FDA149
Software Engineering

Design Patterns
Examples

Peter Bunus
Dept of Computer and Information Science
Linköping University, Sweden
petbu@ida.liu.se

pelab

Linköpings universitet
INSTITUTE OF TECHNOLOGY

pelab

Decorator Pattern

pelab

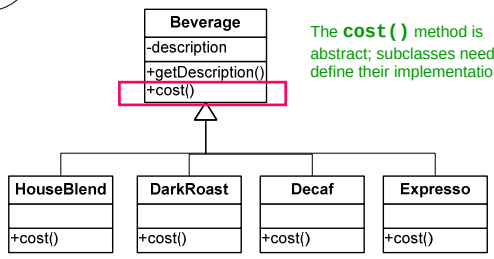
Extending the Business

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?

pelab

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

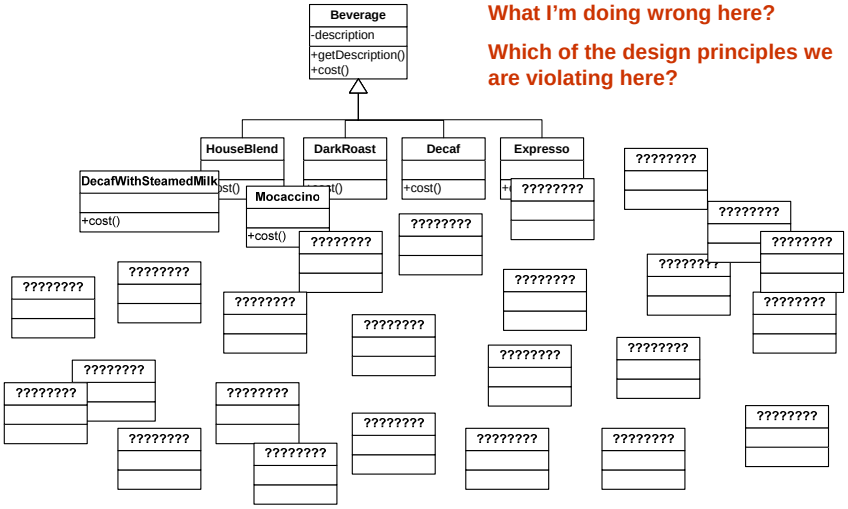


The `cost()` method is abstract; subclasses need to define their implementation

Each subclass implements `cost()` the cost of the beverage

pelab

Class Explosion



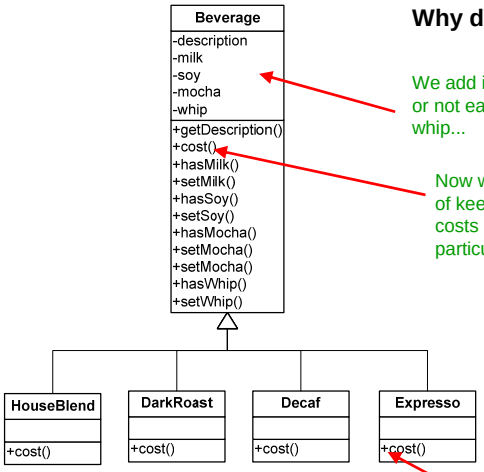
What I'm doing wrong here?

Which of the design principles we are violating here?

pelab

The Constitution of Software Architectcts

- Encapsulate that vary.
- Program to an interface not to an implementation.
- Favor Composition over Inheritance.
- ?????????
- ?????????
- ?????????
- ?????????
- ?????????
- ?????????

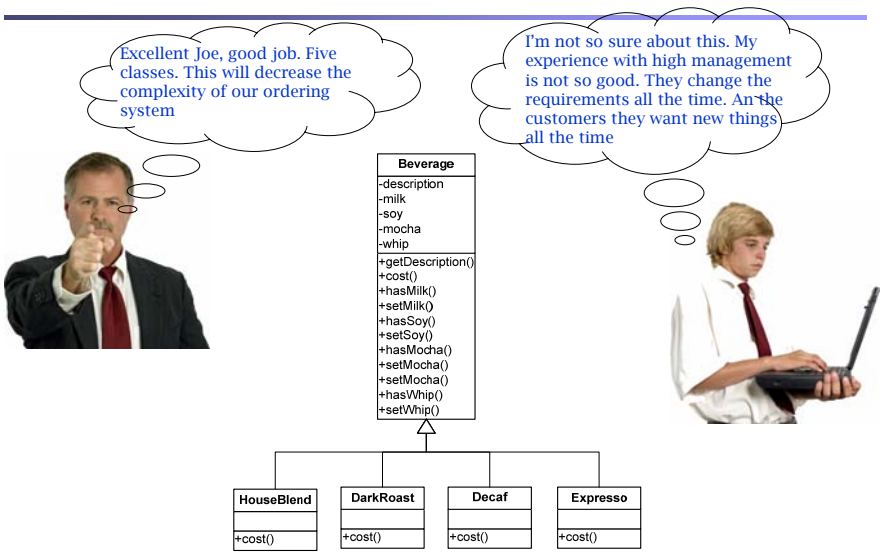


Why do we need so many classes ???

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.



What can happend?

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

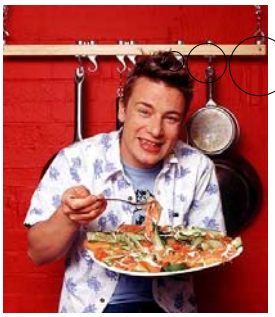
- Price changes for condiments so we need to change the cost method.

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

- How about double espresso.

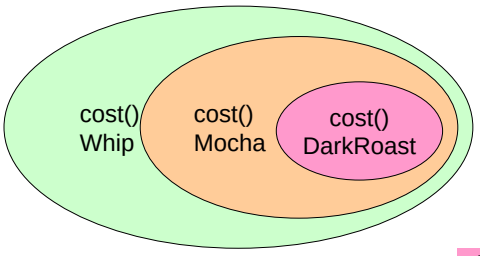
Decorating Coffee

- Inheritance doesn't worked very well for us. What we should do?



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

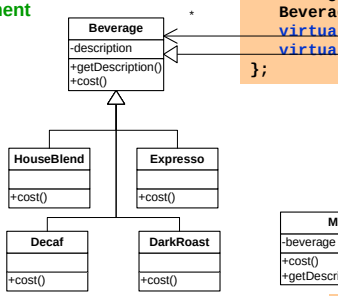
```
class DarkRoast : public Beverage{
public:
  DarkRoast();
  double cost();
};
```

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

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

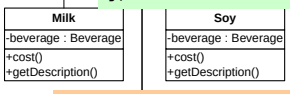

Barista Training for Sofware Engineers

Beverage acts like an abstract component class




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

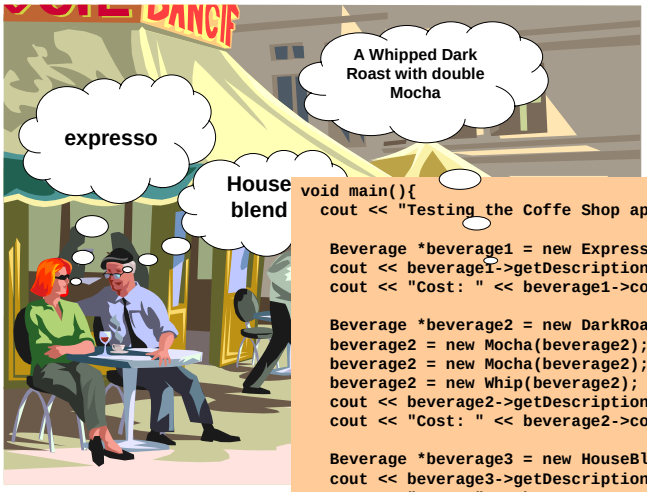
```
class CondimentDecorator : public Beverage{
public:
  CondimentDecorator(){};
  virtual string getDescription()=0;
};
```




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

```
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;
   };
};
```

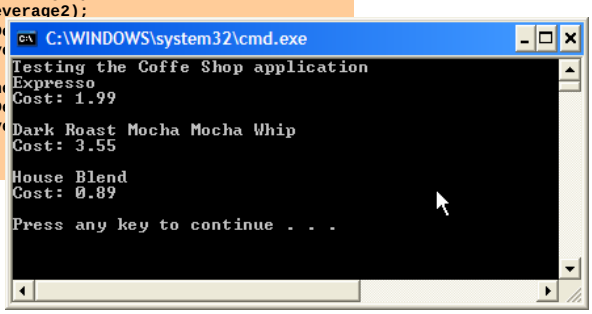
Running the Coffe Shop




```
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;

}
```

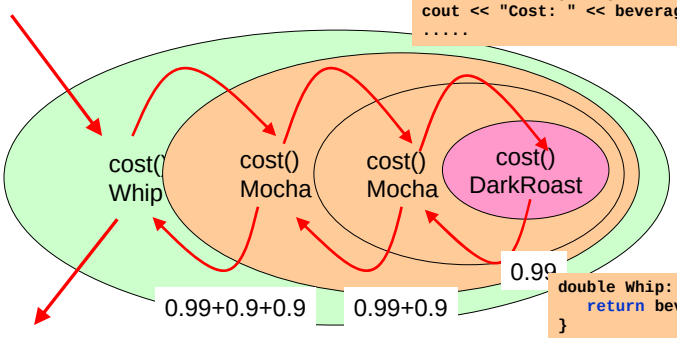

Running the Coffe Shop

```
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->getD
   cout << "Cost: " << bev

   Beverage *beverage3 = n
   cout << beverage3->getD
   cout << "Cost: " << bev

}
```



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;
.....
```



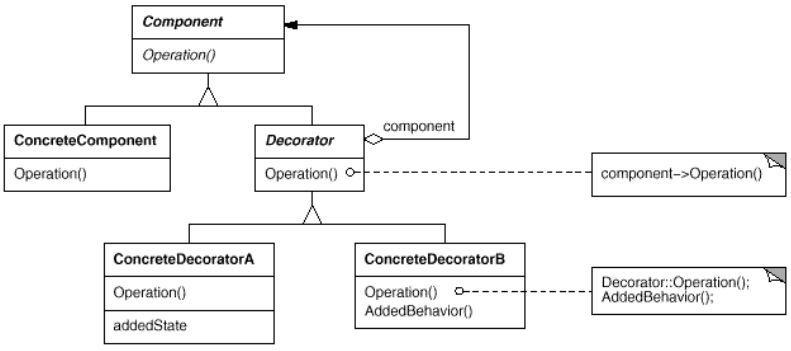
0.99+0.9+0.9 + 0.76 = 3.55

```
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

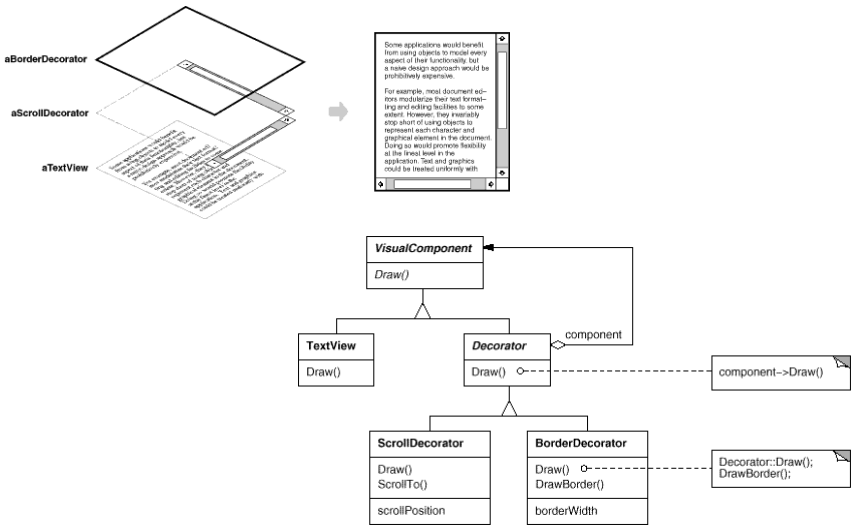


Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality

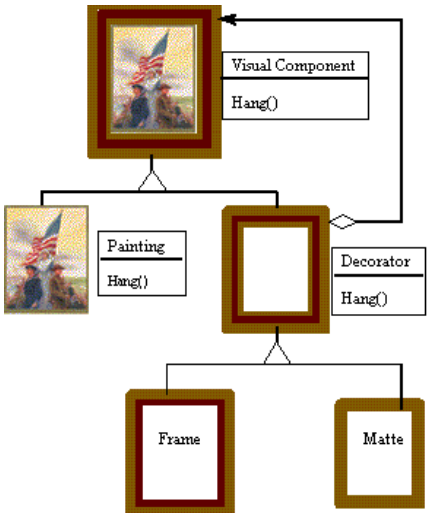
The Constitution of Software Architectcts

- 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



Decorator – Non Software Example



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.

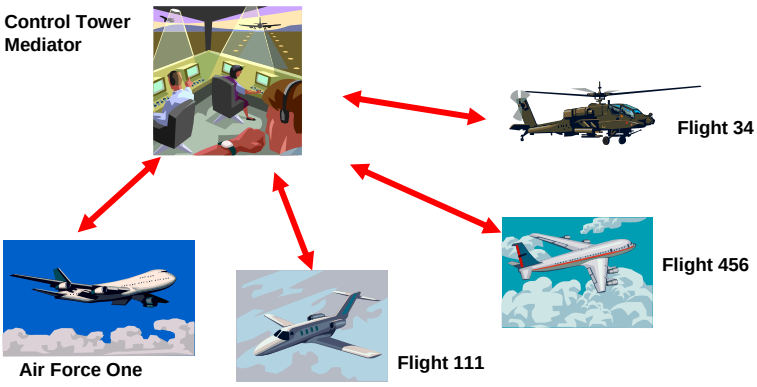
- A Decorator and its enclosed component are not identical. Thus, tests for object types will fail.
- Decorators can lead to a system with “lots of little objects” that all look alike to the programmer trying to maintain the code

What we have learned?

- Inheritance is one form of extension, but not necessarily he best way to achieve flexibility in our design
- In our design we should allow behavior to 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

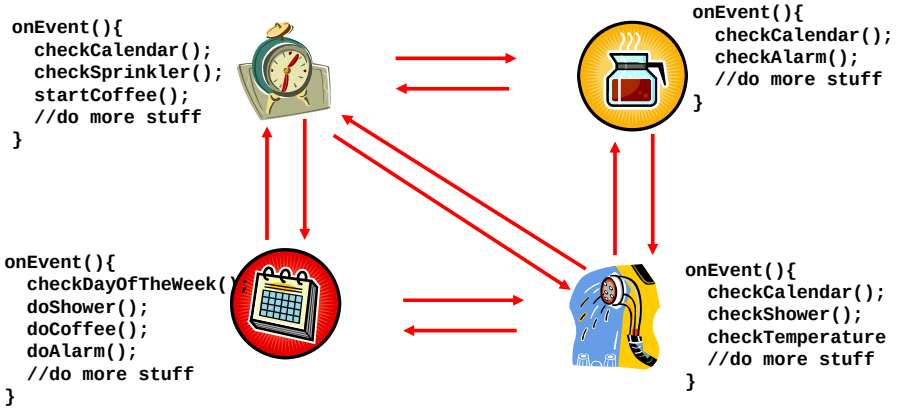
The Mediator – Non Software Example



- 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.

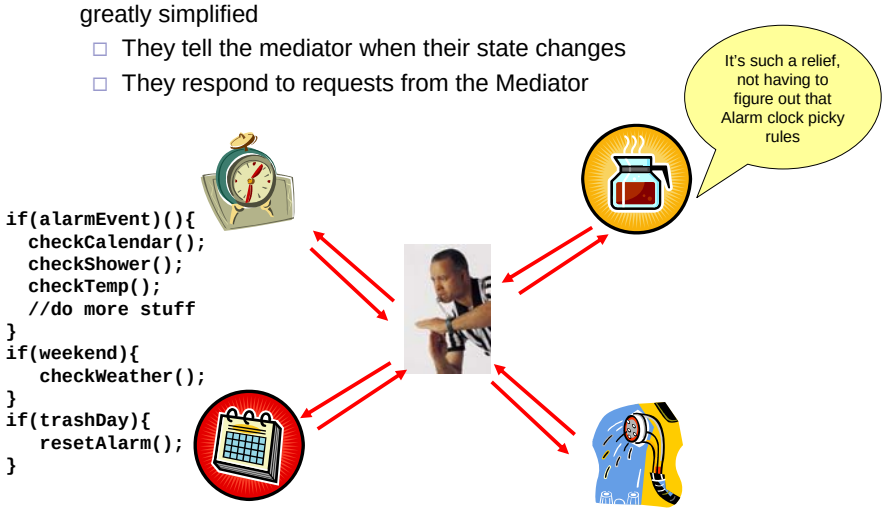
The Mediator – Another Example

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

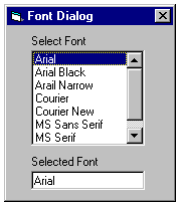


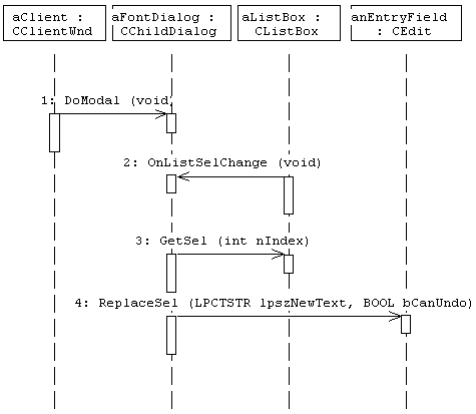
The Mediator in Action

- 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



Mediator and MFC (Microsoft Foundation Classes)

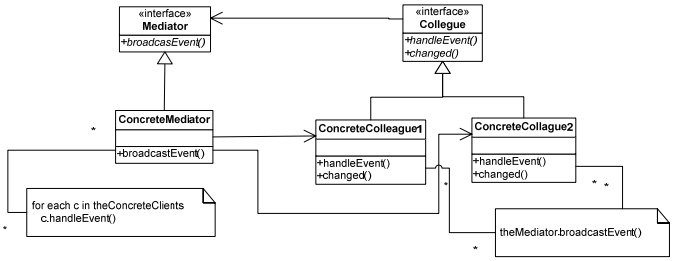




- The Client creates aFontDialog and invokes it.
- The list box tells the FontDialog (it's mediator) that it has changed
- The FontDialog (the mediator object) gets the selection from the list box
- The FontDialog (the mediator object) passes the selection to the entry field edit box

pelab

Actors in the Mediator Pattern



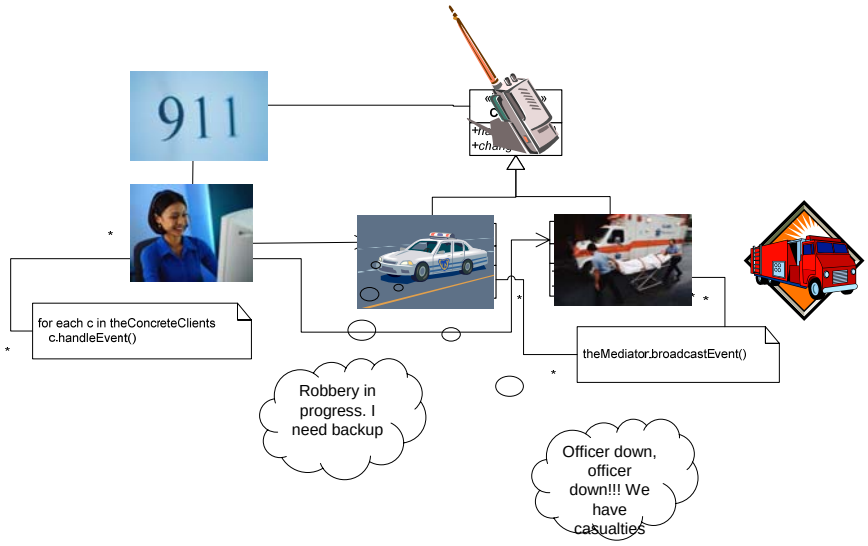
Mediator
defines an interface for communicating with Colleague objects

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

pelab

Yet Another Example



Mediator advantages and disadvantages

- Changing the system behavior means just sub classing 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.

Issues

- 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 — OO Architecture

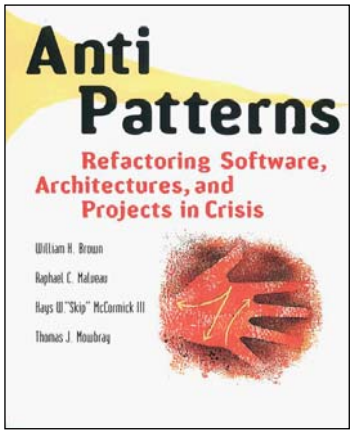
Application-Architecture — Subsystem

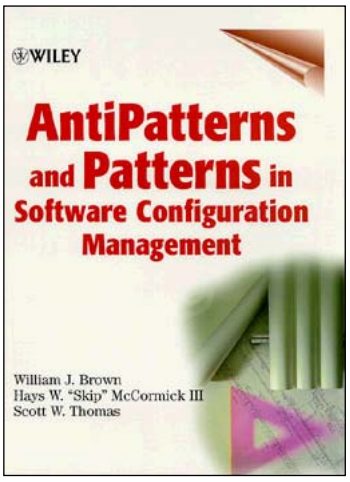
Macro-Architecture — Frameworks

Micro-Architecture — Design-Patterns

Objects — OO Programming

Antipatterns Sources





Congratulations: You have now completed TDDB84