FDA149
Software Engineering

Design Patterns
Examples

Peter Bunus
Dept of Computer and Information Science
Linköping University, Sweden
petbu@ida.liu.se
Decorator Pattern
Joe, people are not coming to our pizza places in the morning. They need coffee in the morning. I decided to open a coffee shop next to each pizzeria. Could you please implement an application for ordering coffee?
The First Design of the Coffee Shop

No problem boss. I can fix this. I have now experience with the pizza store so this will be a piece of cake

Beverage
- description
+ getDescription()
+ cost()

HouseBlend
+ cost()

DarkRoast
+ cost()

Decaf
+ cost()

Expresso
+ cost()

Each subclass implements \texttt{cost()} the cost of the beverage

The \texttt{cost()} method is abstract; subclasses need to define their implementation.
What I'm doing wrong here?
Which of the design principles we are violating here?
The Constitution of Software Architects

- Encapsulate that vary.
- Program to an interface not to an implementation.
- Favor Composition over Inheritance.
Why do we need so many classes???

We add instance variables to represent whether or not each beverage has milk, soy, mocha and whip...

Now we'll implement cost() in Beverage (instead of keeping it abstract), so that it can calculate the costs associated with the condiments for a particular beverage instance.

Subclasses will still override cost(), but they will also invoke the super version so that they can calculate the total cost of the basic beverage plus the costs of the added condiments.
Excellent Joe, good job. Five classes. This will decrease the complexity of our ordering system.

I’m not so sure about this. My experience with high management is not so good. They change the requirements all the time. And the customers they want new things all the time.
What can happen?

- New condiments will appear and will force us to add new methods and change the cost method each time.

- Price changes for condiments so we need to change the cost method.

- New beverages like iced tea. The iced tea class will still inherit the methods like hasWhip().

- How about double espresso.
Inheritance doesn’t worked very well for us. What we should do?

Hi Jamie. One of my guys have problem with coffee classes. Could you please help him out.

1. Take the DarkRoast object
2. Decorate it with a Mocha object
3. Decorate it with the Whip object
4. Call the cost() method and relay on delegation to add to the condiment cost.
Jamie’s recipe

1. Take the DarkRoast object
2. Decorate it with a Mocha object
3. Decorate it with the Whip object
4. Call the cost() method

```cpp
class DarkRoast : public Beverage{
    public:
        DarkRoast();
        double cost();
};

class Mocha : public CondimentDecorator{
    Beverage *beverage;
    public:
        Mocha(Beverage *p_beverage);
        string getDescription();
        double cost();
};

class Whip : public CondimentDecorator{
    Beverage *beverage;
    public:
        Whip(Beverage *p_beverage);
        string getDescription();
        double cost();
};
```
Beverage acts like an abstract component class

```cpp
class Beverage{
    public:
        string description;
        Beverage();
        virtual string getDescription();
        virtual double cost() = 0;
};
```

```cpp
class CondimentDecorator : public Beverage{
    public:
        CondimentDecorator();
        virtual string getDescription() = 0;
};
```

```cpp
class Mocha : public CondimentDecorator{
    Beverage *beverage;
    public:
        Mocha(Beverage *p_beverage){
            beverage = p_beverage;
        }
        string getDescription(){
            return beverage->getDescription() + " Whip";
        }
        double cost(){
            return beverage->cost() + 0.76;
        }
};
```

```cpp
class DarkRoast : public Beverage{
    public:
        DarkRoast();
        double cost();
};
```
void main(){
  cout << "Testing the Coffe Shop application" << endl;
  Beverage *beverage1 = new Expresso();
  cout << beverage1->getDescription() << endl;
  cout << "Cost: " << beverage1->cost() << endl << endl;

  Beverage *beverage2 = new DarkRoast();
  beverage2 = new Mocha(beverage2);
  beverage2 = new Mocha(beverage2);
  beverage2 = new Whip(beverage2);
  cout << beverage2->getDescription() << endl;
  cout << "Cost: " << beverage2->cost() << endl << endl;

  Beverage *beverage3 = new HouseBlend();
  cout << beverage3->getDescription() << endl;
  cout << "Cost: " << beverage3->cost() << endl << endl;
}

A Whipped Dark Roast with double Mocha
void main(){
    cout << "Testing the Coffe Shop application" << endl;
    Beverage *beverage1 = new Expresso();
    cout << beverage1->getDescription() << endl;
    cout << "Cost: " << beverage1->cost() << endl << endl;

    Beverage *beverage2 = new DarkRoast();
    beverage2 = new Mocha(beverage2);
    beverage2 = new Mocha(beverage2);
    beverage2 = new Whip(beverage2);
    cout << beverage2->getDescription() << endl;
    cout << "Cost: " << beverage2->cost() << endl << endl;

    Beverage *beverage3 = new HouseBlend();
    cout << beverage3->getDescription() << endl;
    cout << "Cost: " << beverage3->cost() << endl << endl;
}

Testing the Coffe Shop application
Expresso
Cost: 1.99

Dark Roast Mocha Mocha Whip
Cost: 3.55

House Blend
Cost: 0.89

Press any key to continue . . .
How is the Cost Computed?

.....
Beverage *beverage2 = new DarkRoast();
beverage2 = new Mocha(beverage2);
beverage2 = new Mocha(beverage2);
beverage2 = new Whip(beverage2);
cout << beverage2->getDescription() << endl;
cout << "Cost: " << beverage2->cost() << endl;
.....

double Whip::cost(){
    return beverage->cost() + 0.76;
}
double Mocha::cost(){
    return beverage->cost() + 0.9;
}
double DarkRoast::cost(){
    return 0.99;
}
The Decorator Pattern

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
The Constitution of Software Architects

- Encapsulate that vary.
- Program to an interface not to an implementation.
- Favor Composition over Inheritance.
- Classes should be open for extension but closed for modification
Decorating Text

Some applications would benefit from using objects to model every aspect of their functionality, but a naïve design approach would be prohibitively expensive. For example, most documentations modularize text formatting and editing tools to some extent. However, they invariably stop short of using objects to represent each character and graphical elements in the document. Doing so would promote flexibility at the least level in the application. Text and graphics could be treated uniformly with

```
VisualComponent
  Draw()

TextView
  Draw()

Decorator
  Draw()

ScrollDecorator
  Draw()
  ScrollTo()
  scrollPosition

BorderDecorator
  Draw()
  DrawBorder()
  borderWidth
```

Decorate::Draw(); DrawBorder();
Decorator – Non Software Example

Visual Component
  Hang()

Painting
  Hang()

Frame

Decorator
  Hang()

Matte
The Decorator Advantages/Disadvantages

+ + + + +

- Provides a more flexible way to add responsibilities to a class than by using inheritance, since it can add these responsibilities to selected instances of the class

- Allows to customize a class without creating subclasses high in the inheritance hierarchy.

- A Decorator and its enclosed component are not identical. Thus, tests for object types will fail.

- Decorators can lead to a system with “lots of little objects” that all look alike to the programmer trying to maintain the code
What we have learned?

- Inheritance is one form of extension, but not necessarily the best way to achieve flexibility in our design
- In our design we should allow behavior to be extended without the need to modify the existing code
- Composition and delegation can often be used to add new behaviors at runtime
- The Decorator Pattern involves a set of decorator classes that are used to wrap concrete components
- 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
- Decorators can result in many small objects in our design, and overuse can be complex
The Mediator
The Mediator – Non Software Example

The **Mediator** defines an object that controls how a set of objects interact.

- The pilots of the planes approaching or departing the terminal area communicate with the tower, rather than explicitly communicating with one another.
- The constraints on who can take off or land are enforced by the tower.
- the tower does not control the whole flight. It exists only to enforce constraints in the terminal area.
The Mediator – Another Example

- Bob lives in the HouseOfFuture where everything is automated:
  - When Bob hits the snooze button of the alarm the coffee maker starts brewing coffee
  - No coffee in weekends
  - .......

```java
onEvent()
    checkCalendar();
    checkSprinkler();
    startCoffee();
    //do more stuff

onEvent()
    checkDayOfWeek();
    doShower();
    doCoffee();
    doAlarm();
    //do more stuff
```

```java
onEvent()
    checkCalendar();
    checkAlarm();
    //do more stuff
```

```java
onEvent()
    checkCalendar();
    checkTemperature;
    //do more stuff
```
The Mediator in Action

- With a Mediator added to the system all the appliance objects can be greatly simplified
  - They tell the mediator when their state changes
  - They respond to requests from the Mediator

```
if(alarmEvent)()
  checkCalendar();
  checkShower();
  checkTemp();
  //do more stuff
}
if(weekend){
  checkWeather();
}
if(trashDay){
  resetAlarm();
}
```
The Client creates aFontDialog and invokes it.

The list box tells the FontDialog (it's mediator) that it has changed.

The FontDialog (the mediator object) gets the selection from the list box.

The FontDialog (the mediator object) passes the selection to the entry field edit box.
Actors in the Mediator Pattern

**Mediator**
defines an interface for communicating with Colleague objects

**ConcreteMediator**
implements cooperative behavior by coordinating Colleague objects
knows and maintains its colleagues

**Colleague classes (Participant)**
each Colleague class knows its Mediator object (has an instance of the mediator)
each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague
Yet Another Example

for each c in theConcreteClients
c.handleEvent()

Robbery in progress. I need backup

Officer down, officer down!!! We have casualties
Mediator advantages and disadvantages

- Changing the system behavior means just subclassing the mediator. Other objects can be used as is.

- Since the mediator and its colleagues are only tied together by a loose coupling, both the mediator and colleague classes can be varied and reused independent of each other.

- Since the mediator promotes a One-to-Many relationship with its colleagues, the whole system is easier to understand (as opposed to a many-to-many relationship where everyone calls everyone else).

- It helps in getting a better understanding of how the objects in that system interact, since all the object interaction is bundled into just one class - the mediator class.

- Since all the interaction between the colleagues are bundled into the mediator, it has the potential of making the mediator class very complex and monolithically hard to maintain.
When an event occurs, colleagues must communicate that event with the mediator. This is somewhat reminiscent of a subject communicating a change in state with an observer.

One approach to implementing a mediator, therefore, is to implement it as an observer following the observer pattern.
Seven Layers of Architecture

- Enterprise-Architecture
- Global-Architecture
- System-Architecture
- Application-Architecture
- Macro-Architecture
- Micro-Architecture

OO Architecture
Subsystem
Frameworks
Design-Patterns
Objects
OO Programming
Antipatterns Sources

AntiPatterns
Refactoring Software, Architectures, and Projects in Crisis
William H. Brown
Raphael C. Malveau
Hays W. "Skip" McCormick III
Thomas J. Mowbray

AntiPatterns and Patterns in Software Configuration Management
William J. Brown
Hays W. "Skip" McCormick III
Scott W. Thomas
Congratulations: You have now completed TDDB84