that design patterns aren't only about **design**. Implied or not, the **interaction** of objects is also very important.

The GoF book defines 23 design patterns. The patterns fall into three categories:

- A creational pattern abstracts the instantiation process.
- A structural pattern groups objects into larger structures.
- A behavioral pattern defines better communication among objects.

The Decorator design pattern fits into the structural category, and is one of the most widely used patterns. As a matter of fact, the Java API makes frequent use of the Decorator pattern. So, without further ado…

The Decorator Pattern

In order to describe a pattern, you explain the pattern's intent, the pattern's motivation, how the pattern is implemented, and any consequences from the pattern's use. To tie everything together, you present a UML diagram.

Intent

We again reference the GoF book to describe the intent of the Decorator design pattern:

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

Some authors refer to decorators as "wrappers" because a decorator wraps itself around another object.

Later in this article we explain why the Decorator design pattern provides a flexible alternative to subclassing as we demonstrate why subclassing can be dangerous.

Motivation

The motivation for the Decorator design pattern is to eliminate the consequences of excessive subclassing. For example, consider a sports application where we need to model some major sports leagues.

```
public abstract class Sports
    {
    }

public class Baseball extends Sports
    {
    }

public class Football extends Sports
    {
    }
```

This isn't an example of excessive subclassing and, for all we know, this application won't grow into an example of excessive subclassing. Adding some of the other major sports such as basketball, hockey, and soccer is straightforward, and it's easy to predict common object behavior for all sports.

Excessive subclassing occurs when an application requires a more complex model. The model may need to account for all possible common object behaviors, and these behaviors can be difficult to predict. Even if you overcome the initial design hurdles, refactoring can be a major pain.

Consider an application that models employees in a laboratory environment. The kinds of employees include a principle investigator, a research technician, and an administrative assistant. Figure 1 is our initial UML diagram for this application.

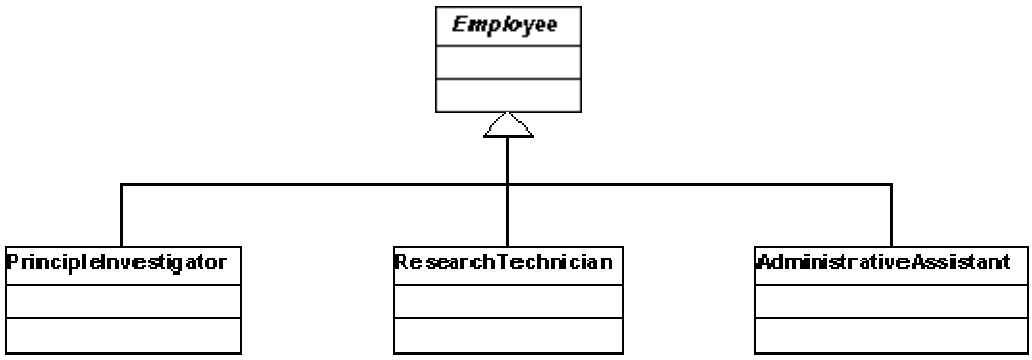

Figure 1: The initial UML diagram for the laboratory employee application

Along with the model, we have some source code.

```
public abstract class Employee
    {
    }

public class PrincipleInvestigator extends Employee
    {
    }

public class ResearchTechnician extends Employee
    {
    }

public class AdministrativeAssistant extends Employee
    {
    }
```

At this point the model is very simple. But employees can have other roles aside from their regular work assignments – roles like computer security officer, blood drive canvasser, or safety captain. So, we should include these additional roles in our model. We can add more objects by extending the **`Employee`** class:

```
public class PrincipleInvestigatorAndComputerSecurityOfficer extends Employee
    {
    }

public class PrincipleInvestigatorAndBloodDriveCanvasser extends Employee
    {
    }

public class PrincipleInvestigatorAndSafetyCaptain extends Employee
    {
    }

public class ResearchTechnicianAndComputerSecurityOfficer extends Employee
    {
    }

public class ResearchTechnicianAndBloodDriveCanvasser extends Employee
    {
    }

public class ResearchTechnicianAndSafetyCaptain extends Employee
```

```
    {
    }

public class AdministrativeAssistantAndComputerSecurityOfficer extends Employee
    {
    }

public class AdministrativeAssistantAndBloodDriveCanvasser extends Employee
    {
    }

public class AdministrativeAssistantAndSafetyCaptain extends Employee
    {
    }
```

By continuing to extend the abstract **`Employee`** base class, our updated UML diagram looks like the one in Figure 2.

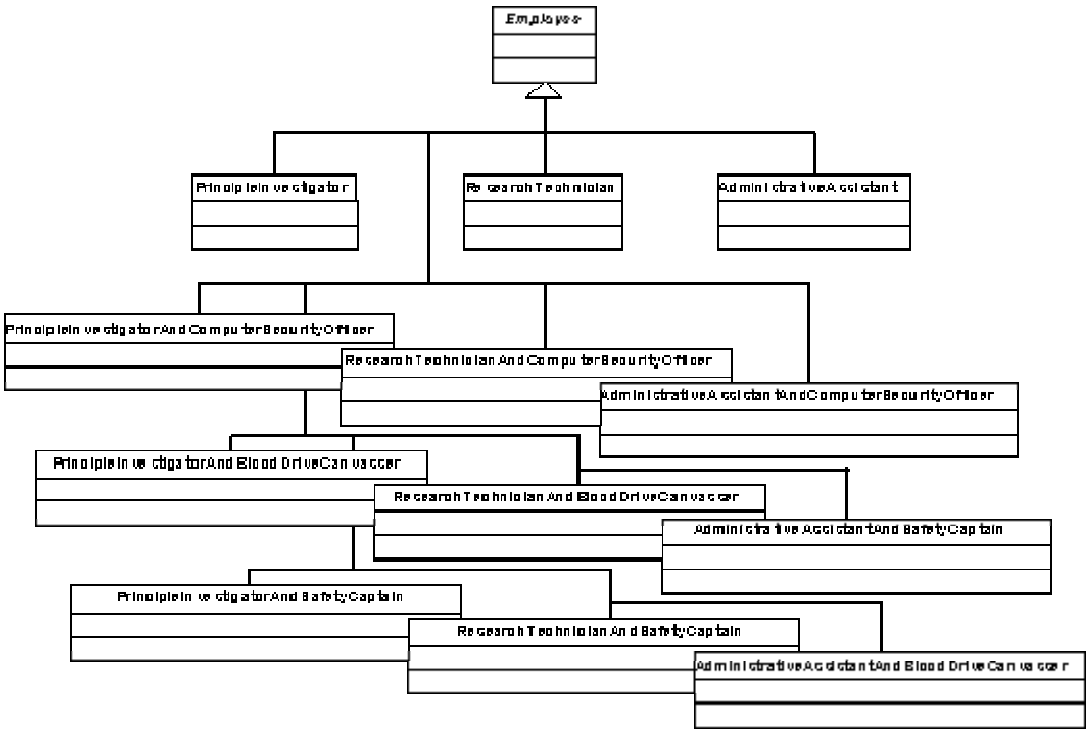

Figure 2: The updated UML diagram for the laboratory employee application

This approach isn't promising. With this approach, we have to account for every combination of employee and employee roles. The number of combinations becomes unmanageable.

We may be tempted to solve the problem using multiple inheritance (in C++) or by implementing interfaces (in Java). But solutions like these aren't dynamic. You can define

```
public class AdministrativeAssistant extends Employee
    implements BloodDriveCanvasser, SafetyCaptain
    {
    }
```

But then you're stuck with a big, bulky class – a class whose functionality doesn't adapt itself to each individual employee.

You can try adding fields to the Employee class:

```
public abstract class Employee
    {
    boolean isComputerSecurityOfficer;
    boolean isSafetyCaptain;
    boolean isBloodDriveCanvasser;
    }
```

But that's not very useful. Adding fields solves only the bookkeeping problem. The fields keep track of employee roles, but the fields don't enforce appropriate behavior for each of the roles.

UML Diagram

Figure 3 shows the official UML diagram for the Decorator pattern.

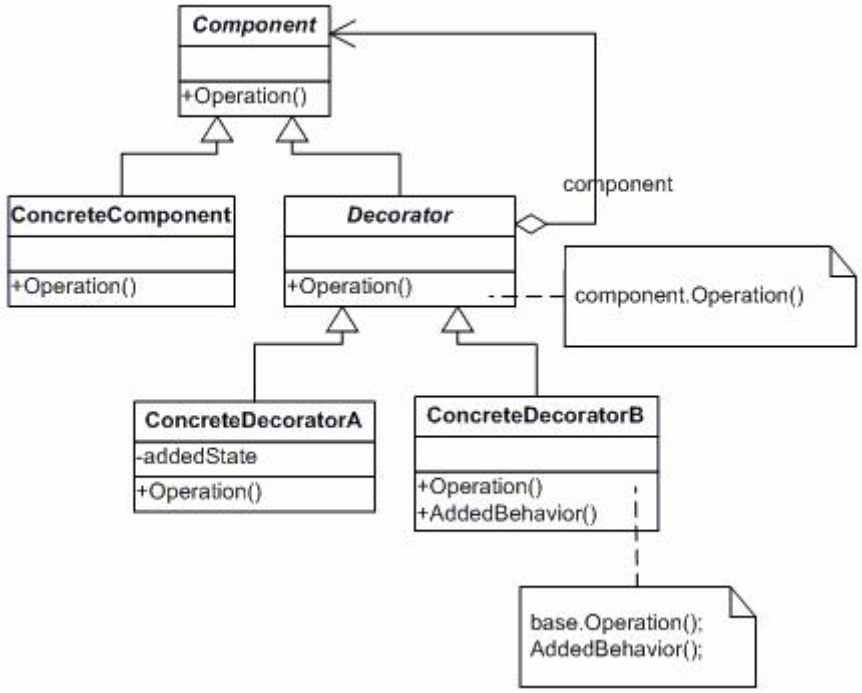

Figure 3: The official UML diagram for the Decorator design pattern

Let's walk through all the class diagrams.

- The **`Component`** class is any abstract class that we want to decorate. Even before being decorated, this class has some defined behavior.
  In our **`Employee`** class, the defined behavior may deal with hire date, benefits, or other such things.
- The **`ConcreteComponent`** class is a concrete type derived from **`Component.`** The **`ConcreteComponent`** has some functionality dynamically added to it.
  For example, our **`PrincipleInvestigator`** class may have code to keep track of the investigator's research projects.
- The **`Decorator`** class is an abstract class This **`Decorator`** class has two important features:
  - The **`Decorator`** is derived from **`Component`**.
  - The **`Decorator`** contains an instance of **`Component`**.

This may seem strange, but it's an important feature of the Decorator design pattern.

- The **component** variable refers to an instance of the **Component** class. Many **Decorator**s may share this instance of **Component**.
  For example, a variable may refer to a **PrincipleInvestigator**, and then be shared by the **SafetyCaptain** and **BloodDriveCanvasser** decorators.
- The **ConcreteDecoratorA** and **ConcreteDecoratorB** classes are concrete types inherited from **Decorator.**
  In our ongoing lab employee example, the concrete decorators **SafetyCaptain**, **BloodDriveCanvasser**, and **ComputerSecurityOfficer** can dynamically add additional functionality to a **PrincipleInvestigator** instance.

The abstract **Decorator** class is a subclass of **Component** but it also contains an instance of **Component**. It's like a robot in a science fiction movie – a big, human-like form with an ordinary person sitting in a driver's seat somewhere inside. The driver has arms, legs and a head, but the robot that contains the driver also has arms, legs and a head. Now imagine that someone invents an even bigger robot – one that's capable of having the first robot in its humongous driver's seat. Now you have the new human-like robot containing the original robot, which in turn contains the human driver. Everything in the system is human-like (with arms, legs and a head) so everything in the chain can do the kinds of things the original human can do (like battle with evildoers, or save innocent babies).

In the lab employee example, the **SafetyCaptain** can decorate an **Employee**. With a particular **Employee** instance in the driver's seat, the **SafetyCaptain** can do exactly what the original **Employee** can do – have a particular hire date, set of benefits, and other things. But being subclassed in some way from **Employee**, the **SafetyCaptain** can be further decorated by the **BloodDriveCanvasser**, making this employee both a **SafetyCaptain** and a **BloodDriveCanvasser**. Best of all, the decorating is done (and if necessary, undone) at runtime.

Using the Decorator

Here's a refactored design for the lab employee application.

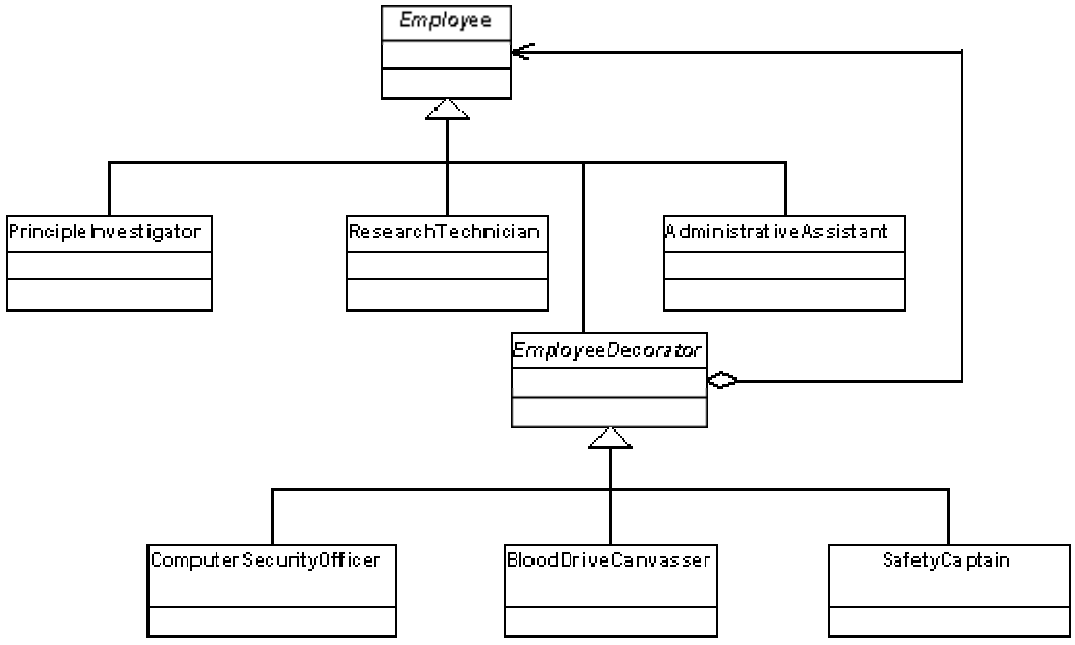

Figure 4: The refactored UML diagram for the laboratory employee application

The source code for this refactored diagram is in Listings 1 to 8.

```
public abstract class Employee
    {
    private String title = "Employee";

    public String getTitle()
        {
        return title;
        }

    public void setTitle(String title)
        {
        this.title = title;
        }

    public abstract String getResponsibility();
    }
```

Listing 1: The Employee (Component) class.

```
public class PrincipleInvestigator extends Employee
    {
    public PrincipleInvestigator()
        {
        super.setTitle("Principle Investigator");
        }

    public String getResponsibility()
        {
        return "Analyze laboratory data";
        }
    }
```

Listing 2. The PrincipleInvestigator (ConcreteComponent) class.

```
public class ResearchTechnician extends Employee
    {
    public ResearchTechnician()
        {
        super.setTitle("Research Technician");
        }

    public String getResponsibility()
        {
        return "Acquire laboratory data";
        }
    }
```

Listing 3. The ResearchTechnician (ConcreteComponent) class.

```
public class AdministrativeAssistant extends Employee
    {
    public AdministrativeAssistant()
        {
```

```
        super.setTitle("Administrative Assistant");
        }

    public String getResponsibility()
        {
        return "Provide secretarial support";
        }
    }
```

Listing 4. The AdministrativeAssistant (ConcreteComponent) class.

```
public abstract class EmployeeDecorator extends Employee
    {
    public abstract String getTitle();
    }
```

Listing 5: The EmployeeDecorator (Decoratator) class.

```
public class ComputerSecurityOfficer extends EmployeeDecorator
    {
    Employee employee;

    public ComputerSecurityOfficer(Employee employee)
        {
        this.employee = employee;
        }

    public String getTitle()
        {
        return employee.getTitle() + "\n\tComputer Security Officer";
        }

    public String getResponsibility()
        {
        return employee.getResponsibility() + "\n\tand handle computer security issues";
        }
    }
```

Listing 6: The ComputerSecurityOfficer (ConcreteDecoratator) class.

```
public class BloodDriveCanvasser extends EmployeeDecorator
    {
    Employee employee;

    public BloodDriveCanvasser(Employee employee)
        {
        this.employee = employee;
        }

    public String getTitle()
        {
        return employee.getTitle() + "\n\tBlood Drive Canvasser";
        }

    public String getResponsibility()
        {
        return employee.getResponsibility() +
            "\n\tand canvass employees for the quarterly blood drive";
        }
```

```
    }
```

Listing 7: The BloodDriveCanvasser (ConcreteDecorator) class.

```
public class SafetyCaptain extends EmployeeDecorator
    {
    Employee employee;

    public SafetyCaptain(Employee employee)
        {
        this.employee = employee;
        }

    public String getTitle()
        {
        return employee.getTitle() + "\n\tSafety Captain";
        }

    public String getResponsibility()
        {
        return employee.getResponsibility() + "\n\tand perform quarterly safety inspections";
        }
    }
```

Listing 8: The SafetyCaptain (ConcreteDecorator) class.

Listing 9 is the client application. Let's walk through some of this code. We start by creating two principle investigator objects, **`investigator01`** and **`investigator02`** using the following statements:

```
Employee investigator01 = new PrincipleInvestigator();
Employee investigator02 = new PrincipleInvestigator();
```

Both principle investigators have the same basic role of "Analyze laboratory data." (See Listing 2.) The first principle investigator, **`investigator01`**, is also a Safety Captain. To add that responsibility, can use the existing **`investigator01`** object, and create a new **`SafetyCaptain`** object:

```
investigator01 = new SafetyCaptain(investigator01);
```

With this code, the **`SafetyCaptain`** object wraps itself around the **`PrincipleInvestigator`** object to dynamically change the principle investigator's behavior. Notice how we pass **`investigator01`** into the **`SafetyCaptain`** class's constructor. When we do this, the **`SafetyCaptain`** (Listing 8) gets its own instance of the **`Employee`** object, and uses that instance to call the original **`PrincipleInvestigator`** 's methods. Along with the original **`PrincipleInvestigator`** 's methods, the new **`SafetyCaptain`** decorator adds its own functionality.

In Listing 8, the **`getResponsibility()`** method implemented in the **`SafetyCaptain`** class calls the **`getResponsibility()`** method that's implemented in its **`Employee`** instance, and then adds the **`"\n\tand perform quarterly safety inspections"`** text.

What about the second principle investigator? Like **`investigator01`** this investigator is a safety captain. But **`investigator02`** has also decided to be a blood drive canvasser. The following code adds this employee's two responsibilities:

```
investigator02 = new SafetyCaptain(investigator02);
investigator02 = new BloodDriveCanvasser(investigator02);
```

For **investigator02**, a **BloodDriveCanvasser** wraps around a **SafetyCaptain** which in turn wraps around a **PrincipleInvestigator**. In Listing 9 we also create instances of **ResearchTechnician** and **AdministrativeAssistant**. Figure 5 is the console output of the code in Listings 1 to 9.

```
public class Laboratory
    {
    public static void main(String[] args)
        {
        System.out.println();
        System.out.println("-- Research Laboratory --");
        System.out.println();

        Employee investigator01 = new PrincipleInvestigator();
        investigator01 = new SafetyCaptain(investigator01);
        System.out.println(investigator01.getTitle() +
            "\n\tResponsibilities include " + investigator01.getResponsibility());

        Employee investigator02 = new PrincipleInvestigator();
        investigator02 = new SafetyCaptain(investigator02);
        investigator02 = new BloodDriveCanvasser(investigator02);
        System.out.println(investigator02.getTitle() +
            "\n\tResponsibilities include " + investigator02.getResponsibility());

        Employee technician = new ResearchTechnician();
        technician = new ComputerSecurityOfficer(technician);
        technician = new BloodDriveCanvasser(technician);
        System.out.println(technician.getTitle() +
            "\n\tResponsibilities include " + technician.getResponsibility());

        Employee admin = new AdministrativeAssistant();
        admin = new BloodDriveCanvasser(admin);
        System.out.println(admin.getTitle() +
            "\n\tResponsibility include " + admin.getResponsibility());
        }
    }
```

Listing 9: The Laboratory class – our client application

```
-- Research Laboratory --

Principle Investigator
    Safety Captain
    Responsibilities include Analyze laboratory data
    and perform quarterly safety inspections
Principle Investigator
    Safety Captain
    Blood Drive Coorinator
    Responsibilities include Analyze laboratory data
    and perform quarterly safety inspections
    and coordinate all blood drive activities
Research Technician
    Computer Security Officer
    Blood Drive Coorinator
    Responsibilities include Acquire laboratory data
    and handle computer security issues
    and coordinate all blood drive activities
```

```
Administrative Assistant
    Blood Drive Coorinator
    Responsibility include Provide secretarial support
    and coordinate all blood drive activities
```

Figure 5: The output of the laboratory employee application

None of this stuff works unless we follow a very important design principle (quoted from the GoF book):

"Program to an interface, not an implementation."

Following this design principle keeps you from being tied to a specific implementation, and allows you to dynamically change behavior at runtime. If you program to an implementation you have:

```
PrincipleInvestigator investigator01 = new PrincipleInvestigator();
```

This code commits you to the **`PrincipleInvestigator`** class. In that case, a statement like

```
investigator01 = new SafetyCaptain(investigator01);
```

in Listing 9 is illegal. You can't dynamically change the behavior of **`investigator01`**. Instead of programming to the implementation, we program to the interface:

```
Employee investigator01 = new PrincipleInvestigator();
```

This allows us to dynamically modify behavior (in this case, add additional responsibilities for an employee) at runtime.

Wrap It Up (I'll Take It)

With apologies to *The Fabulous Thunderbirds*, the Decorator design pattern "wraps it up" by allowing a **`Decorator`** class to wrap itself around an existing object. In the lab employee example, the decorators are **`ComputerSecurityOfficer`**, **`BloodDriveCanvasser`**, and **`SafetyCaptain`**. They wrap themselves around the concrete **`Employee`** objects (**`PrincipleInvestigator`**, **`ResearchTechnician`**, and **`AdministrativeAssistant`**) to dynamically add additional roles and responsibilities. These wrappers don't modify any of the original classes, so everything can take place at runtime. This satisfies another important design principle (quoted from Bertrand Meyer's *Object-Oriented Software Construction*):

"Classes should be open for extension, but closed for modification."

In other words, it's alright to add new behavior dynamically, but it is **not** alright to modify an existing component class.

Typical Use of Decorator Design Pattern

The Decorator design pattern is typically used in GUI applications. You decorate simple buttons, dialog boxes, and other GUI components with additional borders or with other GUI components. But, as we show in this article, you can use this design pattern in other ways.

Use of Decorators in the Java API

The Java API makes use of the Decorator design pattern with the **`Reader`** and **`Writer`** classes. For example, in the code

```
FileReader file = new FileReader("barry.txt");
BufferedReader buffered = new BufferedReader(reader);
```

you're decorating a **`FileReader`** with the **`BufferedReader`** class.

Some Consequences

Use of the Decorator design pattern can have both good and bad consequences.

Making Your Code More Flexible

Decorating is more flexible than plain, old static inheritance. This is, of course, a major benefit. Decorators avoids all of the nasty, convoluted objects with potentially long class names – names like **`AdminstrativeAssistantAndBloodDriveCanvasser`**. This pattern is especially handy because you can add any number of decorators to a particular component.

Keeping It Simple

When you use the Decorator design pattern, you don't have to write complex classes that consider every possible scenario (e.g., every combination of employee roles). You design a simple, abstract class (such as **`Employee`**) and then decorate the class as necessary. You can also develop new concrete decorators without disturbing the original design of the application.

The Identity Crisis

Decorators wrap themselves around components. But an original component isn't the same as a decorated component. You need to be careful about object identity whenever you refer to one of the decorated components.

For example, when you decorate a **`PrincipleInvestigator`** object with the **`ComputerSecurityOfficer`** type, you dynamically change the object's behavior and identity. If some client code expects that object to behave strictly as a **`PrincipleInvestigator`**, then the client may be in for a big (painful) surprise. This danger is inherent in any kind of dynamic type manipulation, with or without the Decorator design pattern.

Object Critters

Even with the Decorator pattern, you can still have an overabundance of classes, especially the concrete decorator classes. But, each of these classes is short and sweet. Each concrete decorator class describes only one thing. It describes the way in which the class decorates a component object.

Conclusion

In the object-oriented programming paradigm, we model real-life objects. The Decorator is a very versatile pattern for the modeling of real-life objects.

Resources

Design Patterns – Elements of Reusable Object-Oriented Software
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
ISBN 0-201-63361-2

Head First Design Patterns
Eric & Elisabeth Freeman
ISBN 0-596-00712-4

Object-Oriented Software Construction
Bertrand Meyer
ISBN 0-13-629155-4

Data & Object Factory
http://www.dofactory.com/Patterns/Patterns.aspx

About the Authors

Barry Burd is a professor in the Department of Mathematics and Computer Science at Drew University in Madison, New Jersey. When he's not lecturing at Drew University, Dr. Burd leads training courses for professional programmers in business and industry. He has lectured at conferences in America, Europe, Australia and Asia. He is the author of several articles and books, including “Java 2 For Dummies” and “Eclipse For Dummies,” both published by Wiley.

Michael P. Redlich is a Senior Research Technician (formerly a Systems Analyst) at ExxonMobil Research & Engineering, Co. in Clinton, New Jersey with extensive experience in developing custom web and scientific laboratory applications. Mike also has experience as a Technical Support Engineer for Ai-Logix, Inc. where he developed computer telephony applications. He has a Bachelor of Science in Computer Science from Rutgers University. Mike's computing experience includes computer security, relational database design and development, object-oriented design and analysis, C/C++, Java, Visual Basic, FORTRAN, Pascal, MATLAB, HTML, XML, ASP, VBScript, and JavaScript in both the PC and UNIX environments.