Design Patterns

The Decorator Design Pattern

Decorator

Design Pattern

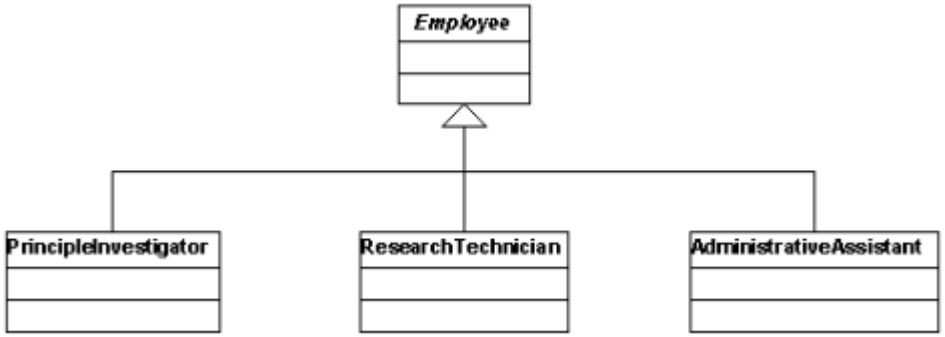
The Problem

You want to add behavior or state to individual objects of the same type at run-time. Inheritance is not feasible because it is static and applies to an entire class.

The Decorator pattern provides a flexible alterative to sub-classing for extending functionality.

Excessive sub-classing 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. The following shows a potential UML class diagram for this application:



Here's some source code to go along with the model:

```
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, you should include these additional roles in your model.

Decorator
Design Pattern

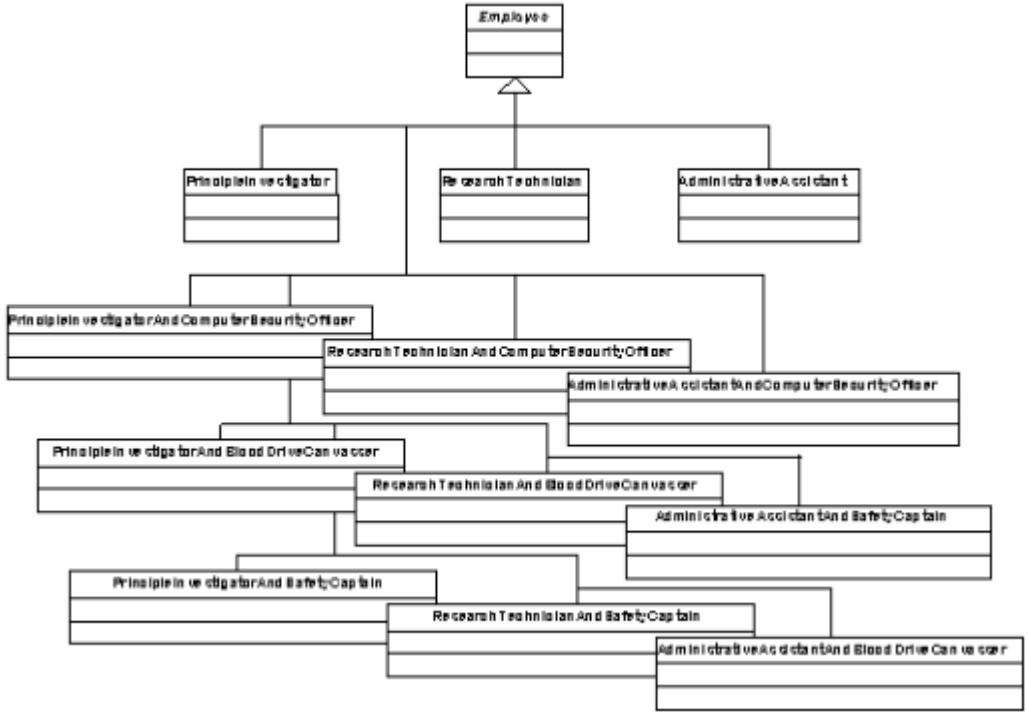
You can add more objects by continuing to extend the Employee class:

```
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
{
}
```

Decorator

Design Pattern

By continuing to extend the abstract Employee base class, the updated UML diagram begins to look like this:



Using this approach, you'll have to account for every combination of employee and employee roles.

The number of combinations becomes unmanageable.

Decorator

Design Pattern

You may be tempted to solve the problem by implementing interfaces, 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.

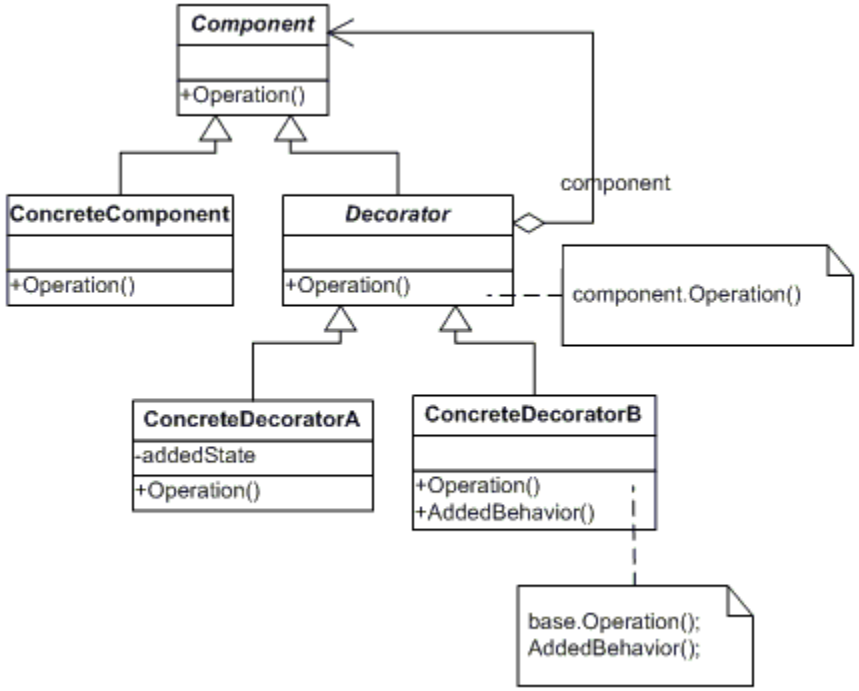
You can try adding fields to the Employee class:

```
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.

The Decorator Pattern

The following shows a UML diagram for the Decorator pattern:



- The Component class is any abstract class you want to decorate. Even before being decorated, this class has some defined behavior. *In the **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, the **PrincipleInvestigator** class may have code to keep track of the investigator's research projects.*

Decorator

Design Pattern

- The Decorator class is an abstract class, with 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 Decorators 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 the 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).

Decorator

Design Pattern

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.

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

Decorator

Design Pattern

Assume the following is the source code behind the UML diagram:

The Employee (Component) Class:

```
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();
}
```

The PrincipleInvestigator (ConcreteComponent) Class:

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

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

The ResearchTechnician (ConcreteComponent) Class:

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

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

The AdministrativeAssistant (ConcreteComponent) Class:

```
public class AdministrativeAssistant extends Employee
{
    public AdministrativeAssistant()
        {
        super.setTitle("Administrative Assistant");
        }

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

Decorator

Design Pattern

The EmployeeDecorator (Decorator) class:

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

The ComputerSecurityOfficer (ConcreteDecorator) 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";
        }
}
```

The BloodDriveCanvasser (ConcreteDecorator) 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";
        }
}
```

Decorator

Design Pattern

The SafetyCaptain (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";
        }
}
```

Decorator

Design Pattern

The following code shows the client application (“main”) putting making use of these classes.

```
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\tResponsibilities include " + admin.getResponsibility());
        }
    }
```

Decorator

Design Pattern

It starts 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".

The first principle investigator, investigator01, is also a Safety Captain. To add that responsibility, you 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 it passes investigator01 into the SafetyCaptain class's constructor. Doing this, the SafetyCaptain 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.

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.

Decorator
Design Pattern

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.

We also create instances of ResearchTechnician and AdministrativeAssistant.

The following shows the output generated by the client code (“main”):

```
-- 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
     Responsibilities include Provide secretarial support
     and coordinate all blood drive activities
```

However, none of this works unless you follow a very important design principle:

"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 this one is illegal:

investigator01 = new SafetyCaptain(investigator01);

You can't dynamically change the behavior of investigator01. If, instead, you program to the interface (or in this case, an abstract parent class type):

Employee investigator01 = new PrincipleInvestigator();

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

Wrap It Up!

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 the Open-Closed design principle:

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

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

The Decorator design pattern is typically used in GUI applications. It's used to decorate simple buttons, dialog boxes, and other GUI components with additional borders or with other GUI components. But, as you've seen, you can use this design pattern in other ways.

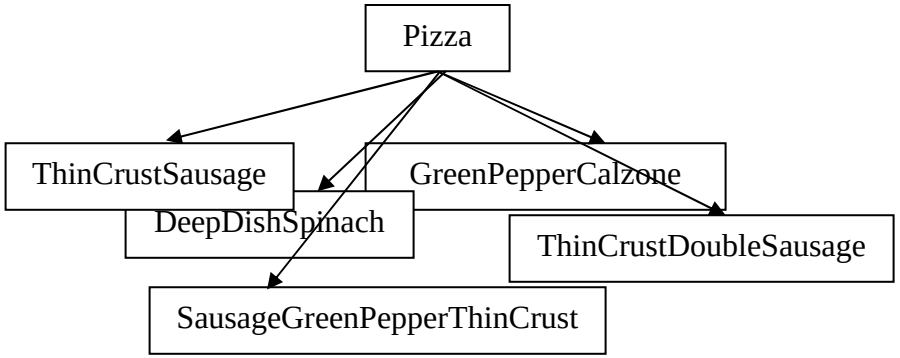
Another Example

For the second example, assume we are writing an application that allows the specification of a pizza!
The pizzas can have different sizes, crust types, toppings, etc. The behaviors of describing and pricing pizzas varies depending upon the particular combination that is selected.

Ultimately, we'd like the application to create any combination of pizzas, and be able to describe them and price them "generically" - allowing each pizza to determine its own description and price.

This is another situation where inheritance would not work out so well. While we could create a parent class "Pizza", the combinations of crust, size, and toppings isn't hierarchical.

We'd end up with a large proliferation of child classes:



Decorator

Design Pattern

Similar to the previous example, we'll create a generic "Component" class - Pizza

The Pizza (Component) Class:

```
public abstract class Pizza {

    public enum Size {

        UNDEFINED, SMALL, MEDIUM, LARGE, XLARGE
    }
    private String description = "Unknown Pizza";
    private Size size = Size.UNDEFINED;

    public Pizza(Size s)
    {
        size = s;
    }
    public abstract double cost();

    public String getDescription() {
        if (getSize() == Size.SMALL) {
            return "Small" + ", " + description;
        } else if (getSize() == Size.MEDIUM) {
            return "Medium" + ", " + description;
        } else if (getSize() == Size.LARGE) {
            return "Large" + ", " + description;
        } else if (getSize() == Size.XLARGE) {
            return "Extra Large" + ", " + description;
        }
        return description;
    }

    public Size getSize() {
        return size;
    }

    public void setDescription(String description) {
        this.description = description;
    }
}
```

Decorator

Design Pattern

The pizza “crust types” will be represented as the “ConcreteComponent” classes – the children of “Pizza”:

The DeepDish (ConcreteComponent) Class:

```
public class DeepDish extends Pizza {

    public DeepDish(Size s) {
        super(s);
        setDescription("Deep Dish");
    }

    public double cost() {
        return 14.99;
    }
}
```

The ThinCrust (ConcreteComponent) Class:

```
public class ThinCrust extends Pizza {

    public ThinCrust(Size s) {
        super(s);
        setDescription("Thin Crust");
    }

    public double cost() {
        return 10.99;
    }
}
```

The Calzone (ConcreteComponent) Class:

```
public class Calzone extends Pizza {

    public Calzone(Size s) {
        super(s);
        setDescription("Calzone");
    }

    public double cost() {
        return 12.99;
    }
}
```

Decorator
Design Pattern

We will create classes to represent the individual toppings available. Each will have a parent class of “ToppingDecorator” – our abstract “decorator” class:

The ToppingDecorator (Decoratator) class:

```
public abstract class ToppingDecorator extends Pizza {

    public ToppingDecorator(Size s) {
        super(s);
    }
    public abstract String getDescription();
}
```

The classes representing the individual toppings will be child classes of ToppingDecorator:

The GreenPepper (ConcreteDecorator) Class:

```
public class GreenPepper extends ToppingDecorator {

    private Pizza pizza;

    public GreenPepper(Pizza pizza) {
        super(pizza.getSize());
        this.pizza = pizza;
    }

    public String getDescription() {
        return pizza.getDescription() + ", Green Pepper";
    }

    public double cost() {
        double cost = pizza.cost();
        if (getSize() == Size.SMALL) {
            cost += 0.80;
        } else if (getSize() == Size.MEDIUM) {
            cost += 1.00;
        } else if (getSize() == Size.LARGE) {
            cost += 1.20;
        } else if (getSize() == Size.XLARGE) {
            cost += 1.40;
        }
        return cost;
    }
}
```

Decorator
Design Pattern

The Sausage (ConcreteDecorator) Class:

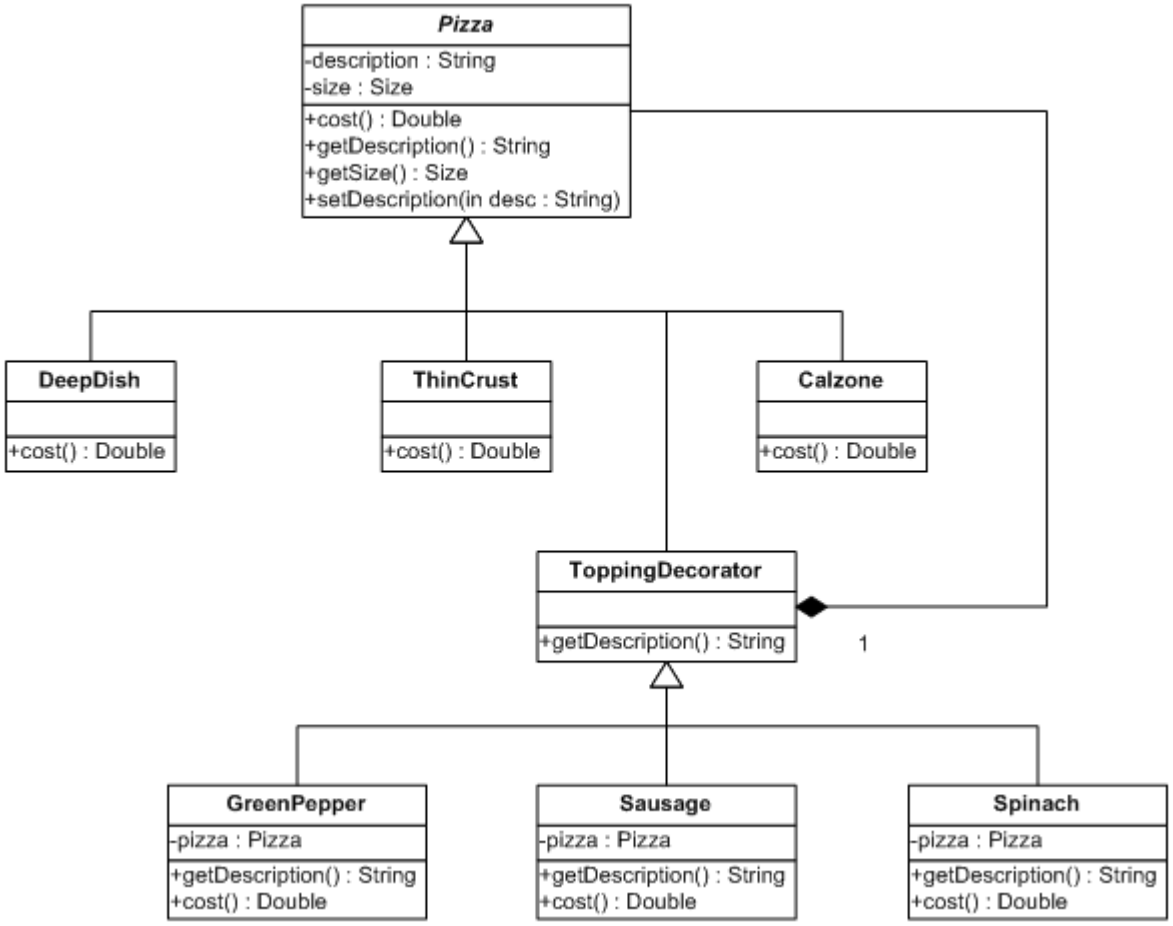
```
public class Sausage extends ToppingDecorator {

    private Pizza pizza;

    public Sausage(Pizza pizza) {
        super(pizza.getSize());
        this.pizza = pizza;
    }

    public String getDescription() {
        return pizza.getDescription() + ", Sausage";
    }

    public double cost() {
        double cost = pizza.cost();
        if (getSize() == Size.SMALL) {
            cost += 1.20;
        } else if (getSize() == Size.MEDIUM) {
            cost += 1.40;
        } else if (getSize() == Size.LARGE) {
            cost += 2.00;
        } else if (getSize() == Size.XLARGE) {
            cost += 2.80;
        }
        return cost;
    }
}
```

The Spinach (ConcreteDecorator) Class:

```
public class Spinach extends ToppingDecorator {

    private Pizza pizza;

    public Spinach(Pizza pizza) {
        super(pizza.getSize());
        this.pizza = pizza;
    }

    public String getDescription() {
        return pizza.getDescription() + ", Spinach";
    }

    public double cost() {
        double cost = pizza.cost();
        if (getSize() == Size.SMALL) {
            cost += 0.90;
        } else if (getSize() == Size.MEDIUM) {
            cost += 1.10;
        } else if (getSize() == Size.LARGE) {
            cost += 1.35;
        } else if (getSize() == Size.XLARGE) {
            cost += 1.80;
        }
        return cost;
    }
}
```

Decorator
Design Pattern

The UML diagram for the Pizza related classes follows the Decorator design pattern as expected.

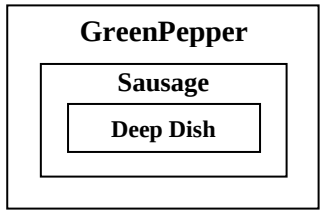


Decorator
Design Pattern

But – How do we use it?

Let's assume we want to make a Large, Deep Dish, Sausage & Green Pepper Pizza. To do this, we would do the following:

```
public static void main(String args[]) {

    // Lets make a Large Deep Dish Sausage & Green Pepper Pizza!

1   Pizza p1 = new DeepDish(Size.LARGE);
    // Now we have a plain large deep dish pizza...

2   p1 = new Sausage(p1);
    // Add Sausage by wrapping the plain large deep dish pizza with "Sausage" topping ...

3   p1 = new GreenPepper(p1);
    // Add GreenPepper by wrapping the large deep dish sausage pizza with "GreenPepper" topping...

4   System.out.println(p1.getDescription() + " $" + p1.cost());

}
```



So, here's what we did:

1: Create the initial Pizza object – a plain (no topping) Large Deep Dish (the Component)

2: Wrap the initial Pizza object with a "Sausage" topping object (a ConcreteDecorator)

3: Wrap "decorated" Pizza object with another decorator - a "GreenPepper" topping object (another ConcreteDecorator)

4: Invoke the "getDescription" and "cost" behaviors on the resulting Pizza

In this way, we can create any combination of pizza size, crust type, and topping – on the fly:

- Small Spinach Calzone
- Medium Double Sausage Thin Crust
- Extra-Large Deep Dish Sausage, Green Pepper, and Spinach
- Etc…

When the "getDescription" method is invoked on the Pizza object the following occurs:

- p1.getDescription() ➔ Since "p1" is actually referring to a GreenPepper object, the "getDescription" method of the GreenPepper class is executed.
- The "getDescription" method of the GreenPepper class looks like this:

```
public String getDescription() {
    return pizza.getDescription() + ", Green Pepper";
}
```

So first, we follow the "pizza" reference and call that object's "getDescription" method. The "pizza" reference in this case refers to a Sausage object (GreenPepper is wrapping the Sausage topping object).

- The "getDescription" method of the Sausage class looks like this:

```
public String getDescription() {
    return pizza.getDescription() + ", Sausage";
}
```

Decorator
Design Pattern

So first, we follow the "pizza" reference and call that object's "getDescription" method. The "pizza" reference in this case refers to the DeepDish object. Since DeepDish does not over-ride "getDescription()" we instead execute the "getDescription()" method of the parent class – the initial Pizza object (Sausage is wrapping the initial Pizza object).

- The "getDescription" method of the Pizza class looks like this:

```
public String getDescription() {
    if (getSize() == Size.SMALL) {
        return "Small" + ", " + description;
    } else if (getSize() == Size.MEDIUM) {
        return "Medium" + ", " + description;
    } else if (getSize() == Size.LARGE) {
        return "Large" + ", " + description;
    } else if (getSize() == Size.XLARGE) {
        return "Extra Large" + ", " + description;
    }
    return description;
}
```

Since the initial Pizza object was created as a LARGE pizza, this method returns the text "Large" plus the value of the "description" field of the Pizza object – "Deep Dish" (that was set by the Concrete Component object DeepDish). So the returned String is "Large, Deep Dish".

- That is returned back to the Sausage object's "getDescription" method (which we saw earlier). That method takes the String returned from the Pizza object and adds ", Sausage" to it. So the returned String is "Large, Deep Dish, Sausage".
- That is returned back to the GreenPepper object's "getDescription" method (which we saw earlier). That method takes the String returned from the Sausage object and adds ", Green Pepper" to it. So the returned String is "Large, Deep Dish, Sausage, Green Pepper".
- So the description for this pizza object is displayed as: `Large, Deep Dish, Sausage, Green Pepper`

Decorator

Design Pattern

When the “cost” method is invoked on the Pizza object the following occurs:

- p1.cost() ➔ Since “p1” is actually referring to a GreenPepper object, the “cost” method of the GreenPepper class is executed.

- The “cost” method of the GreenPepper class looks like this:

```
public double cost() {
    double cost = pizza.cost();
    if (getSize() == Size.SMALL) {
        cost += 0.80;
    } else if (getSize() == Size.MEDIUM) {
        cost += 1.00;
    } else if (getSize() == Size.LARGE) {
        cost += 1.20;
    } else if (getSize() == Size.XLARGE) {
        cost += 1.40;
    }
    return cost;
}
```

So first, we follow the “pizza” reference and call that object’s “cost” method. The “pizza” reference in this case refers to a Sausage object (GreenPepper is wrapping the Sausage topping object).

Decorator
Design Pattern

- The “cost” method of the Sausage class looks like this:

```
public double cost() {
    double cost = pizza.cost();
    if (getSize() == Size.SMALL) {
        cost += 1.20;
    } else if (getSize() == Size.MEDIUM) {
        cost += 1.40;
    } else if (getSize() == Size.LARGE) {
        cost += 2.00;
    } else if (getSize() == Size.XLARGE) {
        cost += 2.80;
    }
    return cost;
}
```

So first, we follow the “pizza” reference and call that object’s “cost” method. The “pizza” reference in this case refers to the parent class DeepDish (Sausage is wrapping the initial DeepDish Pizza object).

- The “cost” method of the DeepDish class looks like this *(the base price for a plain Deep Dish pizza is $14.99)* :

```
public double cost() {
    return 14.99;
}
```

So, the cost value of 14.99 is returned to the caller (Sausage).

Decorator
Design Pattern

- $14.99 is returned back to the Sausage object's "cost()" method (which we saw earlier). The Sausage "cost()" method adds $2.00 to the Deep Dish cost *(you add $2.00 to the cost to add Sausage to a Large pizza*).The resulting value returned back to the caller (GreenPepper) is $16.99

- $16.99 is returned back to the GreenPepper object's "cost()" method (which we saw earlier). The GreenPepper "cost()" method adds $1.20 to the Deep Dish cost *(you add $1.20 to the cost to add Green Peppers to a Large pizza*).The resulting value returned back to the caller ("main") is $18.19

- So the cost for this pizza object is displayed as: `$18.19`

Here again, many additional crust types, sizes, and toppings can be added over time and this design will support the additions without requiring changes to the existing classes.

Decorator

Design Pattern

To further support abstraction, we could make a static PizzaFactory class so users of Pizza objects would not need to know the details of Pizza creation – they would not even be aware the Decorator pattern existed:

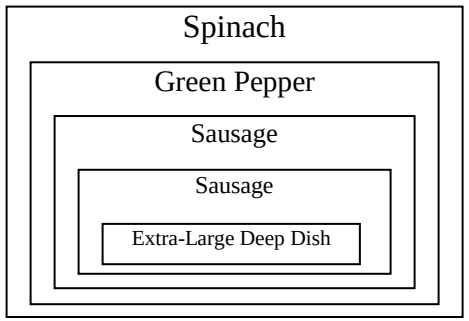
```
public class PizzaFactory {

    public static Pizza buildPizza(Size sz, String type, String... details) throws Exception {
        Pizza p = makePizza(sz, type);
        for (String s : details) {
            p = makeTopping(s, p);
        }

        return p;
    }
    private static Pizza makeTopping(String type, Pizza p) throws Exception {
        if (type.toLowerCase().trim().equals("green pepper")) {
            return new GreenPepper(p);
        } else if (type.toLowerCase().trim().equals("sausage")) {
            return new Sausage(p);
        } else if (type.toLowerCase().trim().equals("spinach")) {
            return new Spinach(p);
        } else {
            throw new Exception("Unknown topping: " + type);
        }
    }
    private static Pizza makePizza(Size sz, String type) throws Exception {
        if (type.toLowerCase().trim().equals("thin crust")) {
            return new ThinCrust(sz);
        } else if (type.toLowerCase().trim().equals("deep dish")) {
            return new DeepDish(sz);
        } else if (type.toLowerCase().trim().equals("calzone")) {
            return new Calzone(sz);
        } else {
            throw new Exception("Unknown type: " + type);
        }
    }
}
```

Using this factory, the "main" could be changed to this:

```
public static void main(String args[]) {

   // If we ask for non-existant option, the factory will throw an exception so we try/catch that here
   try {

      // Lets make a Large Deep Dish Sausage & Green Pepper Pizza!
      Pizza p = PizzaFactory.buildPizza(Size.XLARGE, "Deep Dish", "Sausage", "Sausage", "Green Pepper", "Spinach");
      System.out.println(p.getDescription() + " $" + p.cost());

   }  catch (Exception ex) {
         Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
   }
}
```

The *"buildPizza"* method (seen earlier) will first call the private *"makePizza"* method which will create the appropriate ConcreteComponent (DeepDish, ThinCrust, etc) with the specified size. In this case we've passed **Size.XLARGE and "Deep Dish"** to the factory so we get an Extra-Large Deep Dish pizza.

Next, for each topping passed in, *"makePizza"* calls the private *"makeTopping"* method. That method creates the appropriate ConcreteDecorator (Sausage, Spinach, etc) and uses it to wrap the previously created pizza object. In this case we've passed "Sausage", "Sausage", "Green Pepper", and "Spinach" (i.e., a double-sausage, green pepper & spinach). So we end up with a Spinach object wrapping a GreenPepper object, which is wrapping a Sausage object, which is wrapping another Sausage object, which is wrapping the initial Extra-Large Deep Dish pizza object:



Decorator
Design Pattern

In this case, the "getDescription()" method generates: `Extra Large, Deep Dish, Sausage, Sausage, Green Pepper, Spinach`

Spinach: pizza.getDescription() + **", Spinach"**
↓
GreenPepper: pizza.getDescription() + **", Green Pepper"**
↓
Sausage: pizza.getDescription() + **", Sausage"**
↓
Sausage: pizza.getDescription() + **", Sausage"**
↓
Pizza: **"Extra Large, Deep Dish"**

In this case, the "cost()" method generates: `$23.79`

Spinach: pizza.cost() + **$1.80**
↓
GreenPepper: pizza.cost () + **$1.40**
↓
Sausage: pizza.cost () + **$2.80**
↓
Sausage: pizza.cost () + **$2.80**
↓
Pizza: **$14.99**

The Pros and the Cons

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. Decorator patterns help you avoid all of those 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.

Decorator

Design Pattern

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

Decorator Summary

- Inheritance is one form of extension, but not necessarily the best way to achieve flexibility in our designs.
- In our designs we should allow behavior to be extended without the need to modify existing code (Open-Closed Principle)
- Composition and Delegation can often be used to add new behaviors at runtime.
- The Decorator Pattern provides an alternative to sub-classing for extending behavior.
- The Decorator Pattern involves a set of decorator classes that are used to wrap concrete components.
- Decorator classes mirror the type of the components they decorate. (In fact, they are the same type as the components they decorate, either through inheritance or interface implementation.)
- Decorators change the behavior of their components by adding new functionality before and/or after (or even in place of) method calls to the component.
- You can wrap a component with any number of decorators.
- Decorators are typically transparent to the client of the component; that is, unless the client is relying on the component's concrete type.
- Decorators can result in many small objects in our design, and overuse can be complex.