Decorator Pattern

Shin-Jie Lee (李信杰)
Associate Professor
Computer and Network Center
Department of CSIE
National Cheng Kung University

Design Aspect of Decorator

Responsibilities of an object
without subclassing

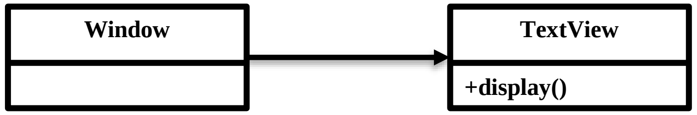
Outline

- Requirements Statement
- Initial Design and Its Problems
- Design Process
- Refactored Design after Design Process
- More Examples
- Recurrent Problems
- Intent
- Decorator Pattern Structure

FileViewer (Decorator)

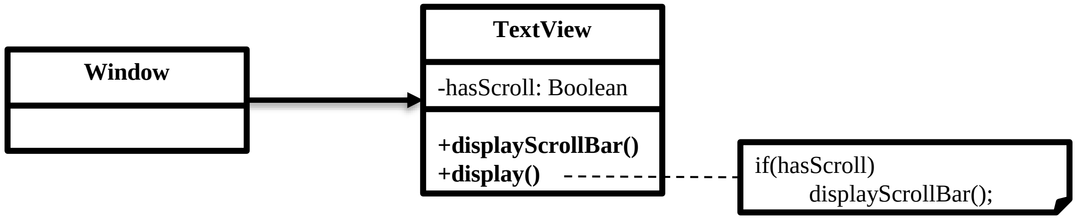
Requirements Statement$_1$

- In FileViewer,
  - We have a TextView object that displays text in a window.



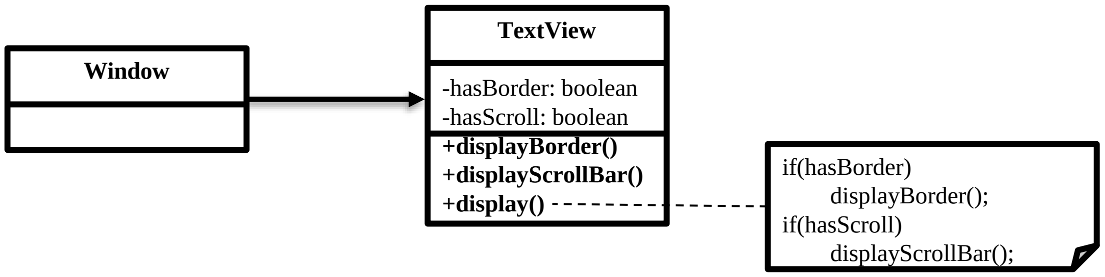
Requirements Statement$_2$

- In FileViewer,
  - TextView has no scroll bars by default, because we might not always need them.



Requirements Statement$_3$

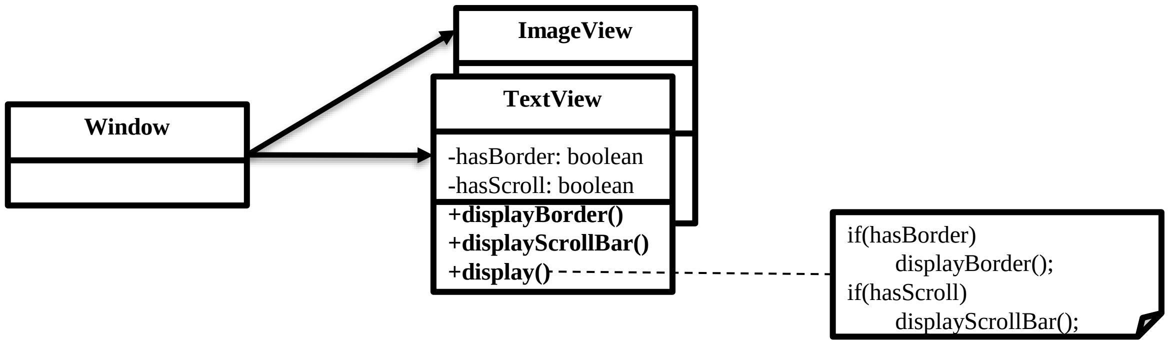
- In FileViewer,
  - We can also add a thick black border around the TextView.



```
if(hasBorder)
      displayBorder();
if(hasScroll)
      displayScrollBar();
```

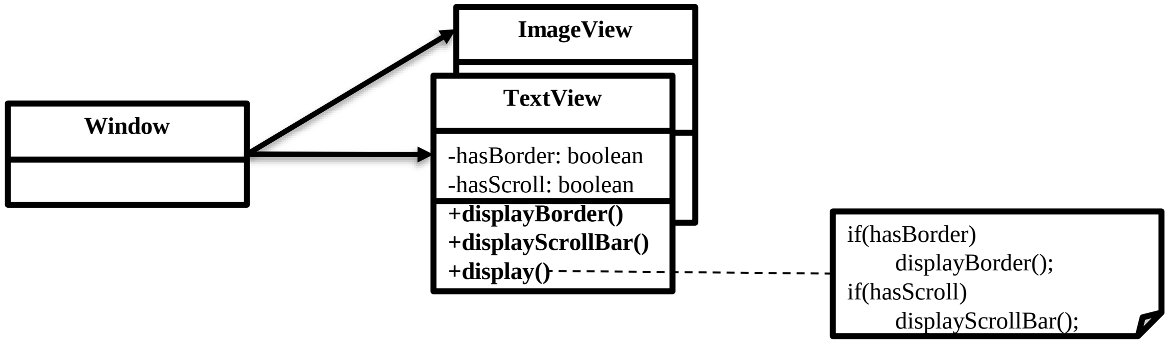
Requirements Statement$_4$

- In FileViewer,
  - It is highly likely that we will support various file formats for display in the future.



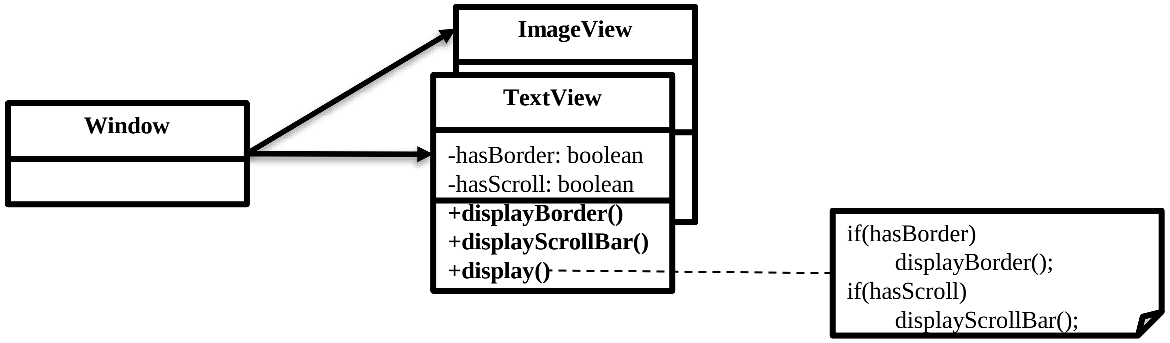
```
if(hasBorder)
        displayBorder();
if(hasScroll)
        displayScrollBar();
```

Initial Design



```
if(hasBorder)
    displayBorder();
if(hasScroll)
    displayScrollBar();
```

Problems with Initial Design

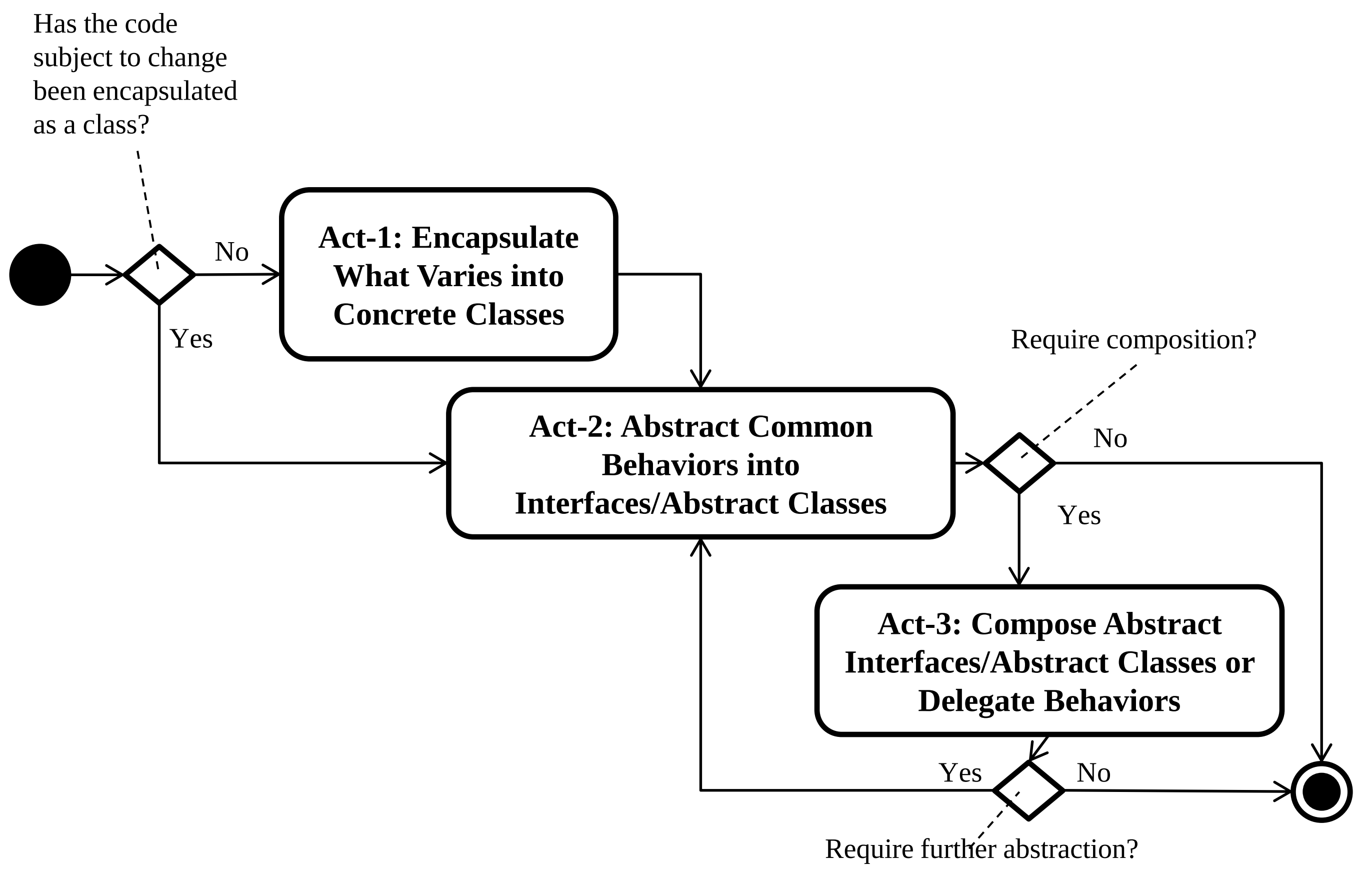


```
if(hasBorder)
        displayBorder();
if(hasScroll)
        displayScrollBar();
```

Problem 1: If we want not only scroll bars or thick borders but many other UI components, such as toolbar, we need re-open TextView for modification to meet the new requirement.

Problem 2: At a later time, if we want to support various kinds of file formats, like image, we need to duplicate displayBorder() and displayScrollBar().

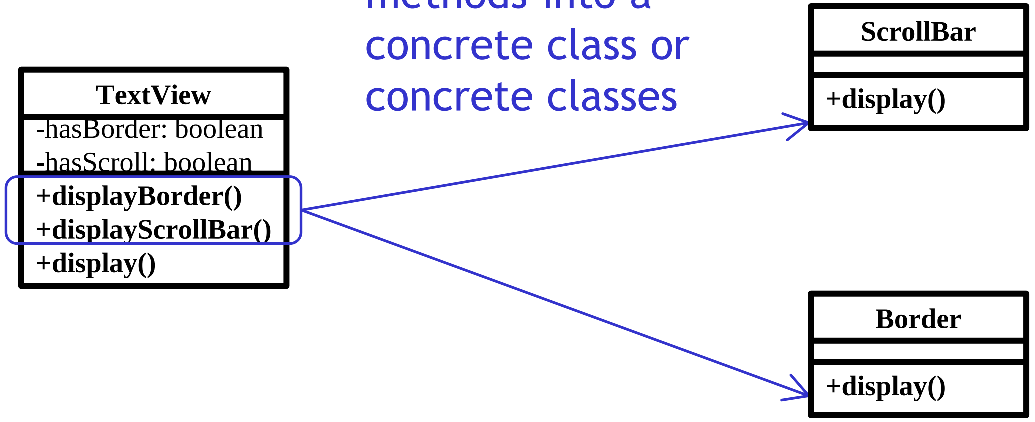
Design Process for Change



Has the code subject to change been encapsulated as a class?
No
Yes
Act-1: Encapsulate What Varies into Concrete Classes
Act-2: Abstract Common Behaviors into Interfaces/Abstract Classes
Require composition?
No
Yes
Act-3: Compose Abstract Interfaces/Abstract Classes or Delegate Behaviors
Yes
No
Require further abstraction?

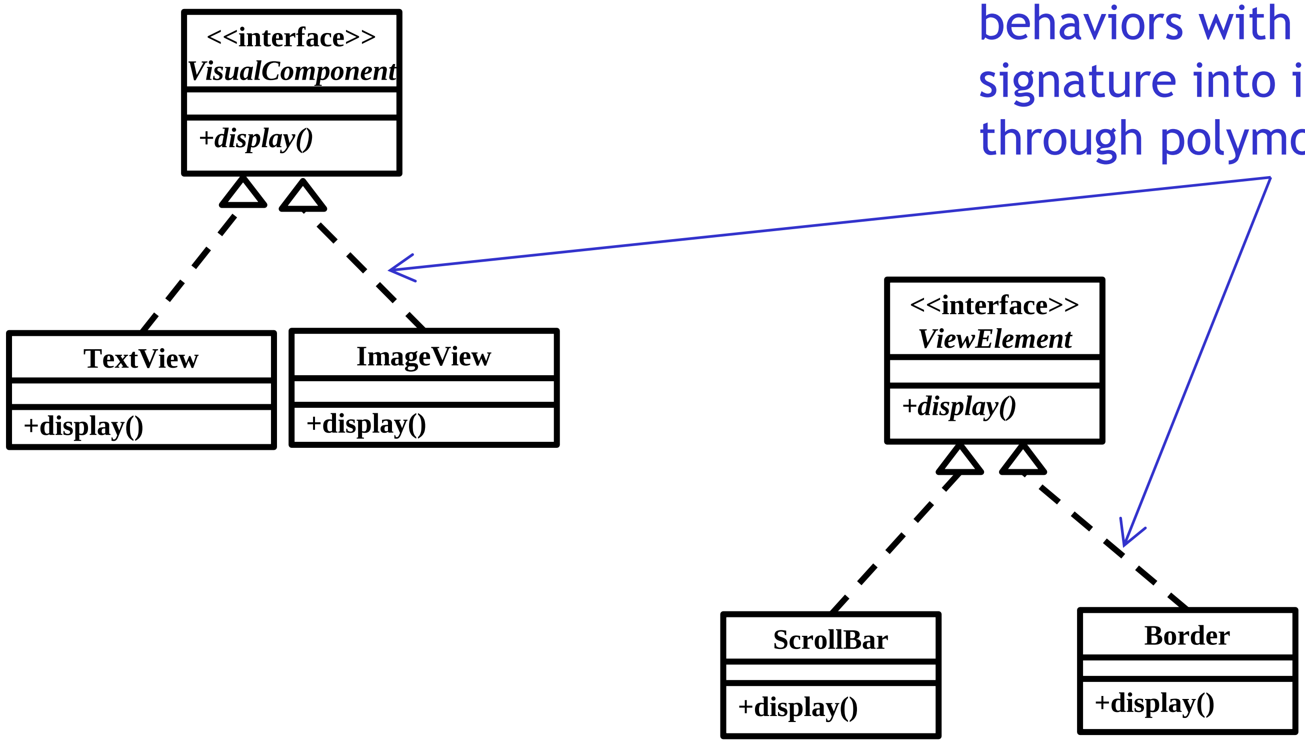
Act-1: Encapsulate What Varies

Act-1.2: Encapsulate a method or a set of methods into a concrete class or concrete classes

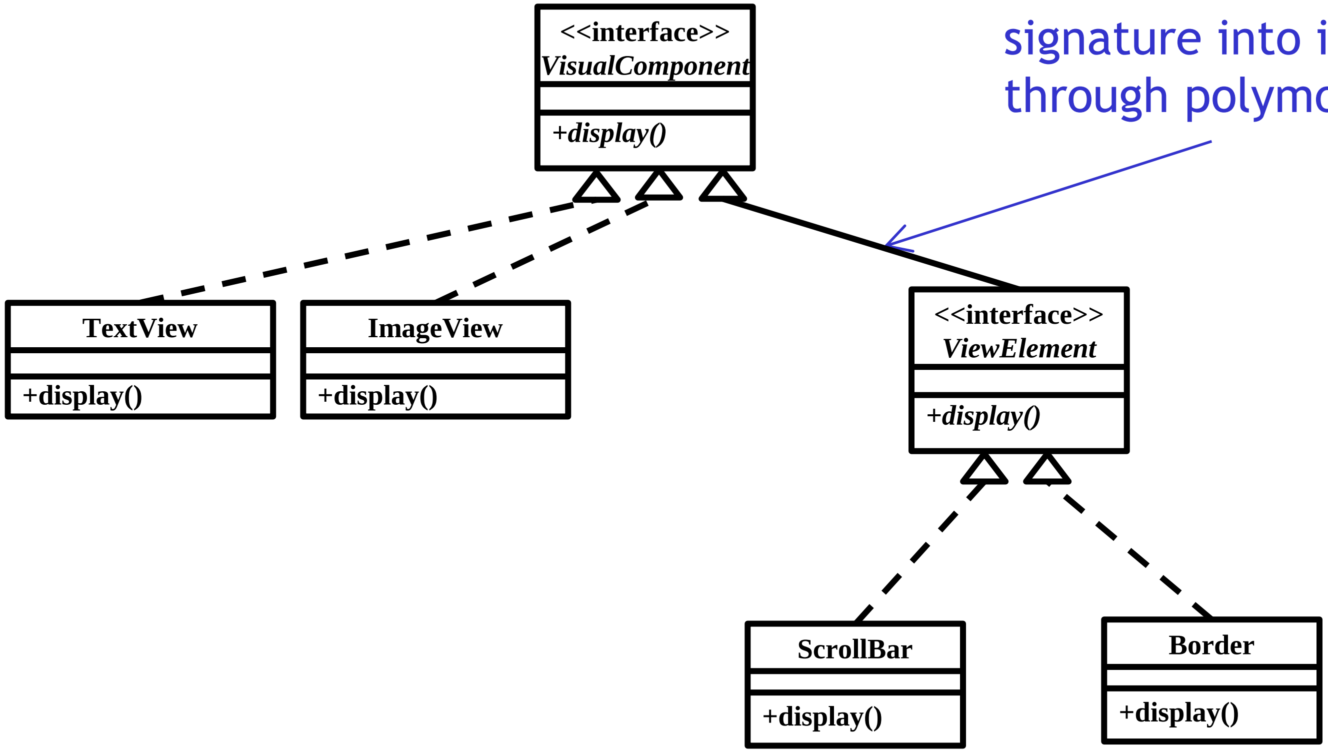


Act-2: Abstract Common Behaviors into Interfaces/Abstract Classes

Act-2.1: Abstract common behaviors with a same signature into interface through polymorphism



Act-2: Abstract Common Behaviors into Interfaces/Abstract Classes

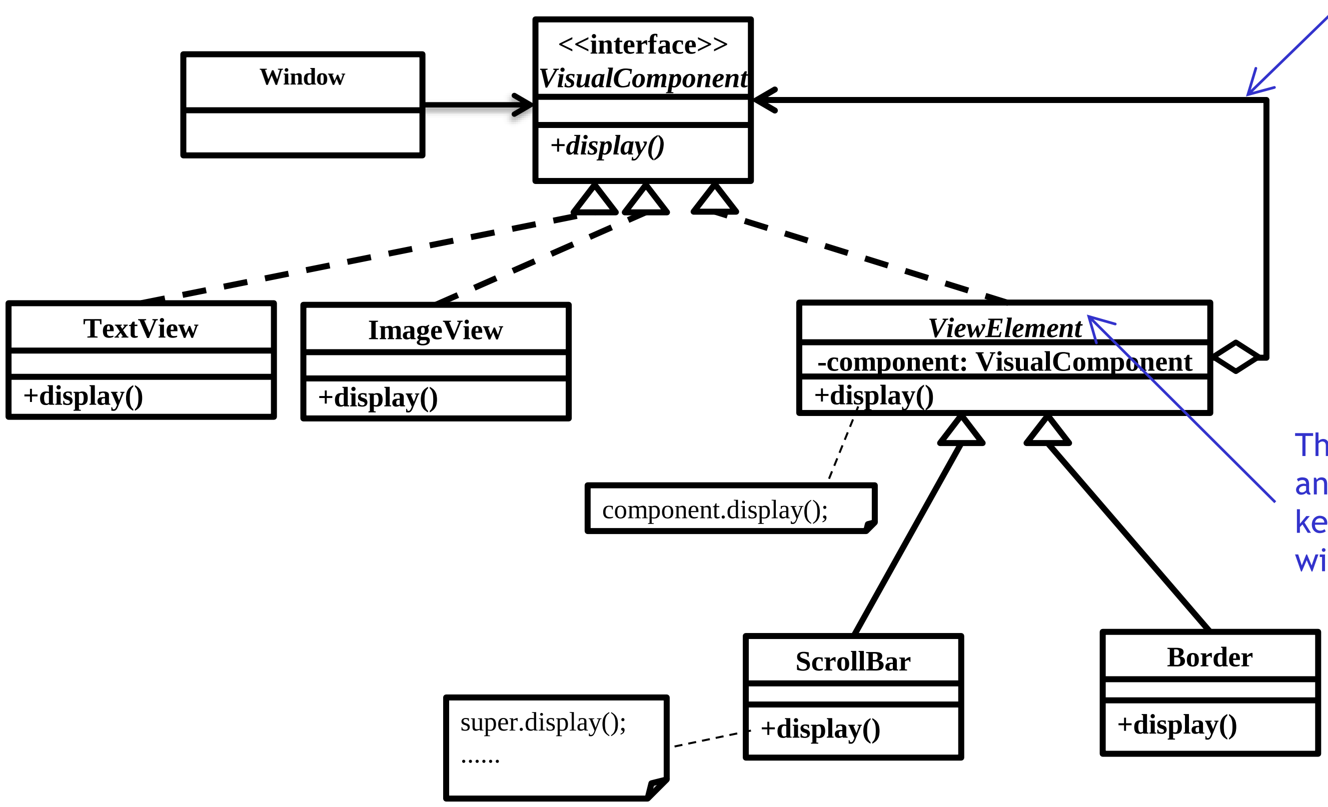


Act-2.1: Abstract common behaviors with a same signature into interface through polymorphism

Act-3: Compose Abstract Behaviors

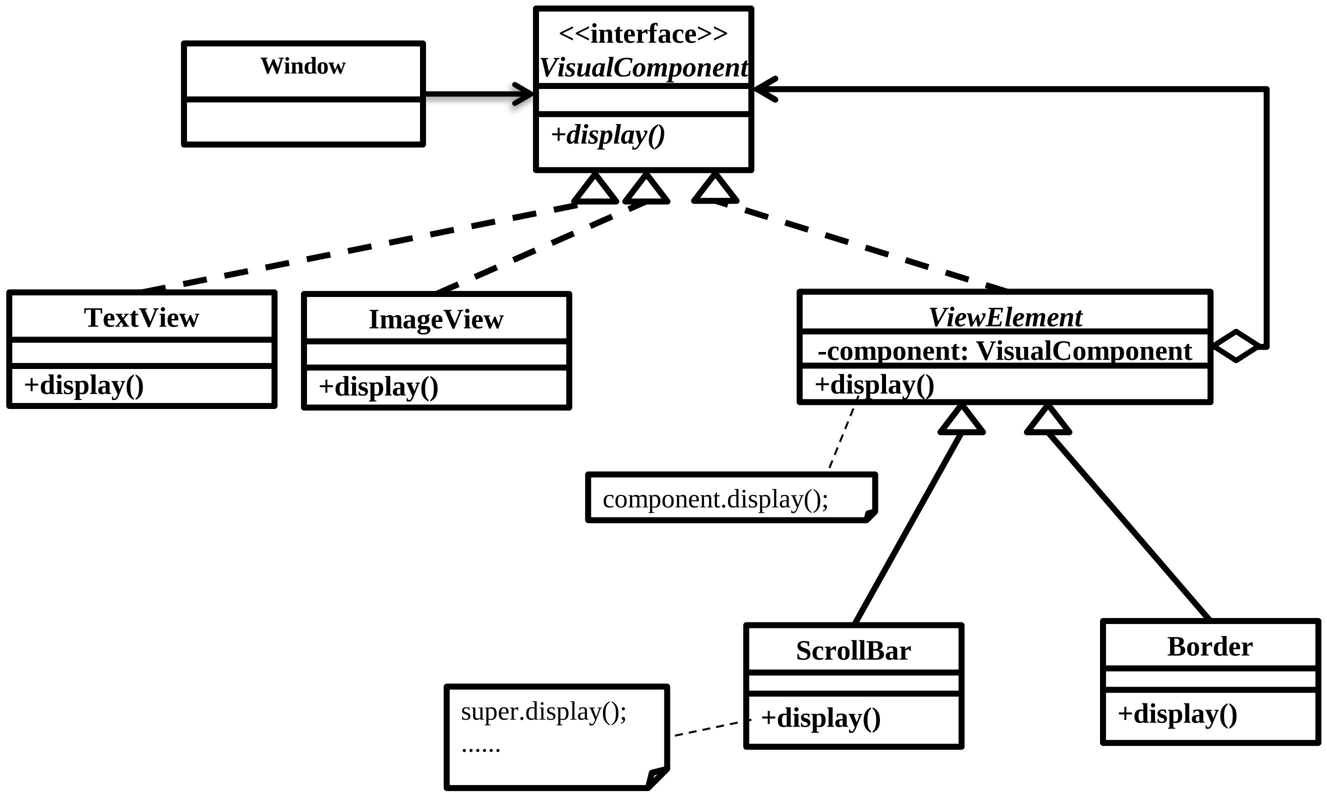
Act-3.1: Compose behaviors (multiple methods) of an interface or an abstract class.

- To enable composing behavior recursively



The interface is changed into an abstract class in order to keep the composition relation with the component attribute

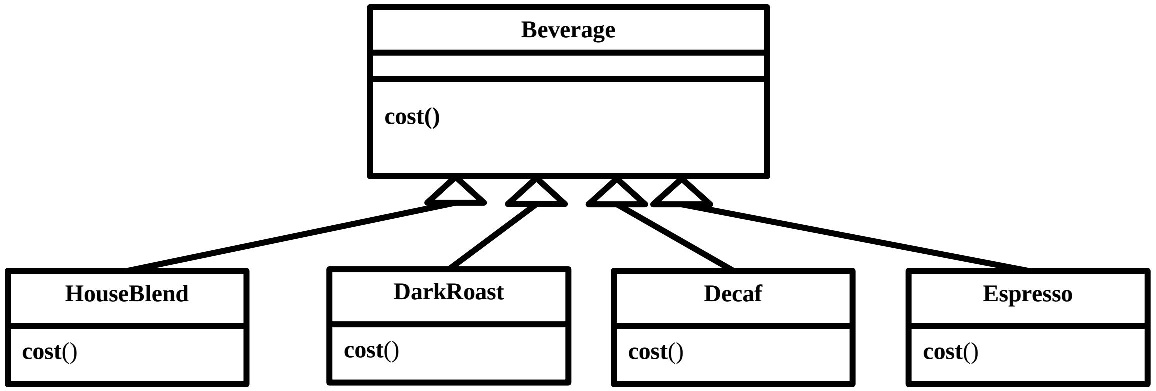
Refactored Design after Design Process



Starbuzz Coffee (Decorator)

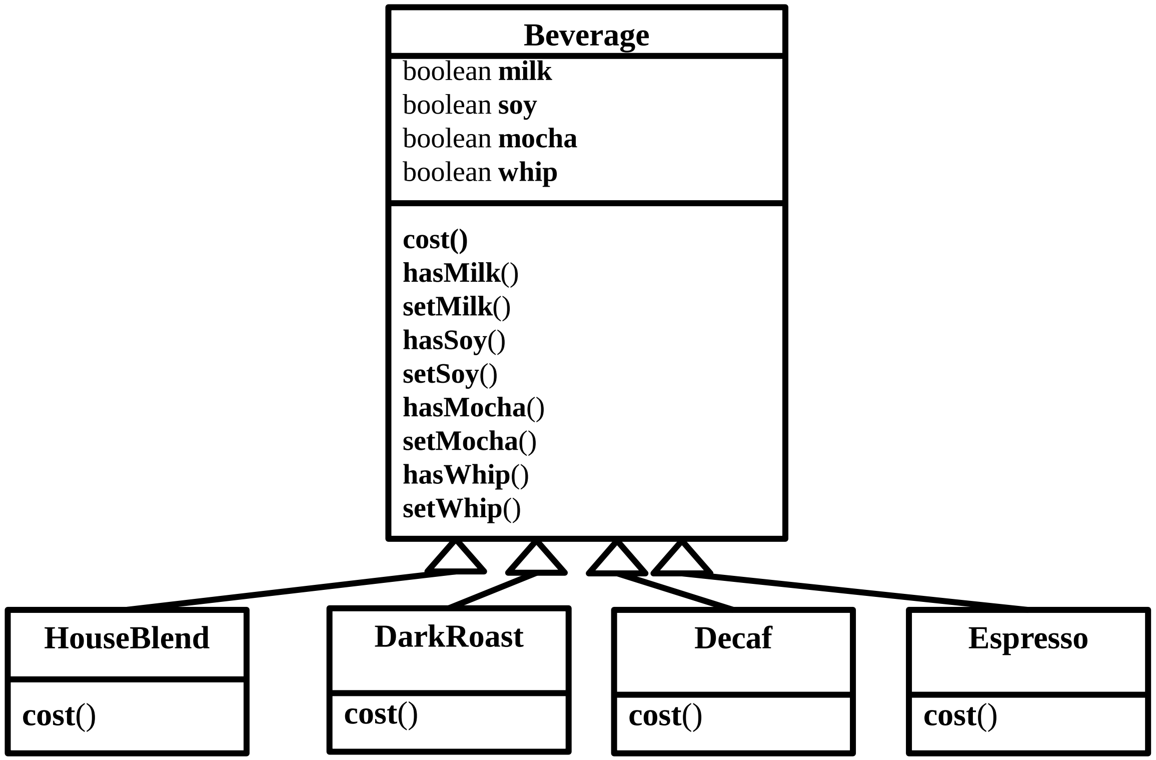
Requirements Statement1

- Starbuzz Coffee
  - Starbuzz Coffee shops are scrambling to update their ordering systems to match their beverage offerings (e.g. HouseBlend, DarkRoast, Decaf and Espresso) to summate how they cost.

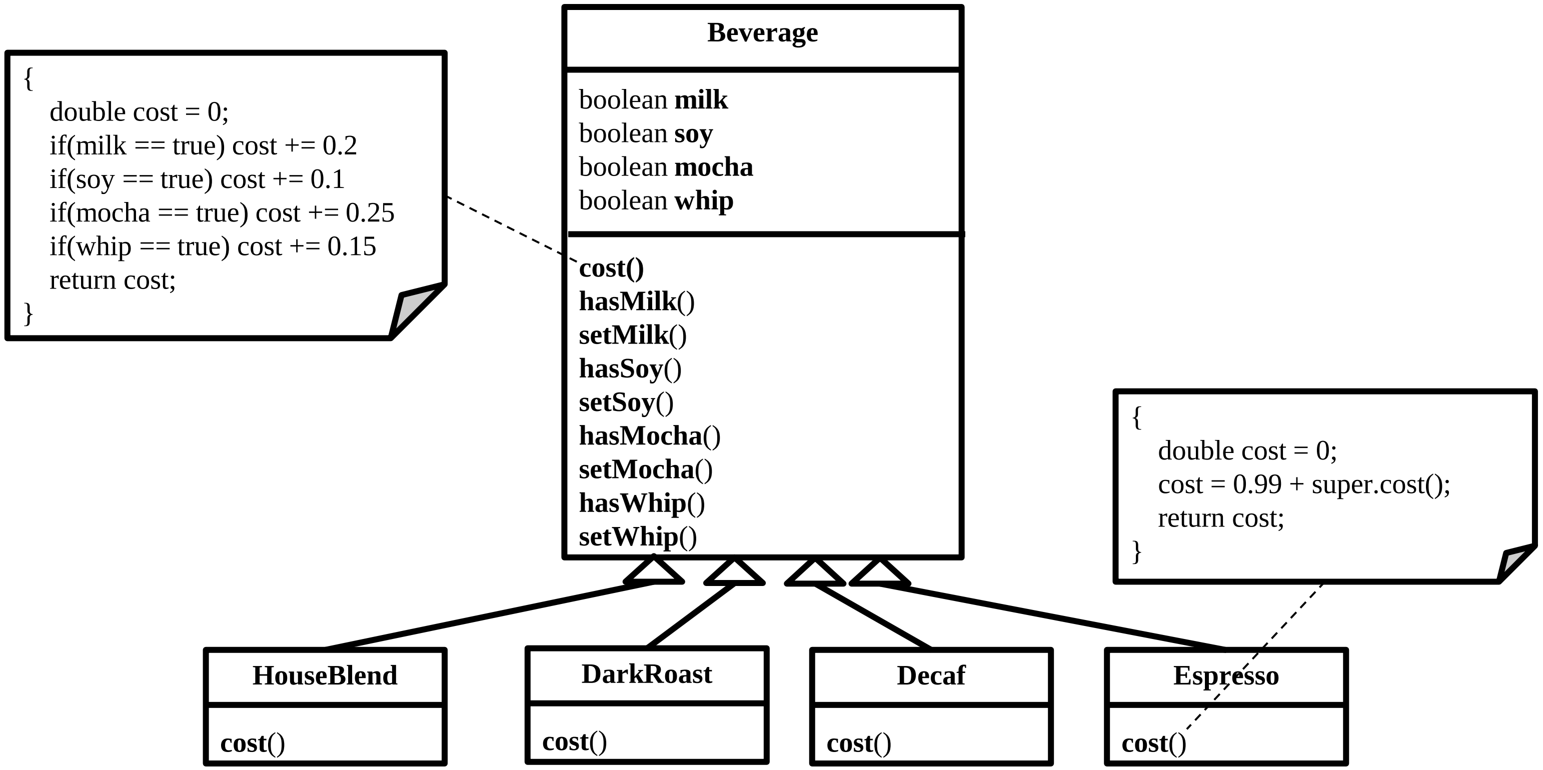


Requirements Statement₂

- In addition to your coffee, you can also ask for several condiments like steamed milk, soy, and mocha, and have these, so they really need to get them built into their order system

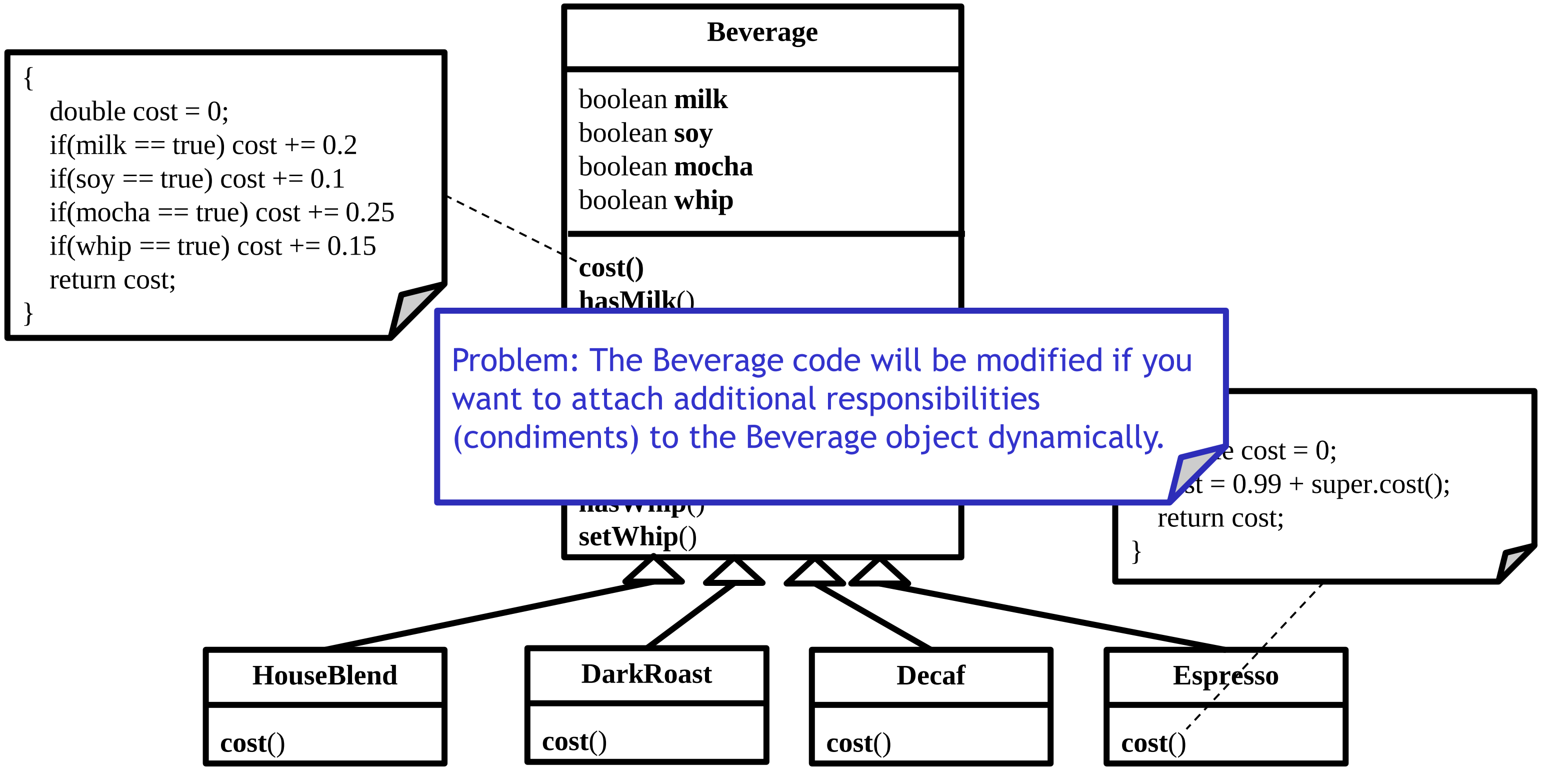


Initial Design - Class Diagram


```
{
   double cost = 0;
   if(milk == true) cost += 0.2
   if(soy == true) cost += 0.1
   if(mocha == true) cost += 0.25
   if(whip == true) cost += 0.15
   return cost;
}
```

```
{
   double cost = 0;
   cost = 0.99 + super.cost();
   return cost;
}
```

Problems with Initial Design

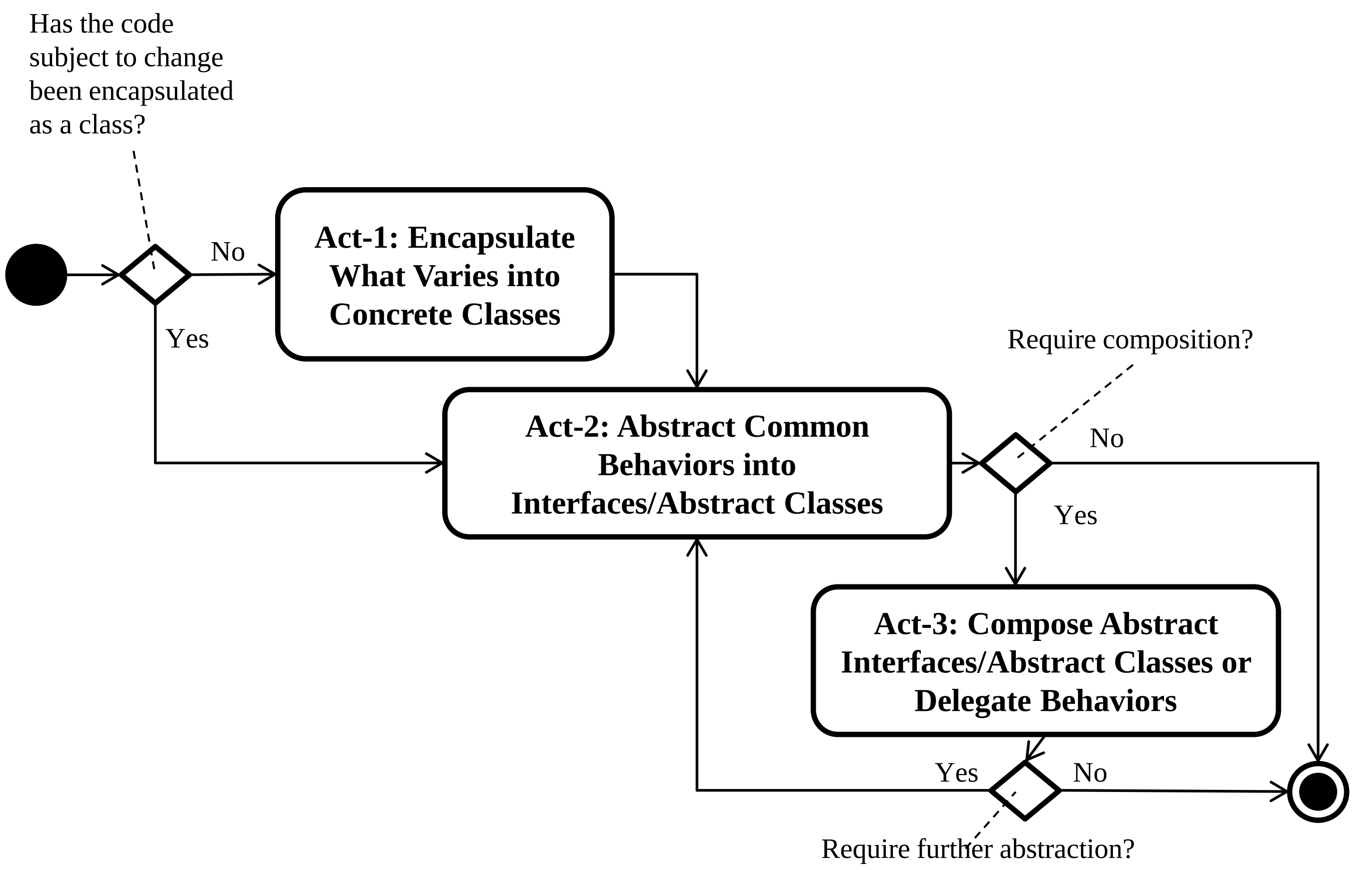


```
{
   double cost = 0;
   if(milk == true) cost += 0.2
   if(soy == true) cost += 0.1
   if(mocha == true) cost += 0.25
   if(whip == true) cost += 0.15
   return cost;
}
```

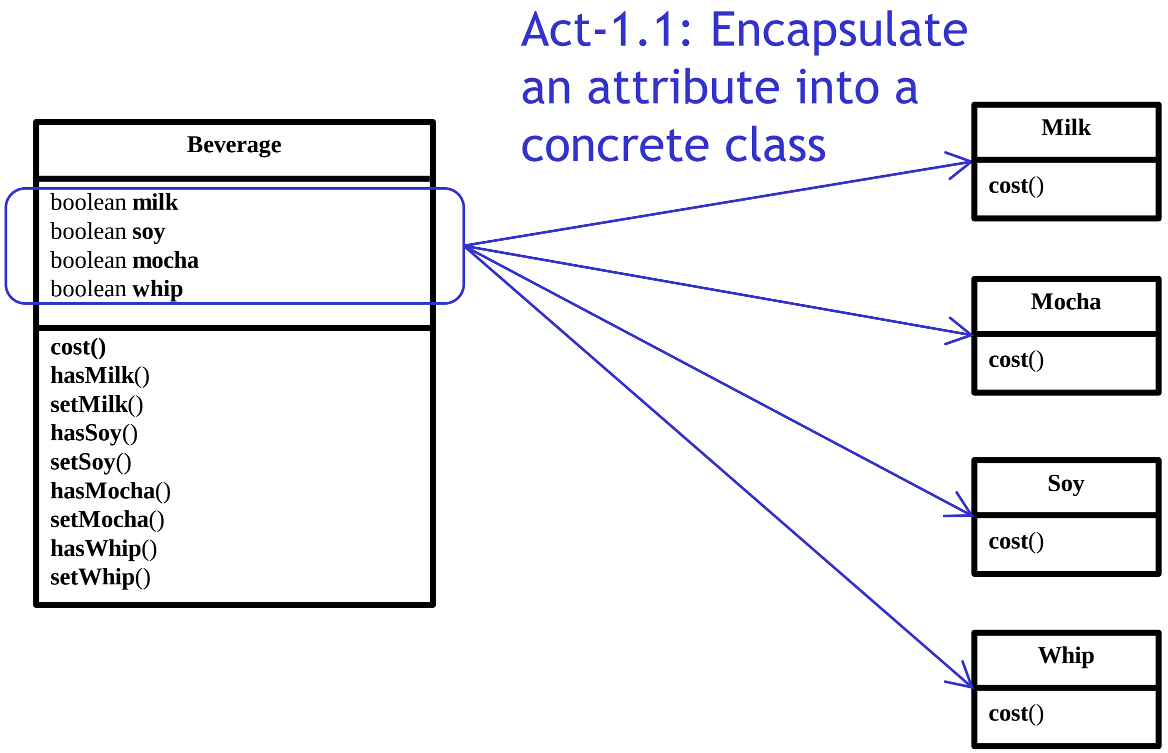
```
   e cost = 0;
   st = 0.99 + super.cost();
   return cost;
}
```

Problem: The Beverage code will be modified if you want to attach additional responsibilities (condiments) to the Beverage object dynamically.

Design Process for Change

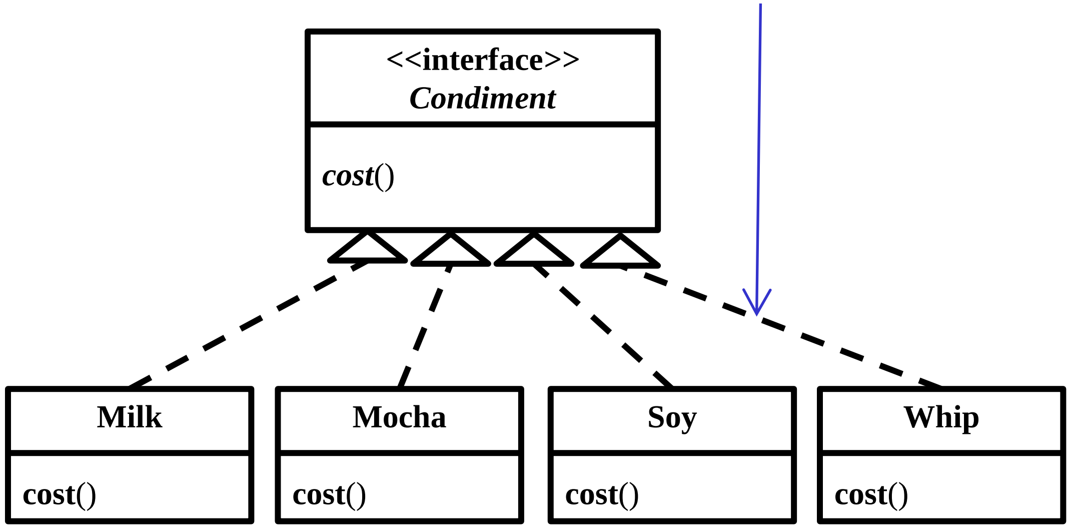

Has the code subject to change been encapsulated as a class?
No
Yes
Act-1: Encapsulate What Varies into Concrete Classes
Act-2: Abstract Common Behaviors into Interfaces/Abstract Classes
Require composition?
No
Yes
Act-3: Compose Abstract Interfaces/Abstract Classes or Delegate Behaviors
Yes
No
Require further abstraction?

Act-1: Encapsulate What Varies



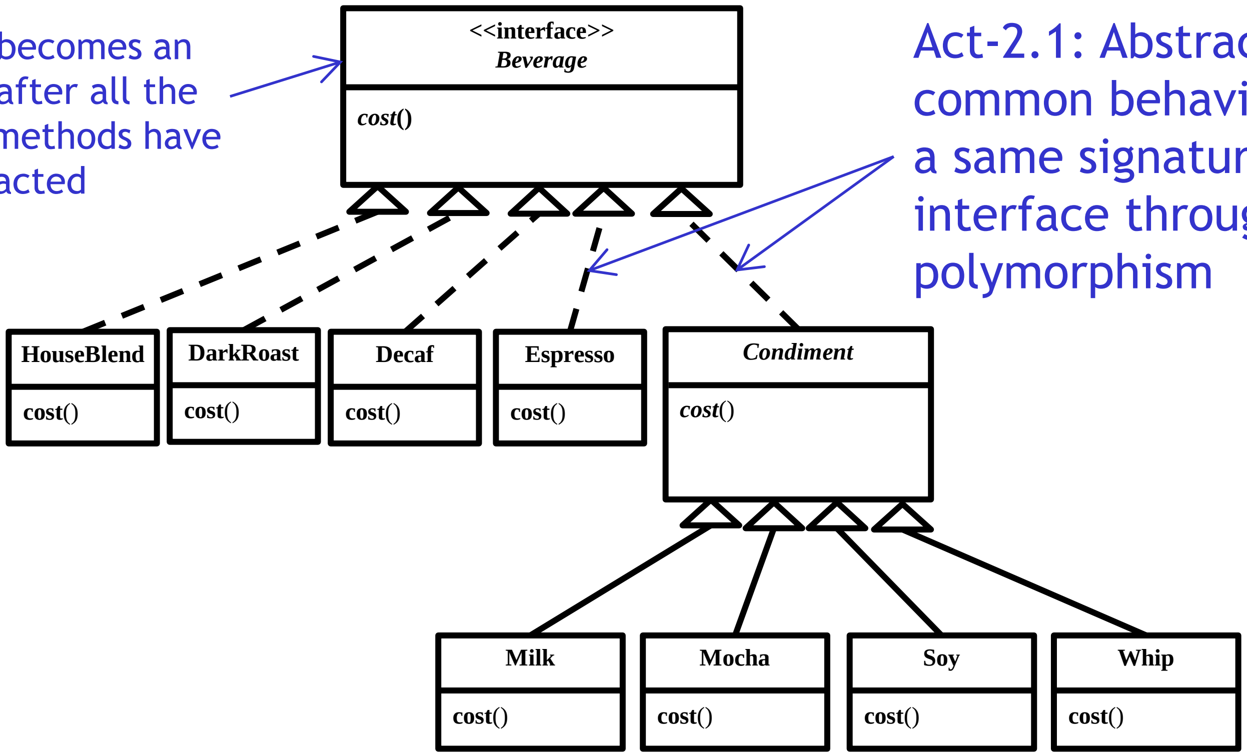
Act-2: Abstract Common Behaviors into Interfaces/Abstract Classes

Act-2.1: Abstract common behaviors with a same signature into interface through polymorphism



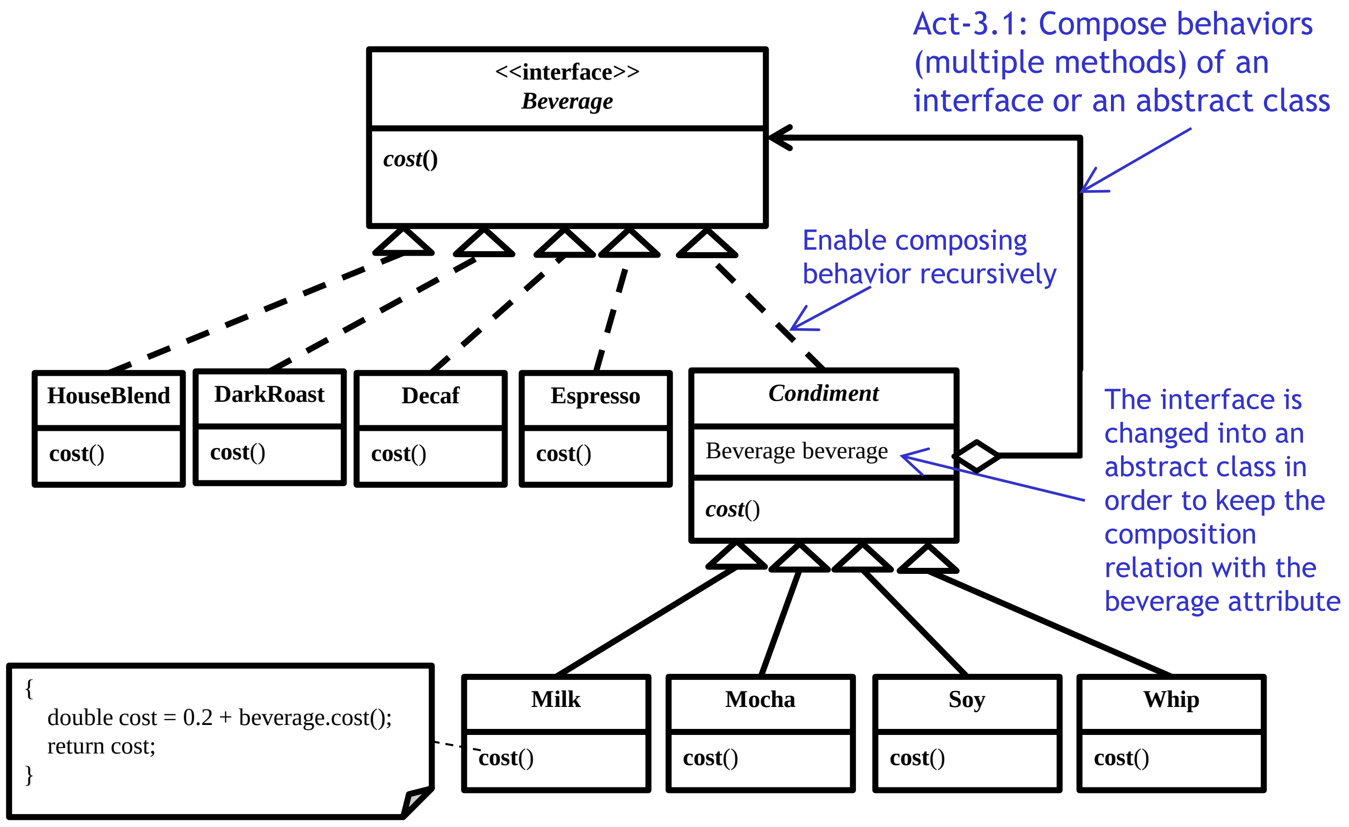
Act-2: Abstract Common Behaviors into Interfaces/Abstract Classes

Beverage becomes an interface after all the concrete methods have been extracted



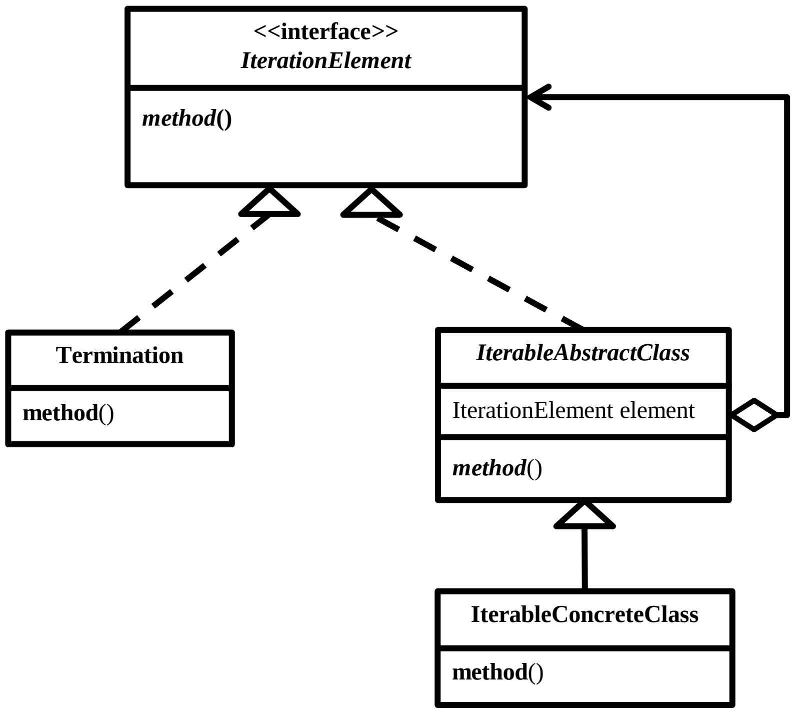
Act-2.1: Abstract common behaviors with a same signature into interface through polymorphism

Act-3: Compose Abstract Behaviors

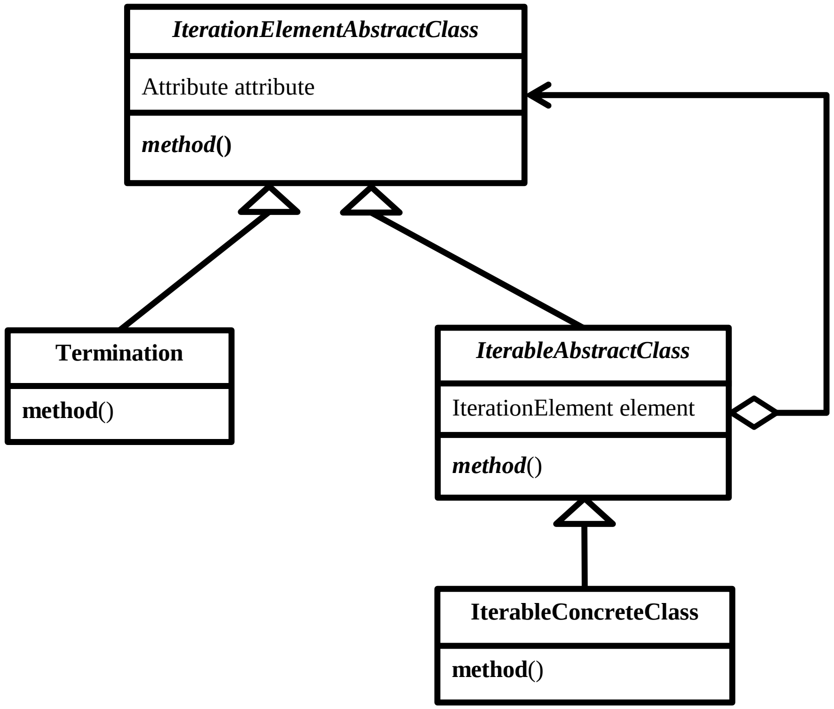

Act-3.1: Compose behaviors (multiple methods) of an interface or an abstract class
<<interface>>
Beverage
cost()
Enable composing behavior recursively
HouseBlend
cost()
DarkRoast
cost()
Decaf
cost()
Espresso
cost()
Condiment
Beverage beverage
cost()
The interface is changed into an abstract class in order to keep the composition relation with the beverage attribute
Milk
cost()
Mocha
cost()
Soy
cost()
Whip
cost()

```
{
    double cost = 0.2 + beverage.cost();
    return cost;
}
```

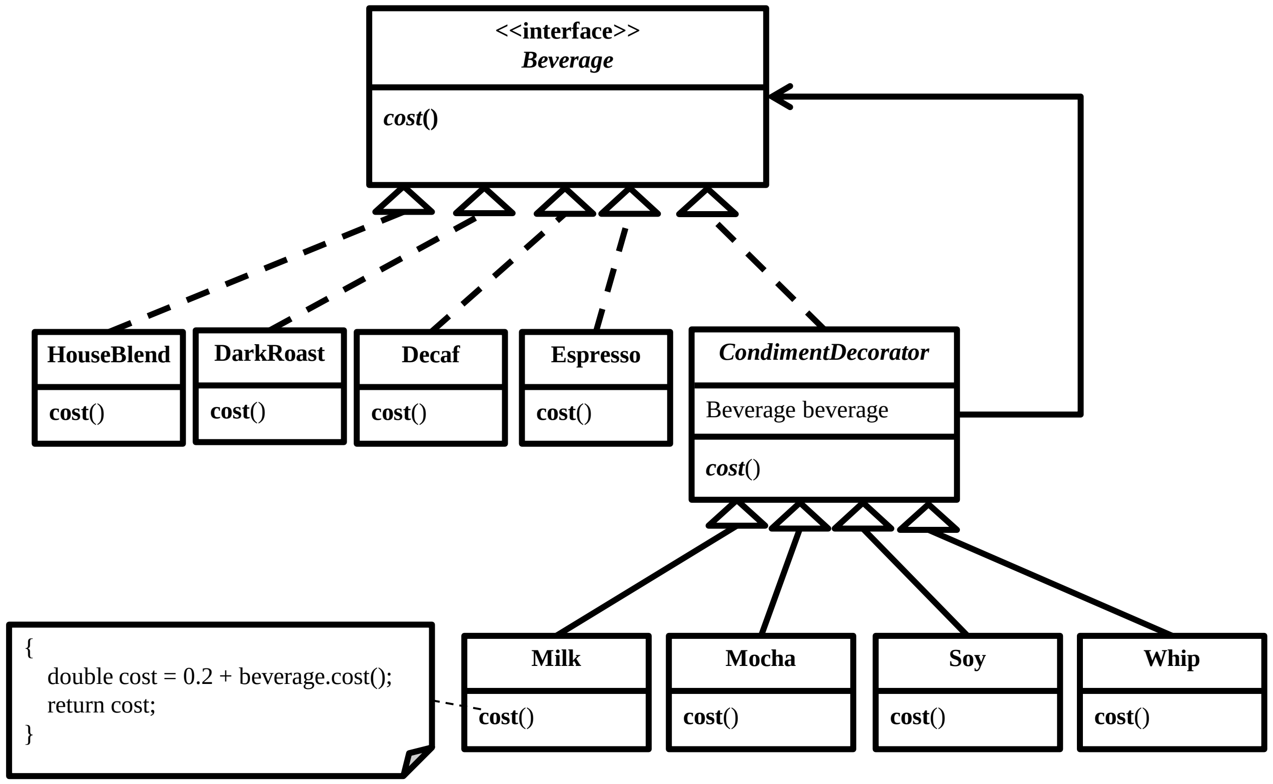
Recursive Design$_1$



Recursive Design$_2$



Refactored Design after Design Process



```
{
    double cost = 0.2 + beverage.cost();
    return cost;
}
```

Example

1. Create a Milk object
2. Create a Mocha object
3. Create a DarkRoast object
4. Decorate the DarkRoast object with the Mocka object (Set the Beverage attribute object value of the Mocka object to be the DarkRoast object)
5. Decorate the Mocka object with the Milk object (Set the Beverage attribute object value of the Milk object to be the Mocka object)
6. Calculate the cost by invoking the cost() of the top decorator (the Milk object)

Recurrent Problem$_1$

- A class will be modified if you want to attach additional responsibilities (decorators) to an object dynamically.
  - Sometimes we want to add responsibilities to individual objects, not to an entire class. A graphical user interface toolkit.
  - For example, should let you add properties like borders or behaviors like scrolling to any user interface component.

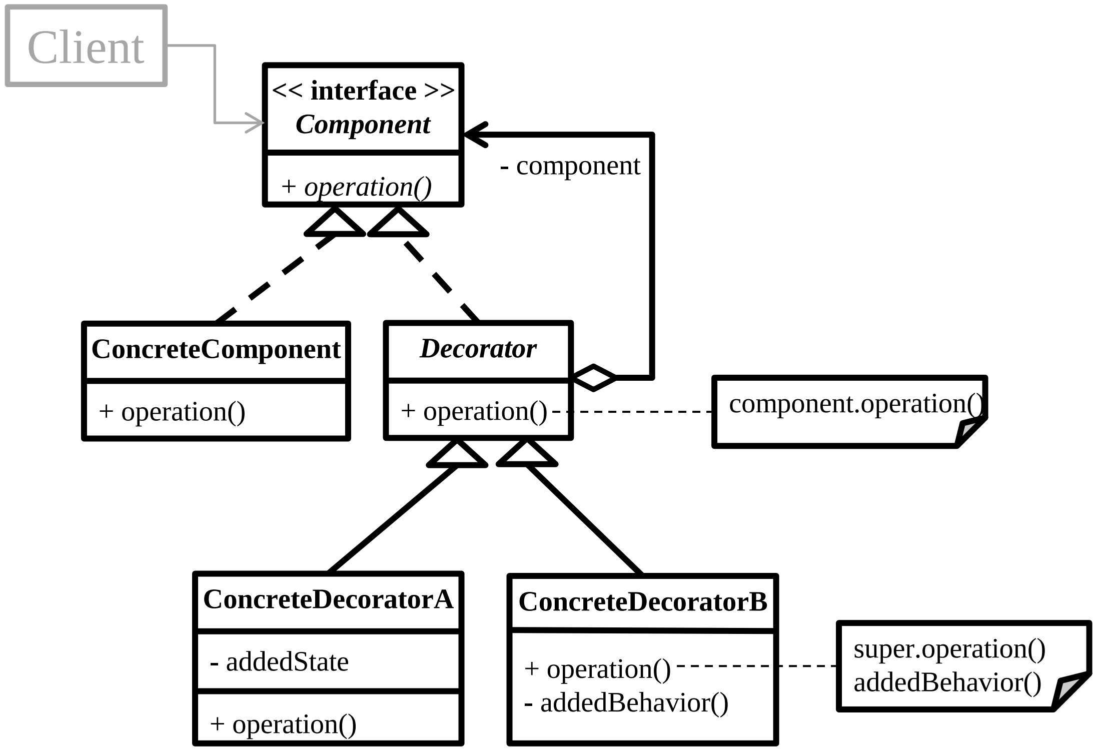
Recurrent Problem$_2$

- One way to add responsibilities is with inheritance. Inheriting a border from another class puts a border around every subclass instance.
- This is inflexible, however, because the choice of border is made statically.
- A client can't control how and when to decorate the component with a border.

Intent

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

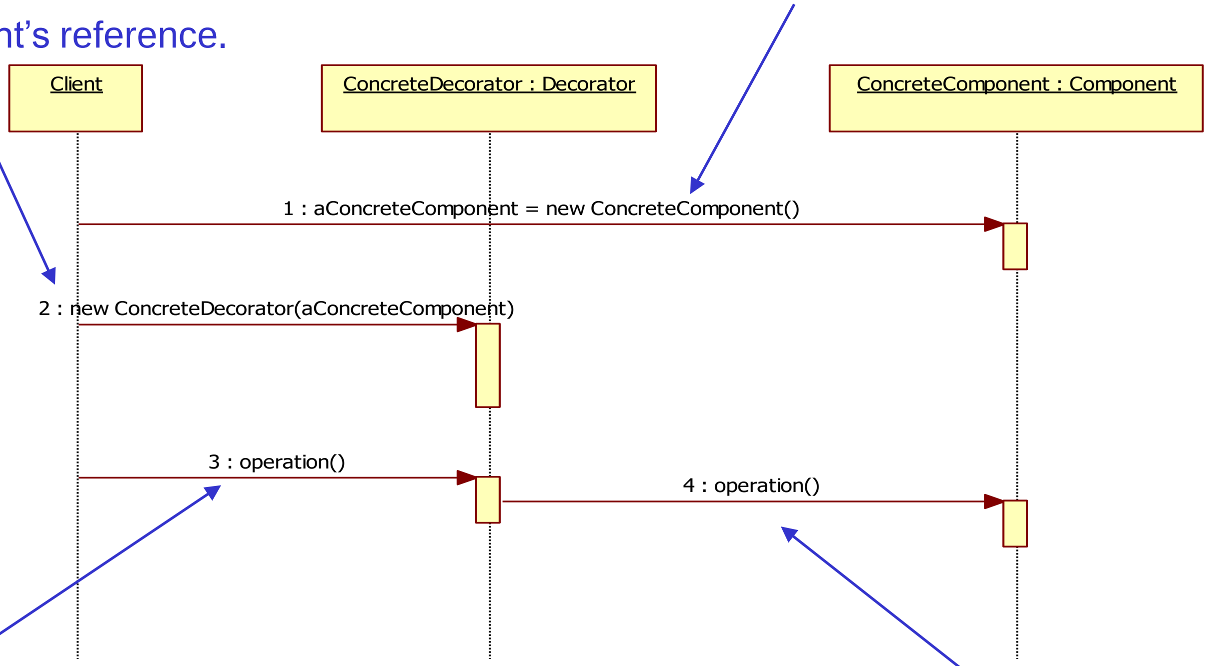
Decorator Pattern Structure$_1$



Decorator Pattern Structure$_2$

2. Client is responsible for creating ConcreteDecorator's instance with ConcreteComponent's reference.

1. Client is responsible for creating ConcreteComponent's instance.



Client
ConcreteDecorator : Decorator
ConcreteComponent : Component
1 : aConcreteComponent = new ConcreteComponent()
2 : new ConcreteDecorator(aConcreteComponent)
3 : operation()
4 : operation()

3. Client performs the operation through invoking ConcreteDecorator's method.

4. ConcreteDecorator delegates request to what it wrapped(ConcreteComponent).

Decorator Pattern Structure$_3$

	Instantiation	Use	Termination
Client	Other class except classes in the decorator pattern	Other class except classes in the decorator pattern	Other class except classes in the decorator pattern
Component	X	Client and ConcreteDecorator use this interface to invoke ConcreteComponent's and ConcreteDecorator's operation through polymorphism	X
Concrete Component	The client class or other class except classes in the decorator pattern	Client and ConcreteDecorator uses this class to invoke the operation implementation through polymorphism	Classes who hold the reference of ConcreteComponent
Decorator	X	ConcreteDecorator use this abstract class to compose another ConcreteDecorator and ConcreteComponent dynamically	X
Concrete Decorator	The client class or other class except classes in the decorator pattern	Another ConcreteDecorator uses this class to invoke the operation implementation through polymorphism	Classes who hold the reference of ConcreteDecorator