Decorator Pattern

Prof. Jonathan Lee (李允中)
Department of CSIE
National Taiwan University

Design Aspect of Decorator

Responsibilities of an object without subclassing

Outline

- ❑ FileViewer Requirements Statements
- ❑ Initial Design and Its Problems
- ❑ Design Process
- ❑ Refactored Design after Design Process
- ❑ Recurrent Problems
- ❑ Intent
- ❑ Decorator Pattern Structure
- ❑ NTU Coffee Shop: Another Example
- ❑ Homework

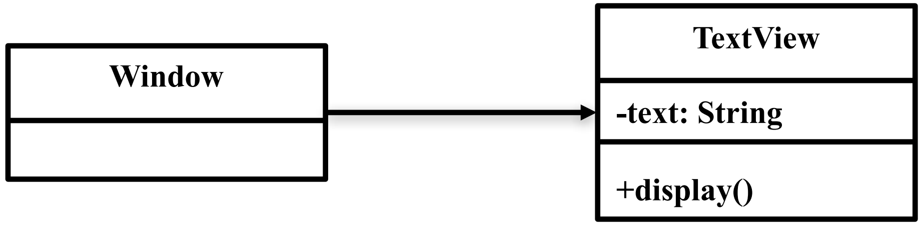
FileViewer (Decorator)

Prof. Jonathan Lee (李允中)

Department of Computer Science and
Information Engineering
National Taiwan University

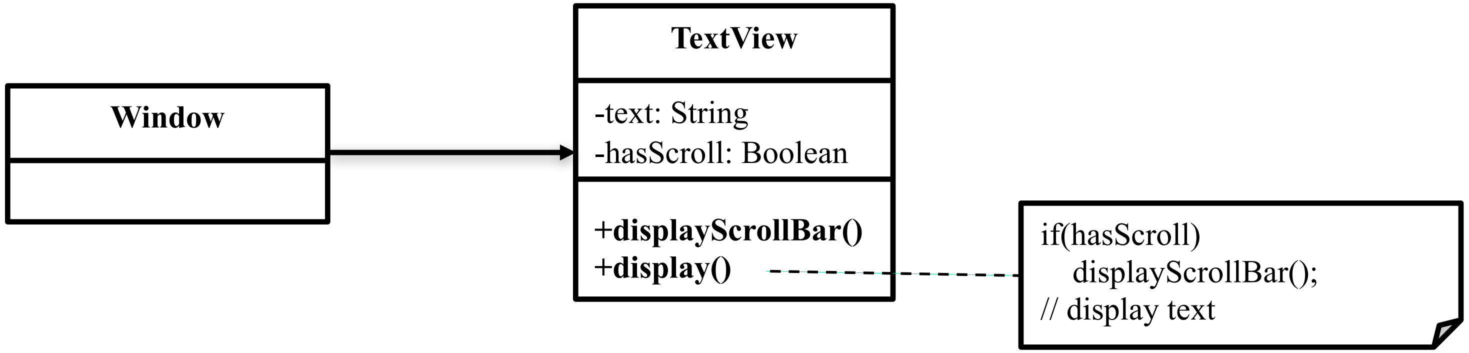
Requirements Statements$_1$

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



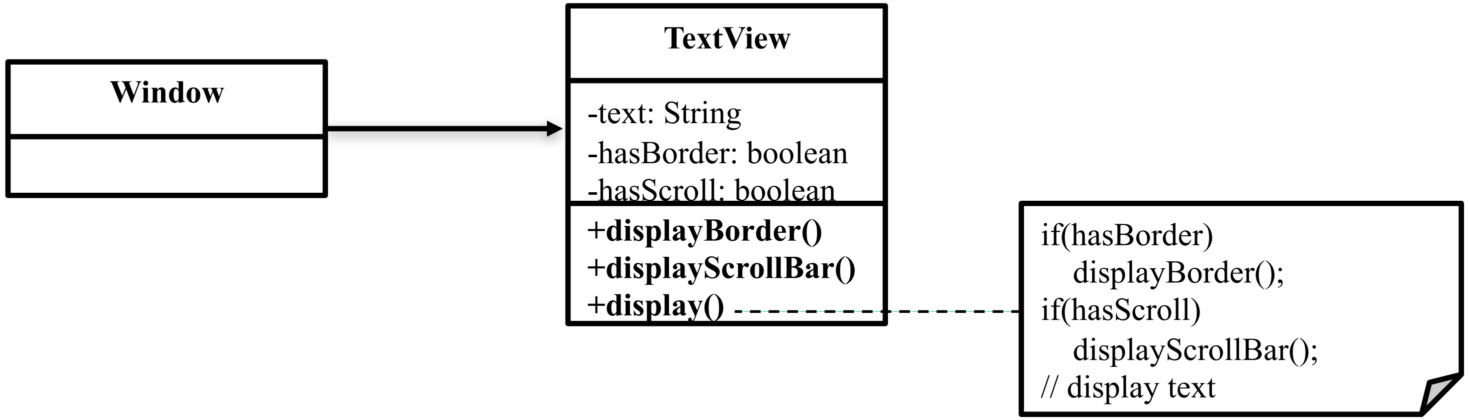
Requirements Statements$_2$

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



Requirements Statements$_3$

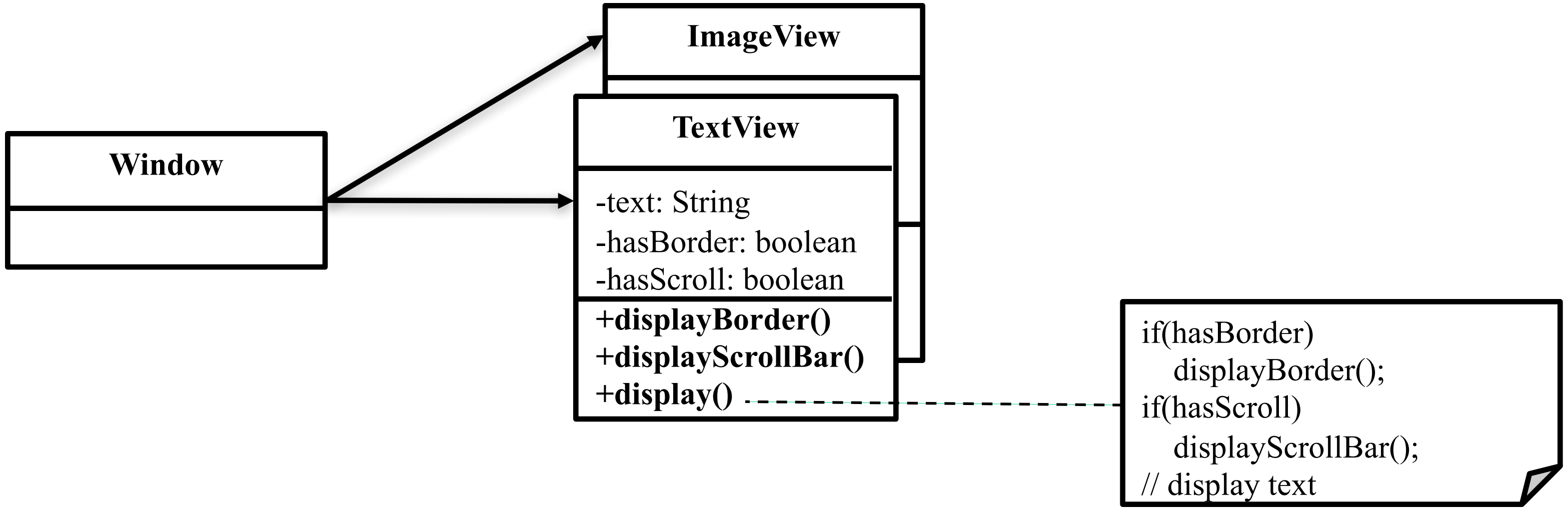
- In FileViewer,
  - We can also add a thick black border around the TextView.



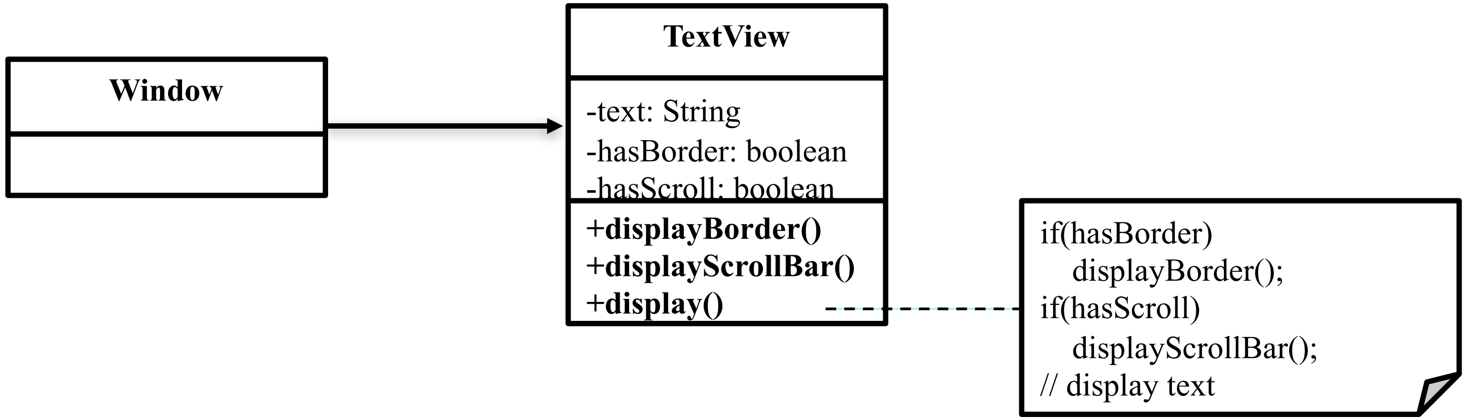
```
if(hasBorder)
   displayBorder();
if(hasScroll)
   displayScrollBar();
// display text
```

Requirements Statements$_4$

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

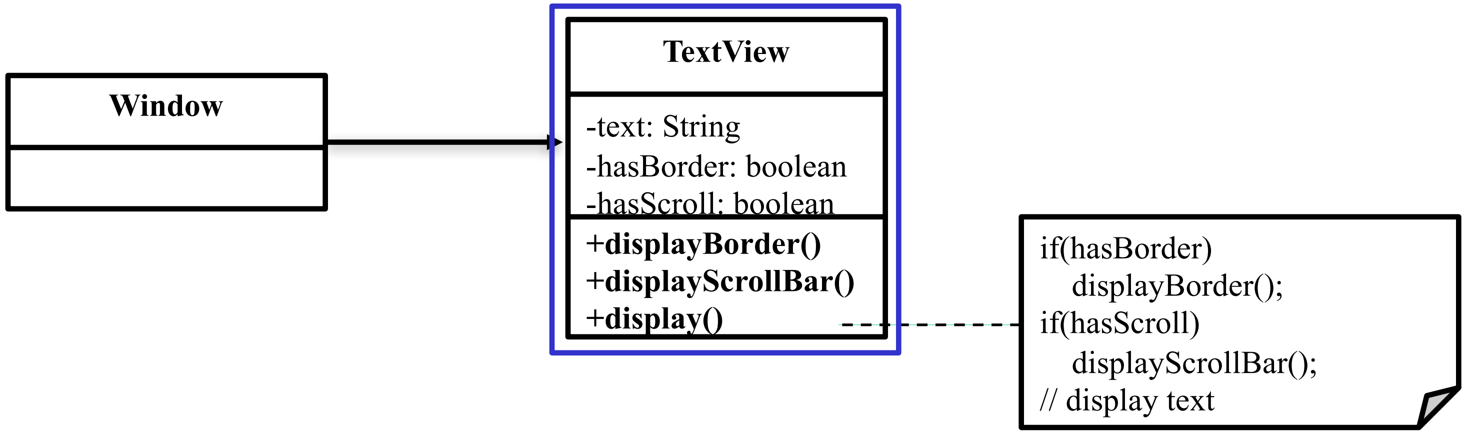


Initial Design



```
if(hasBorder)
   displayBorder();
if(hasScroll)
   displayScrollBar();
// display text
```

Problems with Initial Design

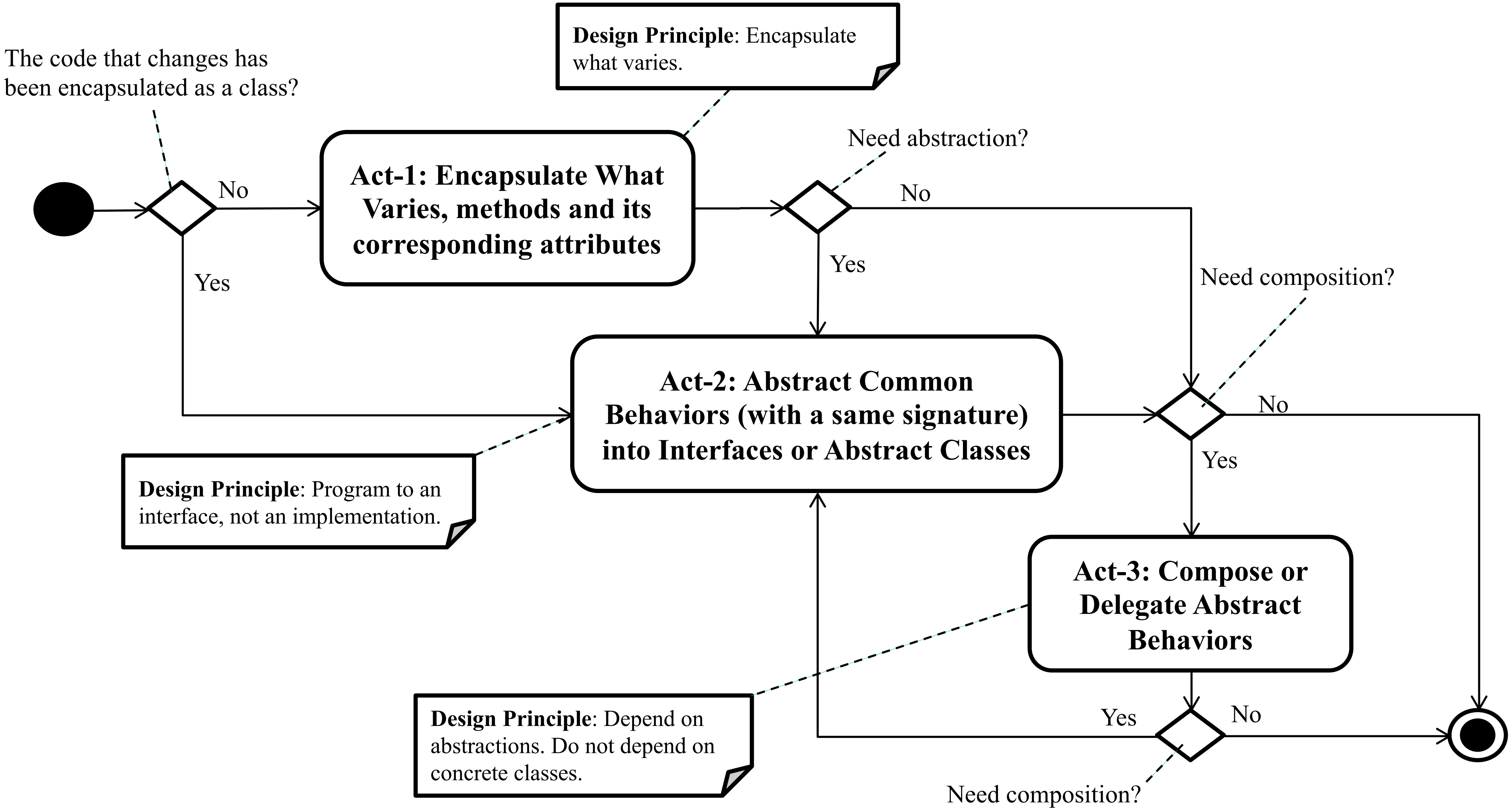


```
if(hasBorder)
    displayBorder();
if(hasScroll)
    displayScrollBar();
// display text
```

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 drawBorder() and drawScrollBar().

Design Process for Change



The code that changes has been encapsulated as a class?

Design Principle: Encapsulate what varies.

Act-1: Encapsulate What Varies, methods and its corresponding attributes

Need abstraction?

Act-2: Abstract Common Behaviors (with a same signature) into Interfaces or Abstract Classes

Design Principle: Program to an interface, not an implementation.

Need composition?

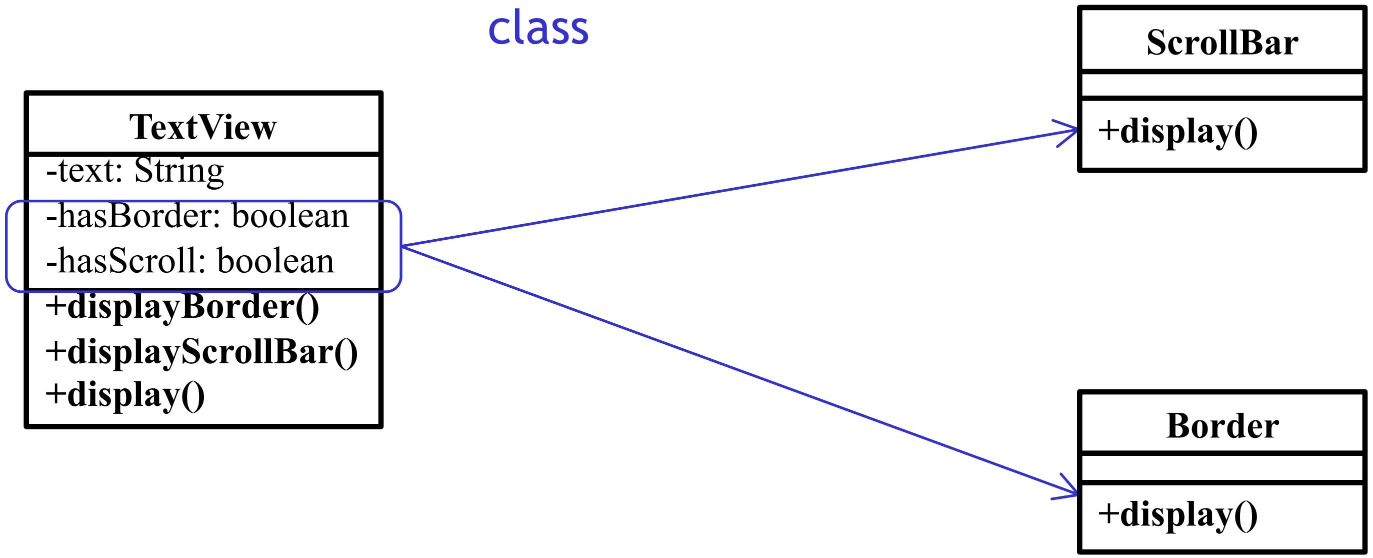
Act-3: Compose or Delegate Abstract Behaviors

Design Principle: Depend on abstractions. Do not depend on concrete classes.

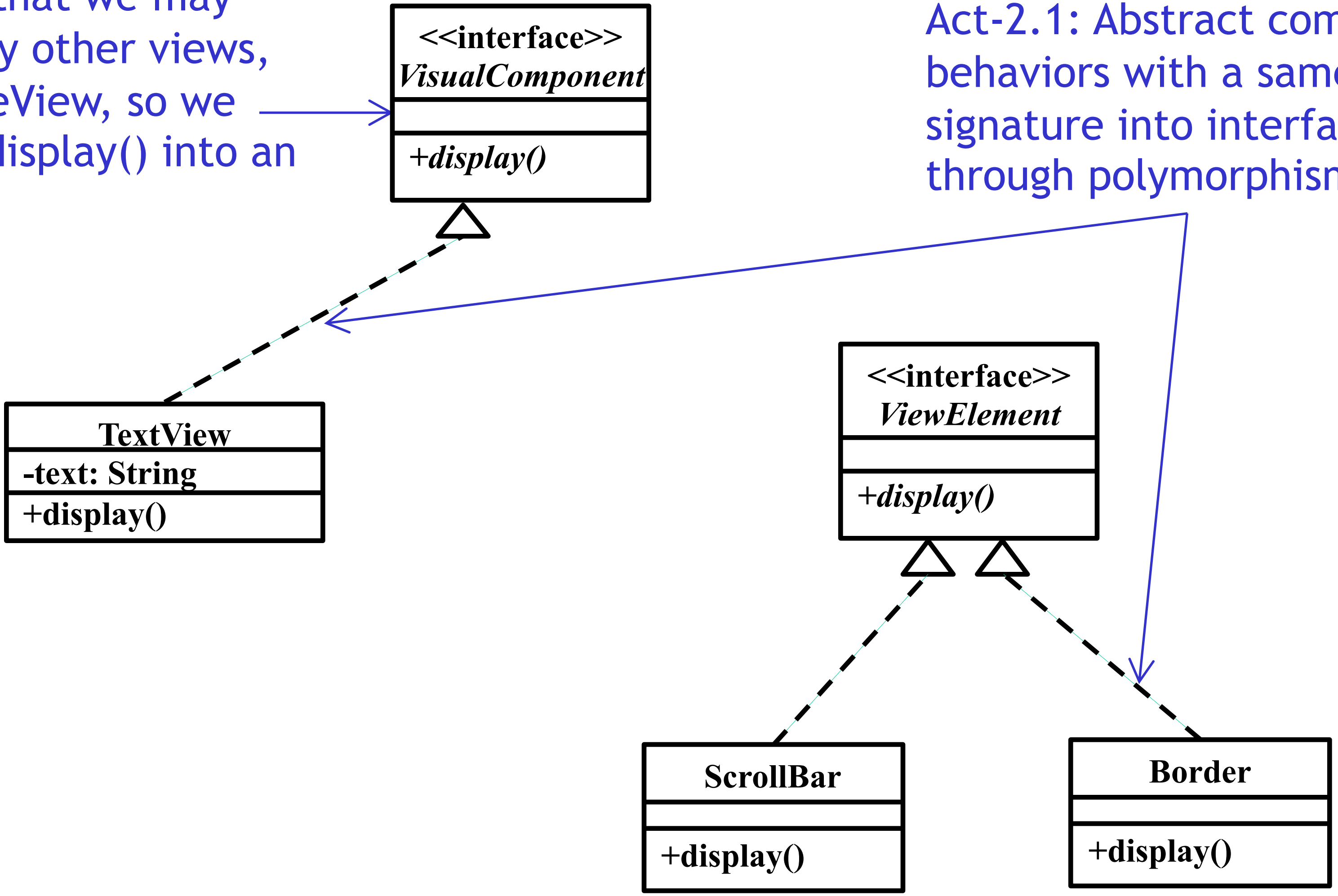
Need composition?

Act-1: Encapsulate What Varies

Act-1.1: Encapsulate an attribute into a concrete class



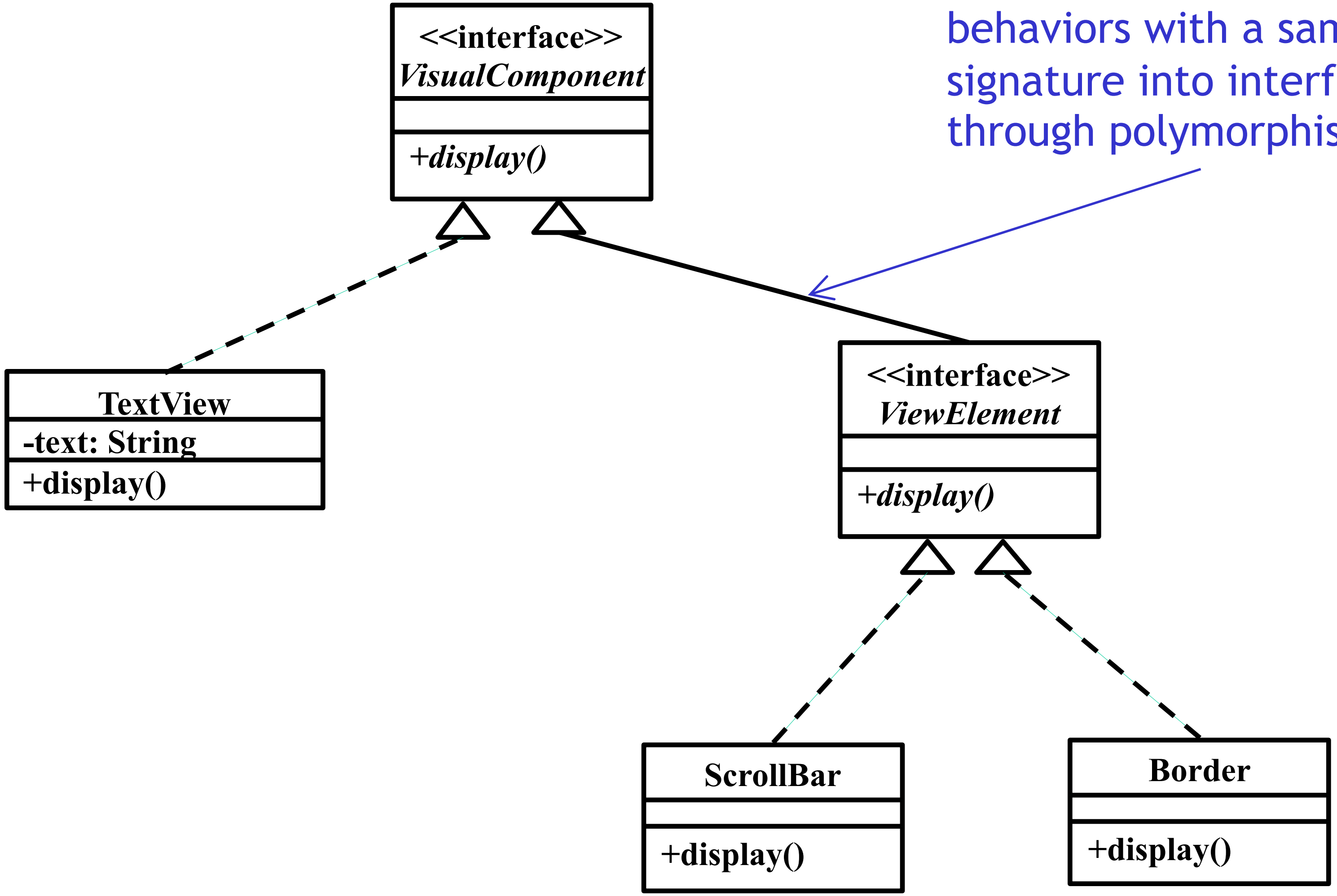
Act-2: Abstract Common Behaviors into Interfaces/Abstract Classes



Consider that we may have many other views, like ImageView, so we abstract display() into an interface.

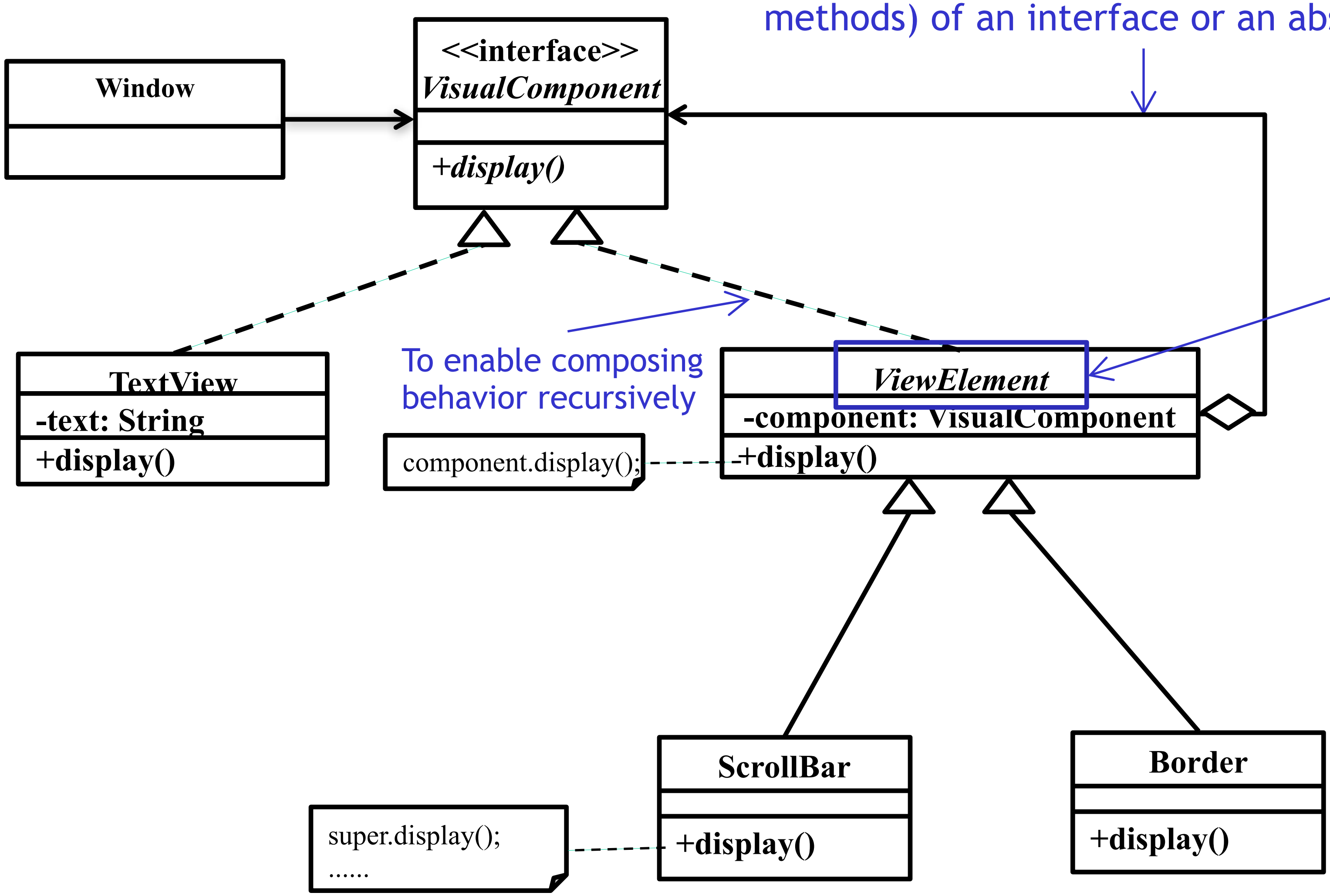
Act-2.1: Abstract common behaviors with a same signature into interface through polymorphism

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

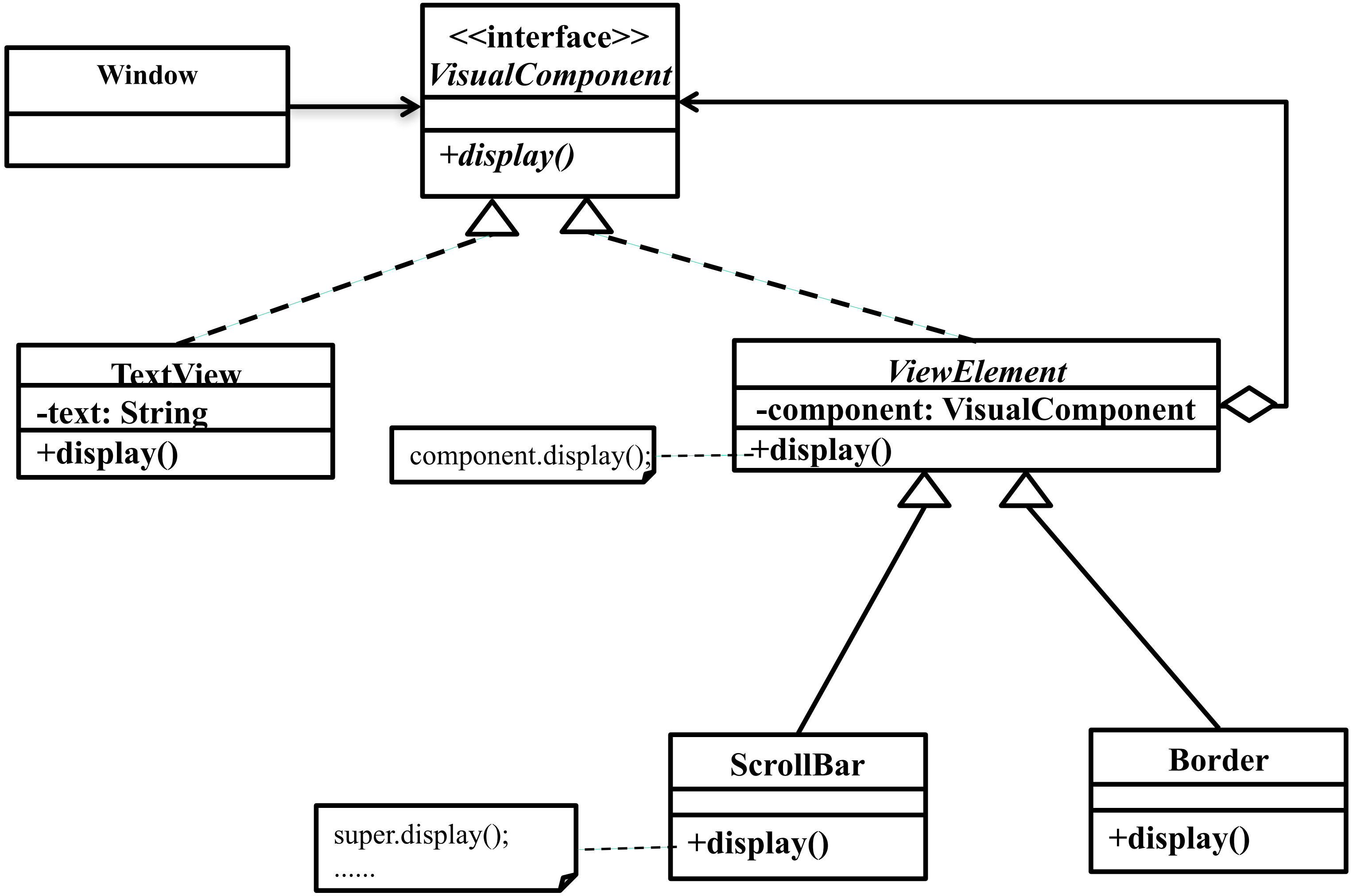


Act-3.1: Compose behaviors (multiple methods) of an interface or an abstract class

To enable composing behavior recursively

The interface is changed into an abstract class in order to keep the composition relation with the visual component attribute (hold reference to)

Refactored Design after Design Process



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$_2$

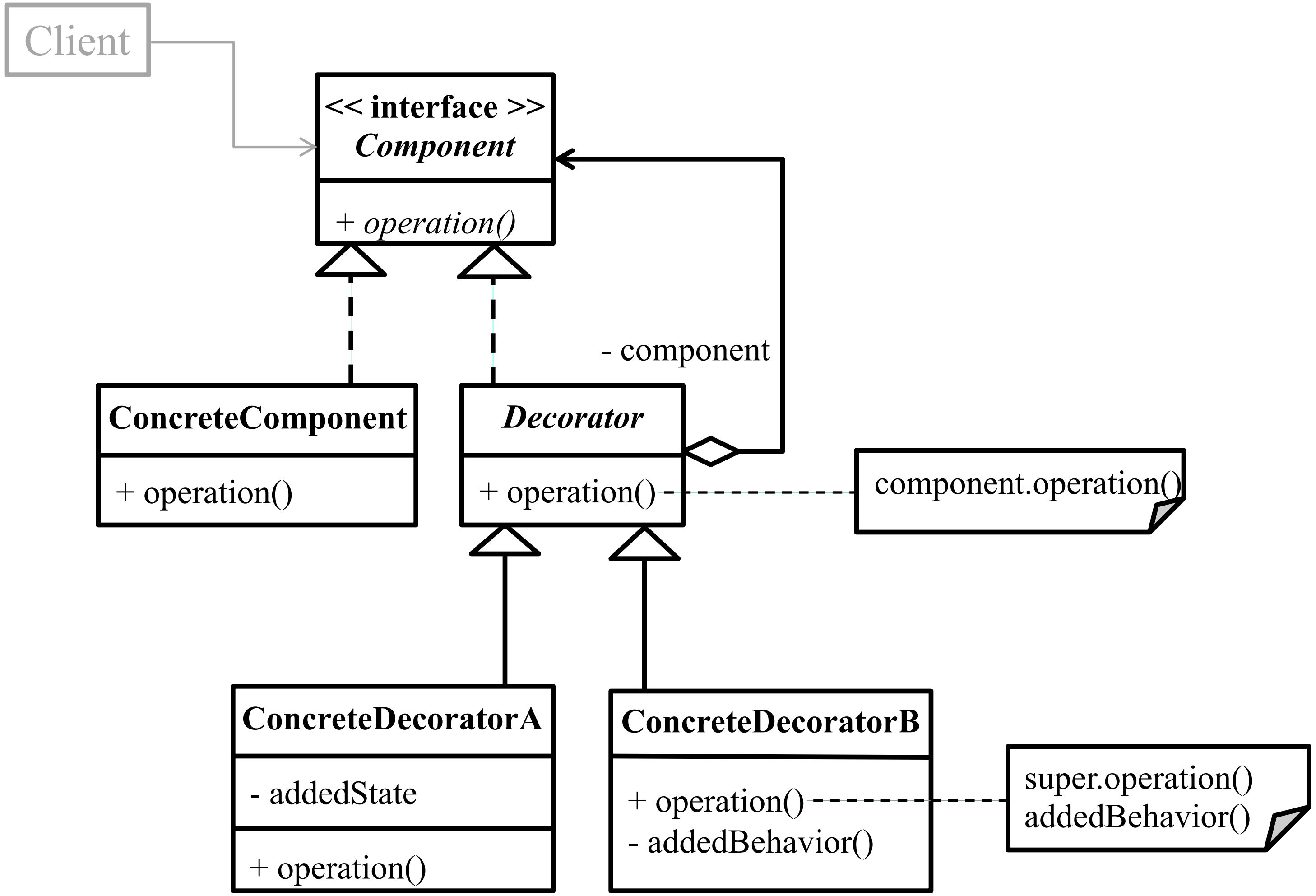
- One way to add responsibilities is with inheritance. Inheriting a border from another class puts a border around every subclass instance.
- This is inflexible, however, because the choice of border is made statically.
- A client can't control how and when to decorate the component with a border.

Intent

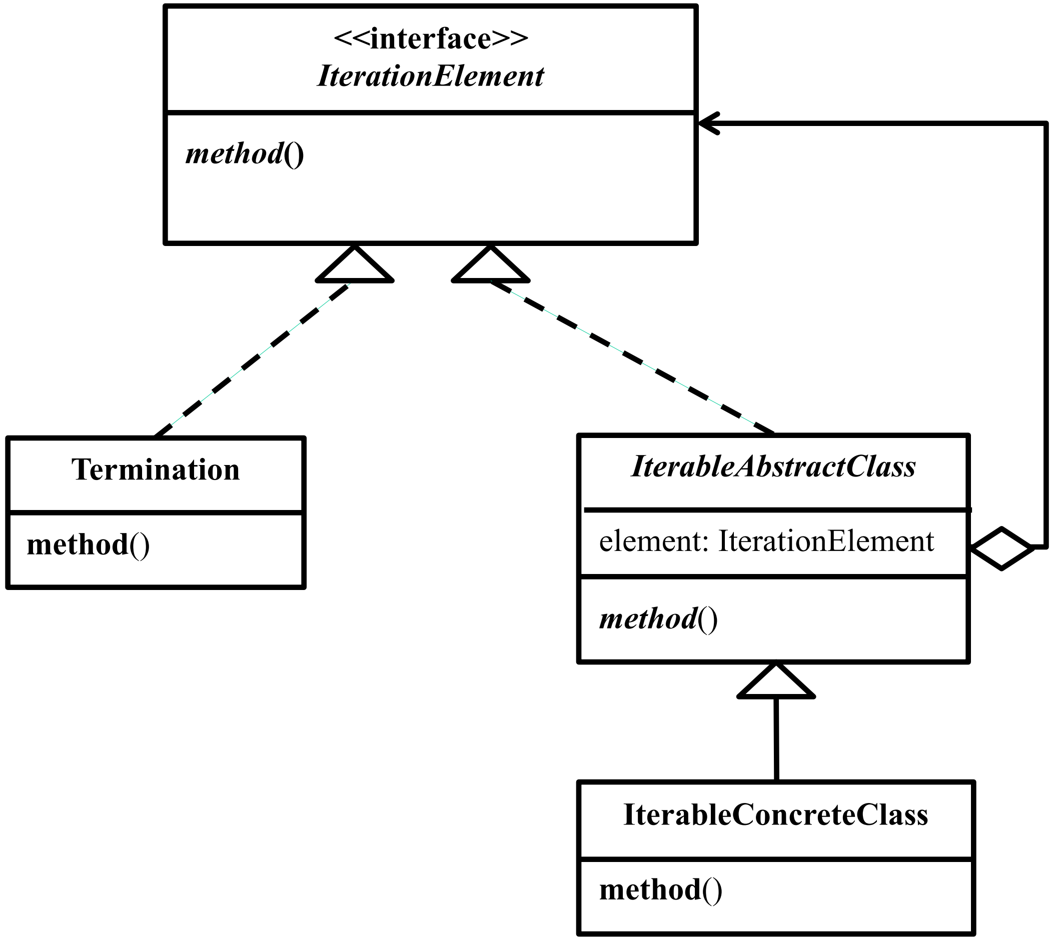
- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

- Extending responsibilities via subclassing forces developers to consider that a new class would have to be made for every possible combination. By contrast, decorators are objects, created at runtime, and can be combined on a per-use basis.

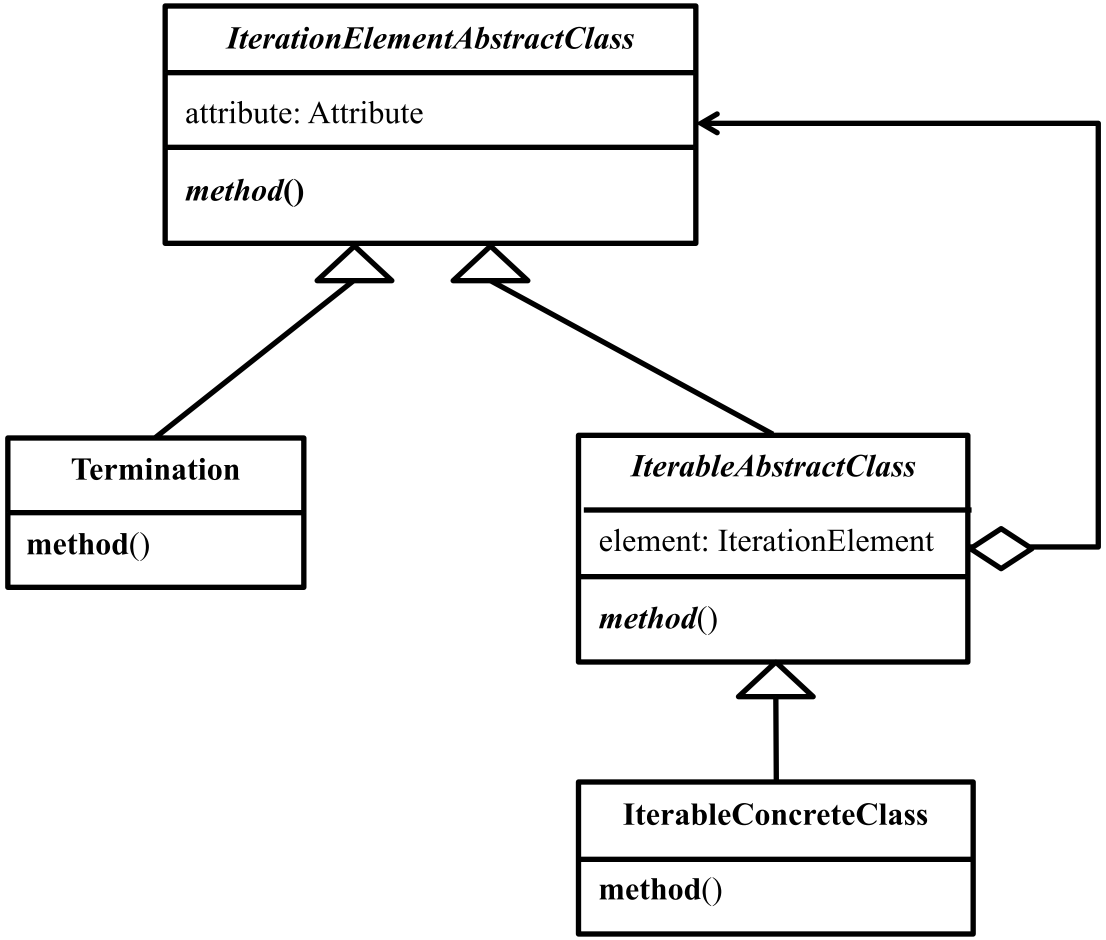
Decorator Pattern Structure$_1$



Recursive Design$_1$



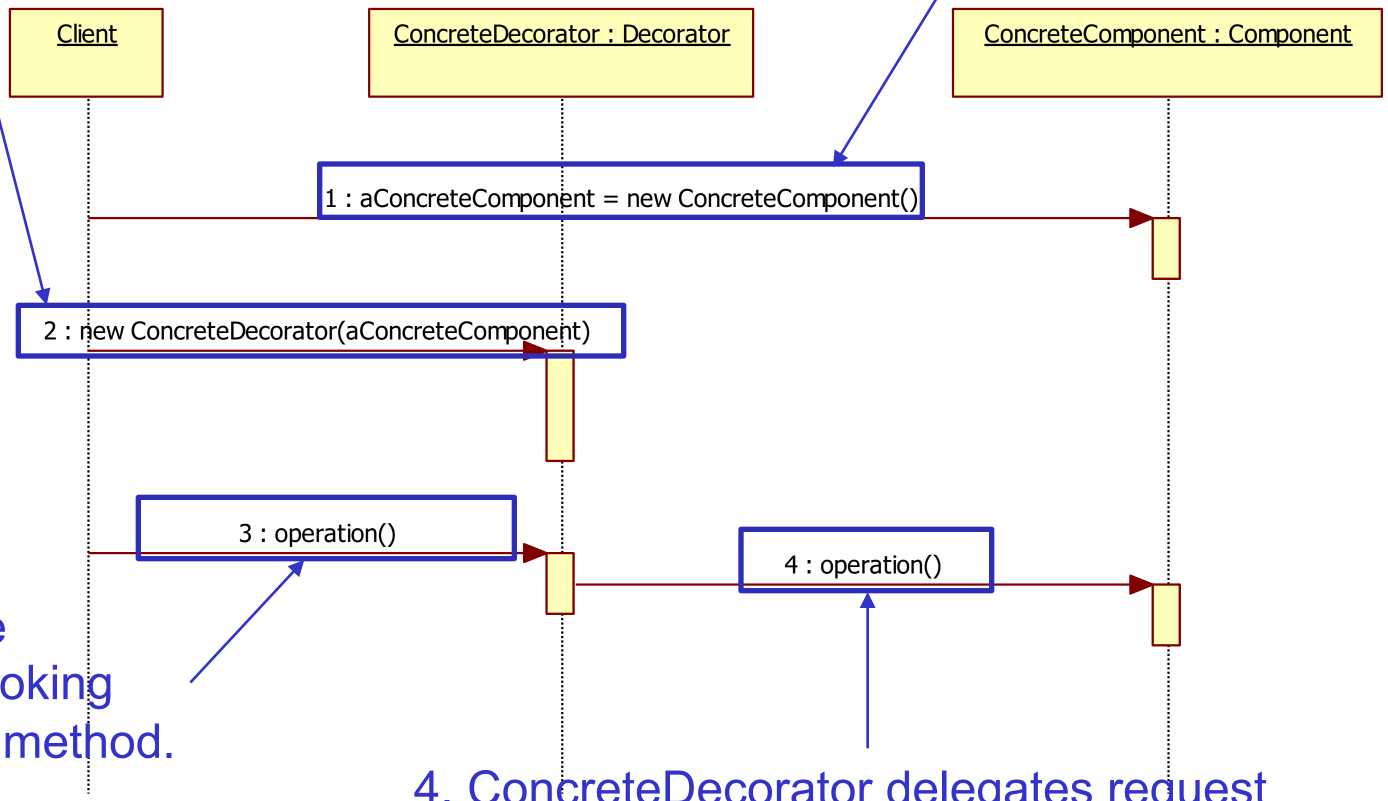
Recursive Design$_2$



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

Software Engineering Lab

selab.csie.ntu.edu.tw

NTU Coffee Shop (Decorator)

Prof. Jonathan Lee (李允中)

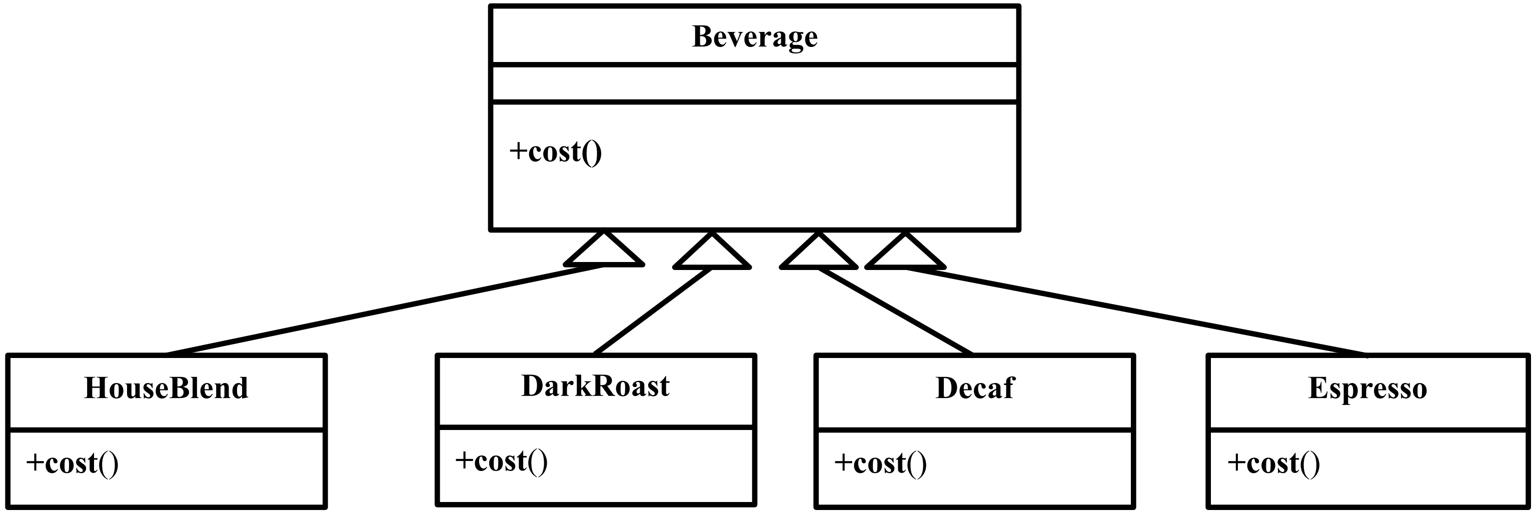
Department of Computer Science and Information Engineering
National Taiwan University

Requirements Statements

- NTU Coffee Shop
  - NTU Coffee shop is scrambling to update its ordering systems to match its beverage offerings (e.g. HouseBlend, DarkRoast, Decaf and Espresso) to calculate how they cost.
  - In addition to your coffee, you can also ask for several condiments like steamed milk, soy, and mocha. Therefore, NTU Coffee Shop needs to have them built into its ordering system to summate the cost.

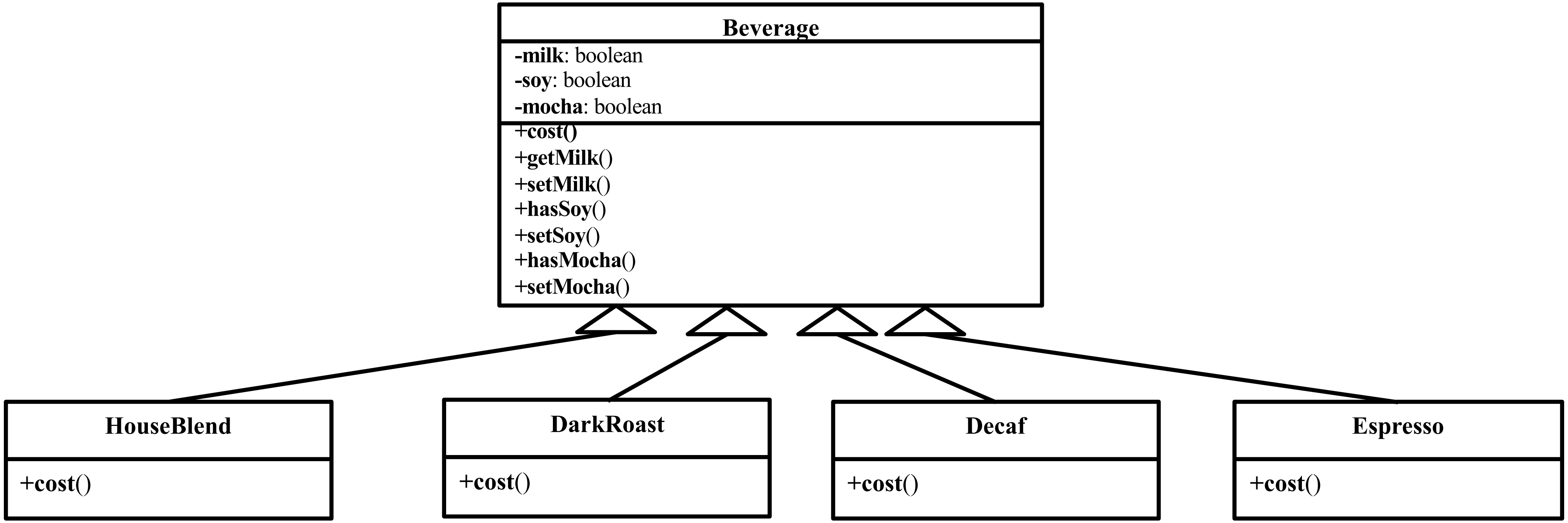
Requirements Statements$_1$

- NTU Coffee
  - NTU Coffee shop is scrambling to update its ordering systems to match its beverage offerings (e.g. HouseBlend, DarkRoast, Decaf and Espresso) to calculate how they cost.

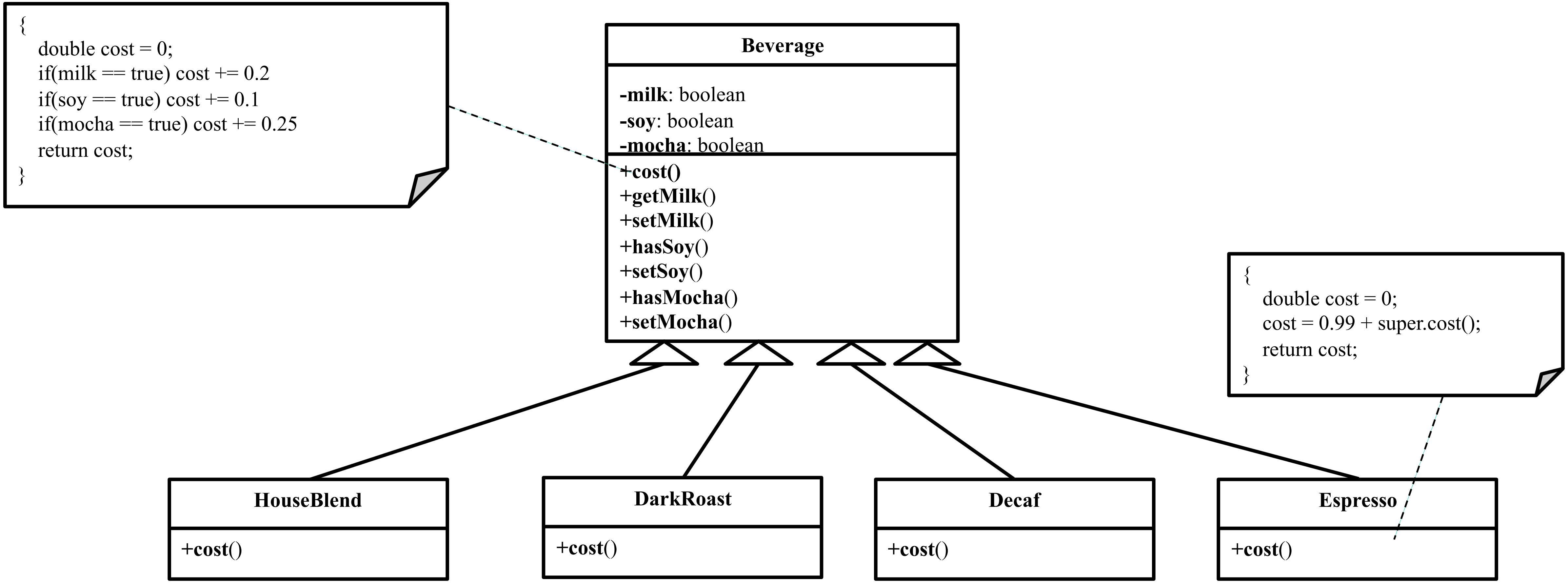


Requirements Statements$_2$

- In addition to your coffee, you can also ask for several condiments like steamed milk, soy, and mocha. Therefore, NTU Coffee Shop needs to have them built into its ordering system to summate the cost.



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
   return cost;
}
```

Beverage

-milk: boolean
-soy: boolean
-mocha: boolean

+cost()
+**getMilk**()
+**setMilk**()
+**hasSoy**()
+**setSoy**()
+**hasMocha**()
+**setMocha**()

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

HouseBlend

+**cost**()

DarkRoast

+**cost**()

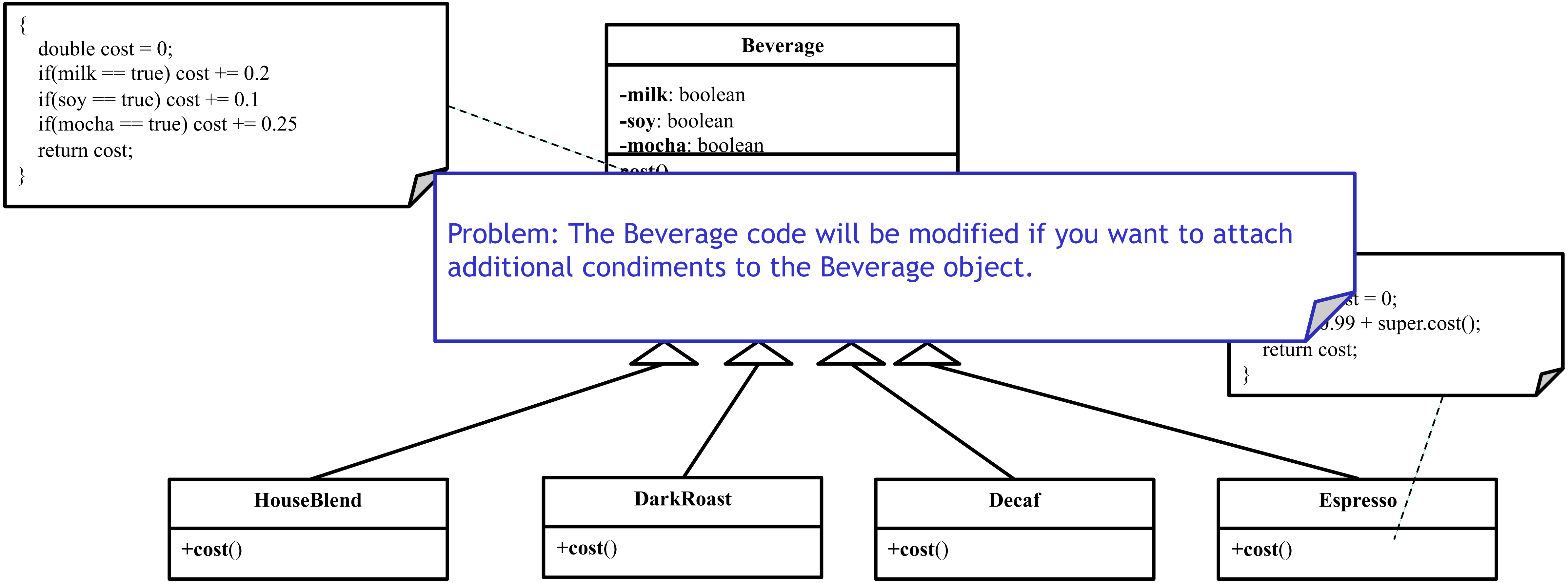
Decaf

+**cost**()

Espresso

+**cost**()

Problems with the Initial Design

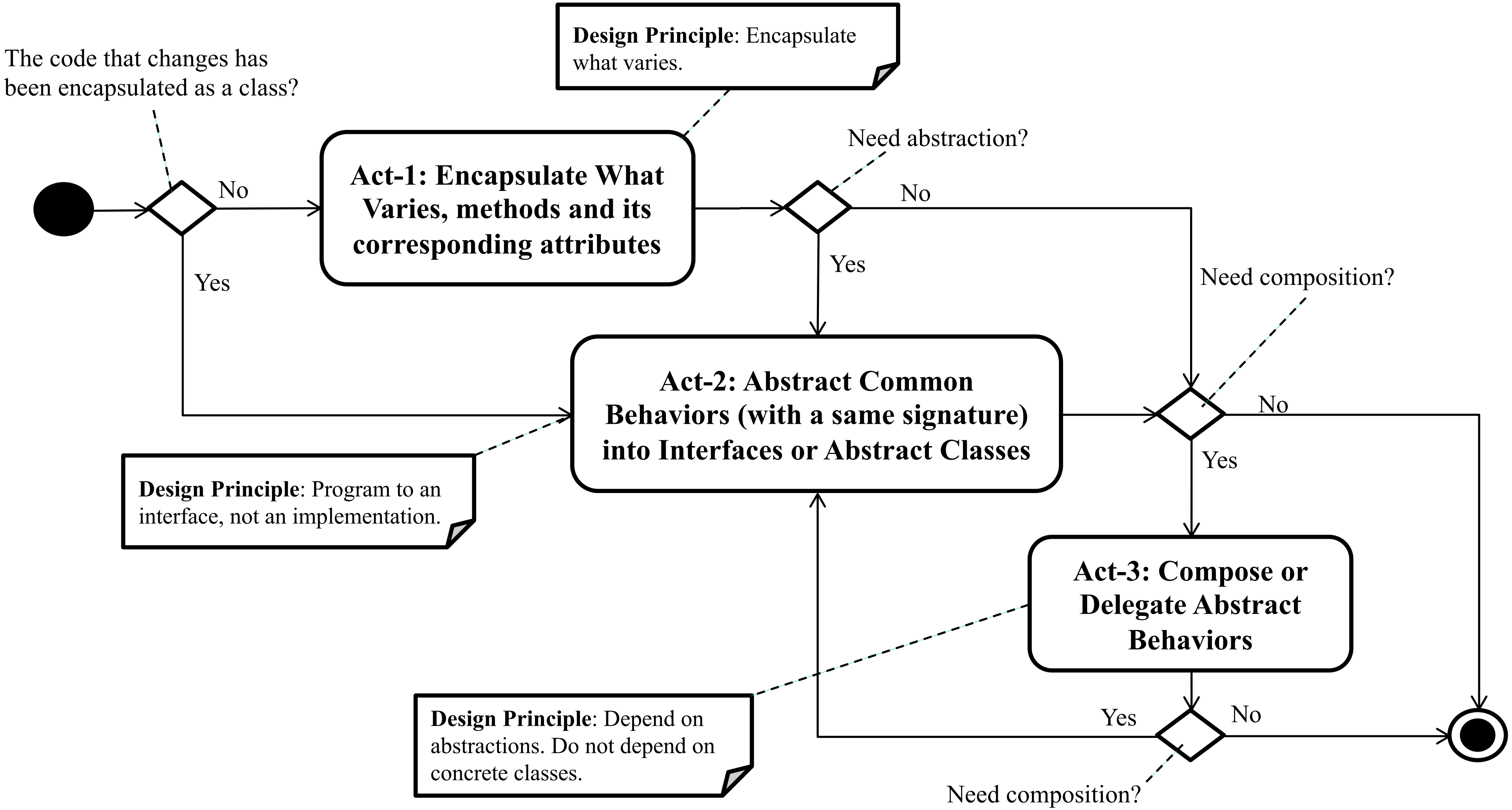


```
{
    double cost = 0;
    if(milk == true) cost += 0.2
    if(soy == true) cost += 0.1
    if(mocha == true) cost += 0.25
    return cost;
}
```

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

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

Design Process for Change



The code that changes has been encapsulated as a class?

Design Principle: Encapsulate what varies.

Act-1: Encapsulate What Varies, methods and its corresponding attributes

Need abstraction?

Act-2: Abstract Common Behaviors (with a same signature) into Interfaces or Abstract Classes

Design Principle: Program to an interface, not an implementation.

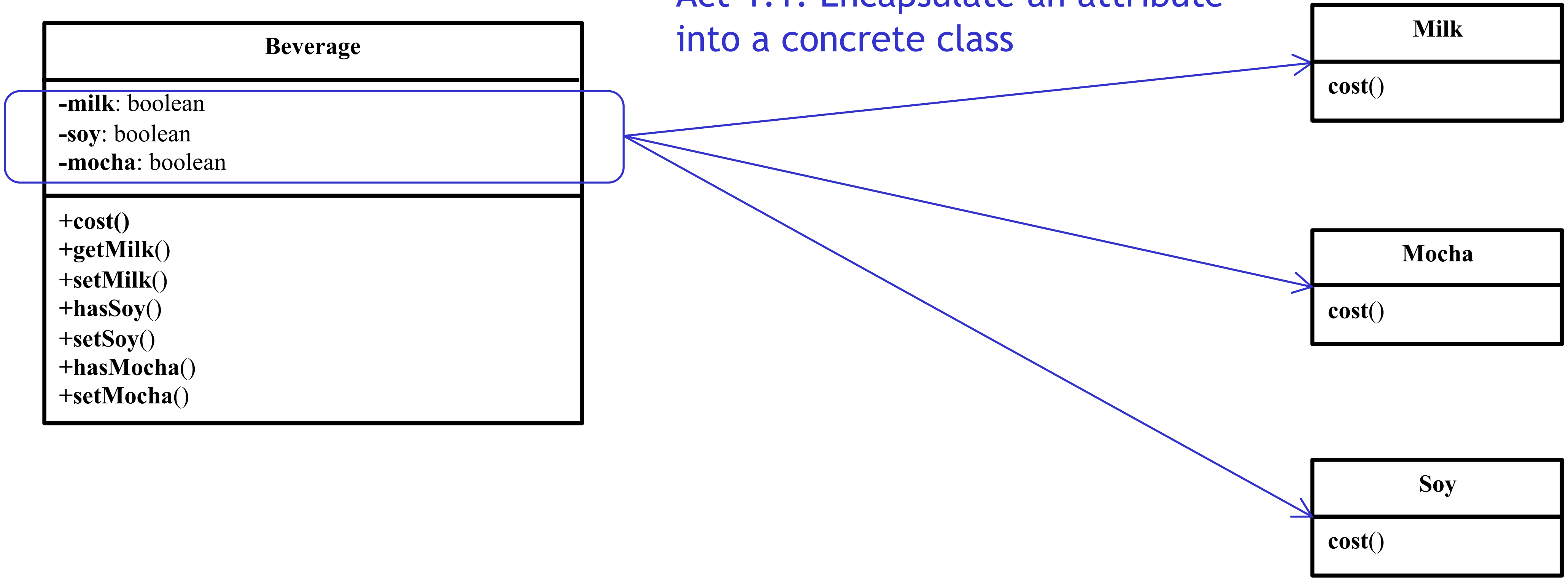
Need composition?

Act-3: Compose or Delegate Abstract Behaviors

Design Principle: Depend on abstractions. Do not depend on concrete classes.

Need composition?

Act-1: Encapsulate What Varies



Act-1.1: Encapsulate an attribute into a concrete class

Beverage

-milk: boolean
-soy: boolean
-mocha: boolean

+cost()
+getMilk()
+setMilk()
+hasSoy()
+setSoy()
+hasMocha()
+setMocha()

Milk

cost()

Mocha

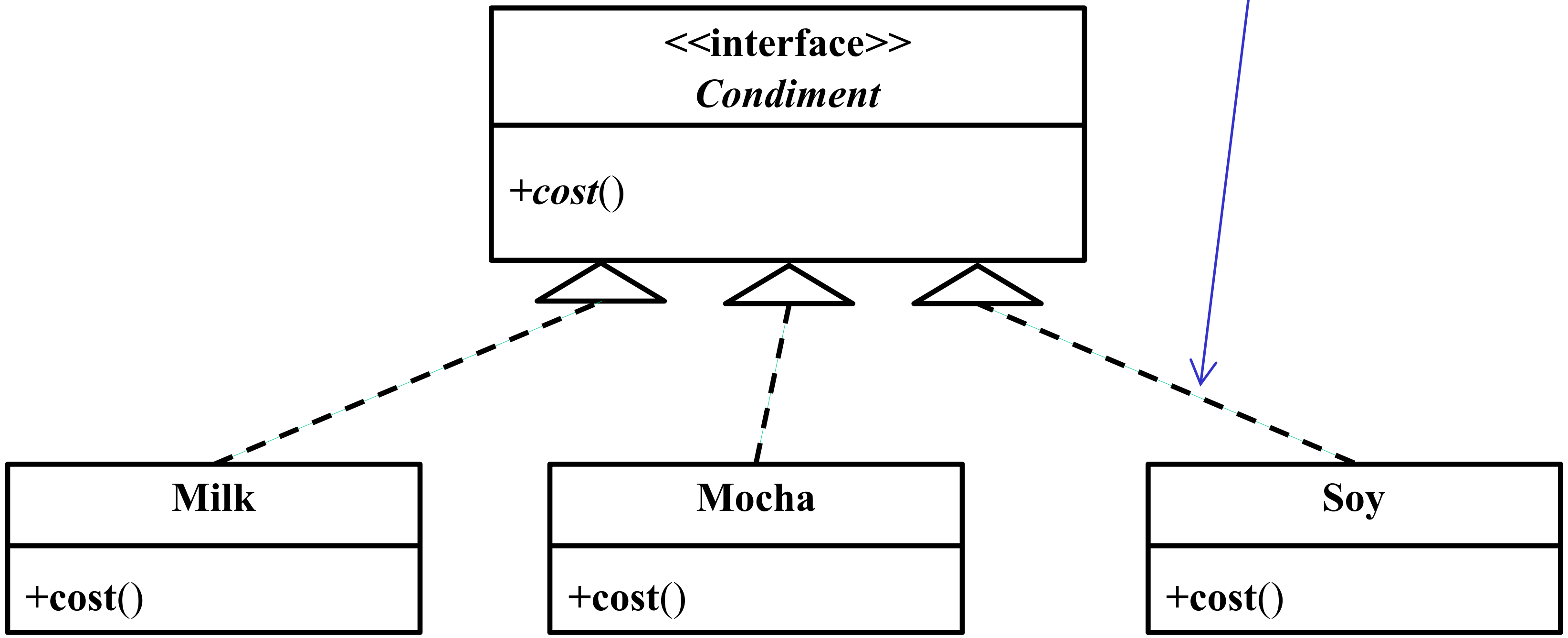
cost()

Soy

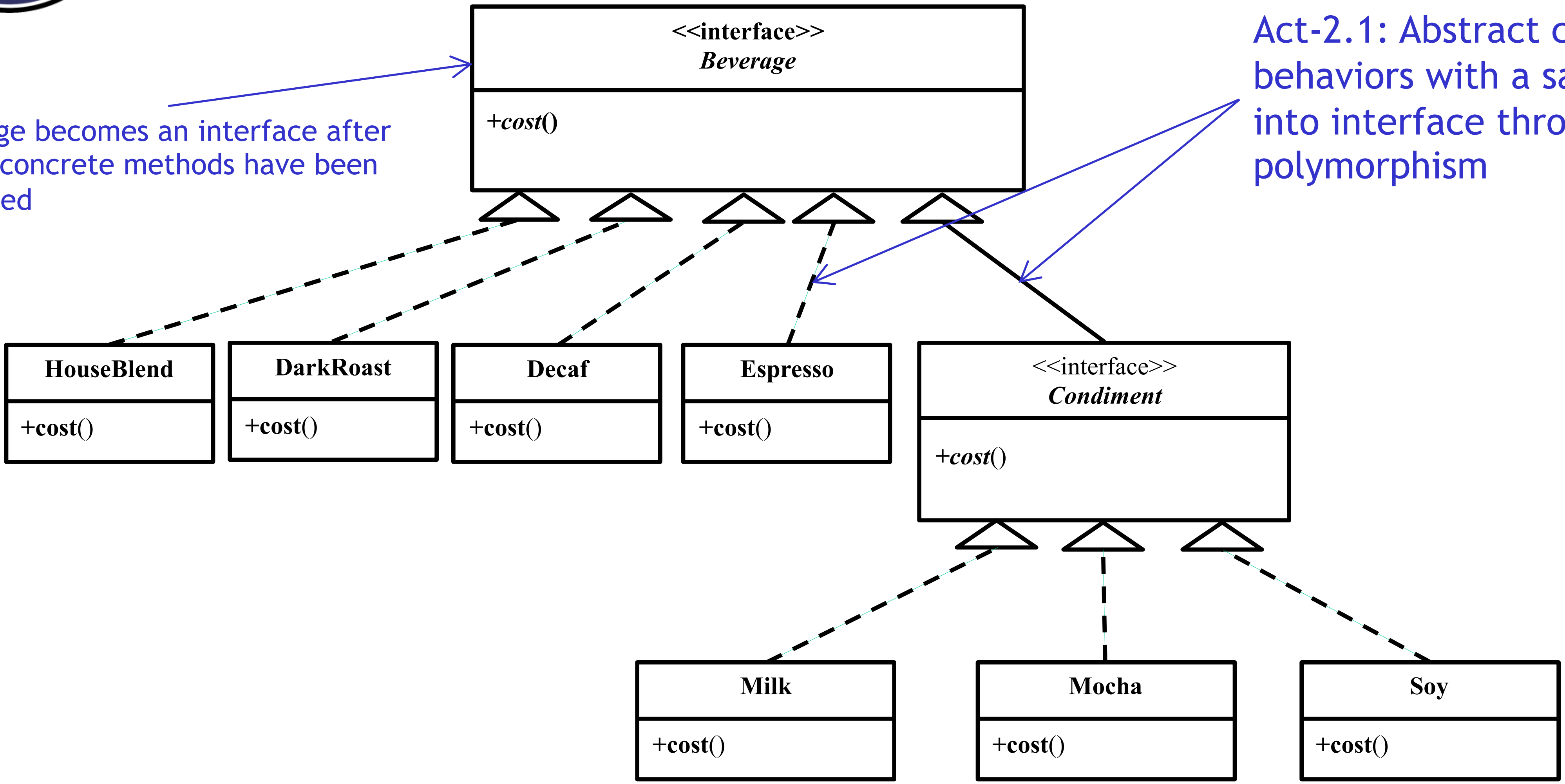
cost()

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

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



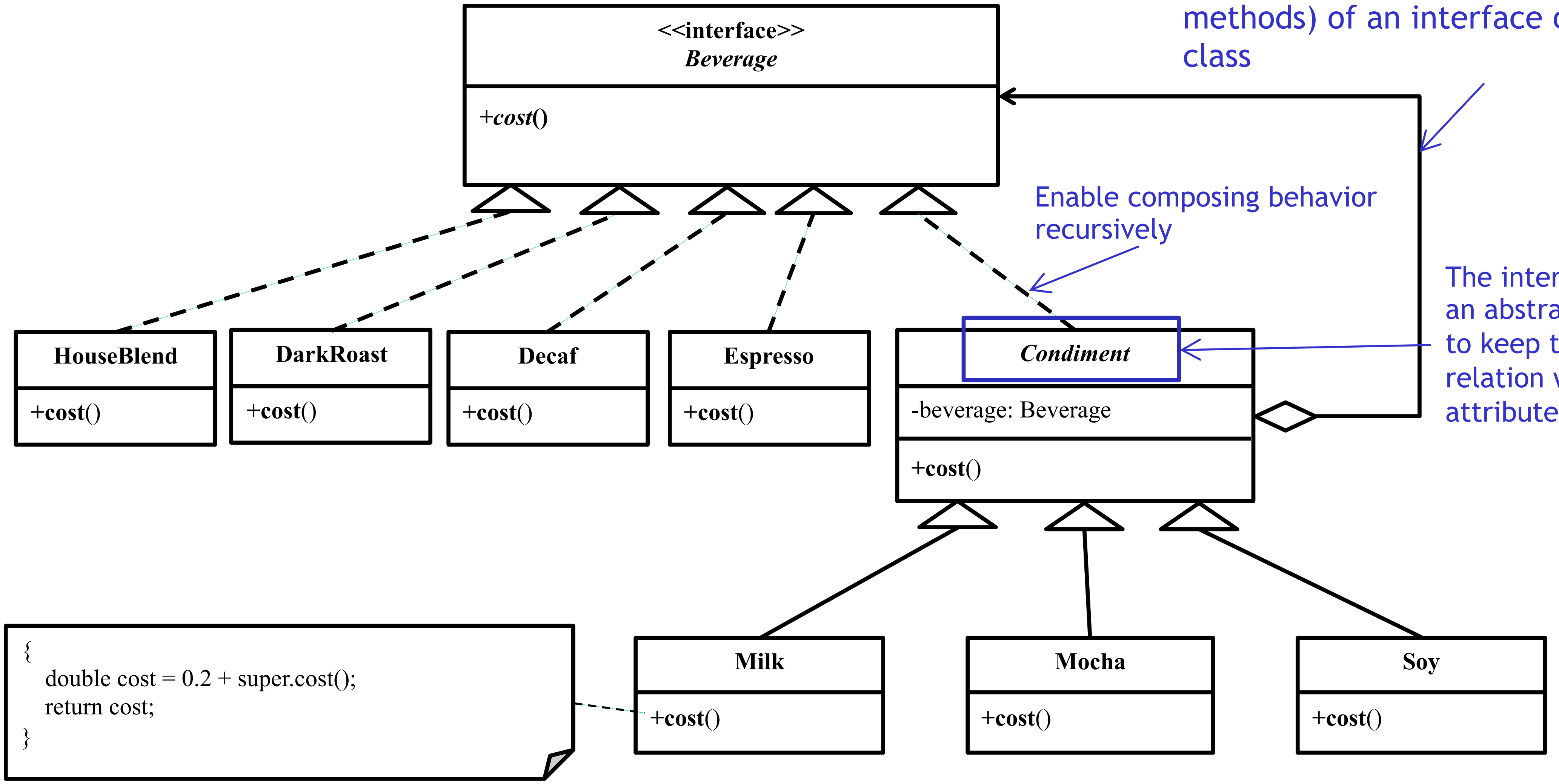
Act-2: Abstract Common Behaviors into Interfaces/Abstract Classes



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



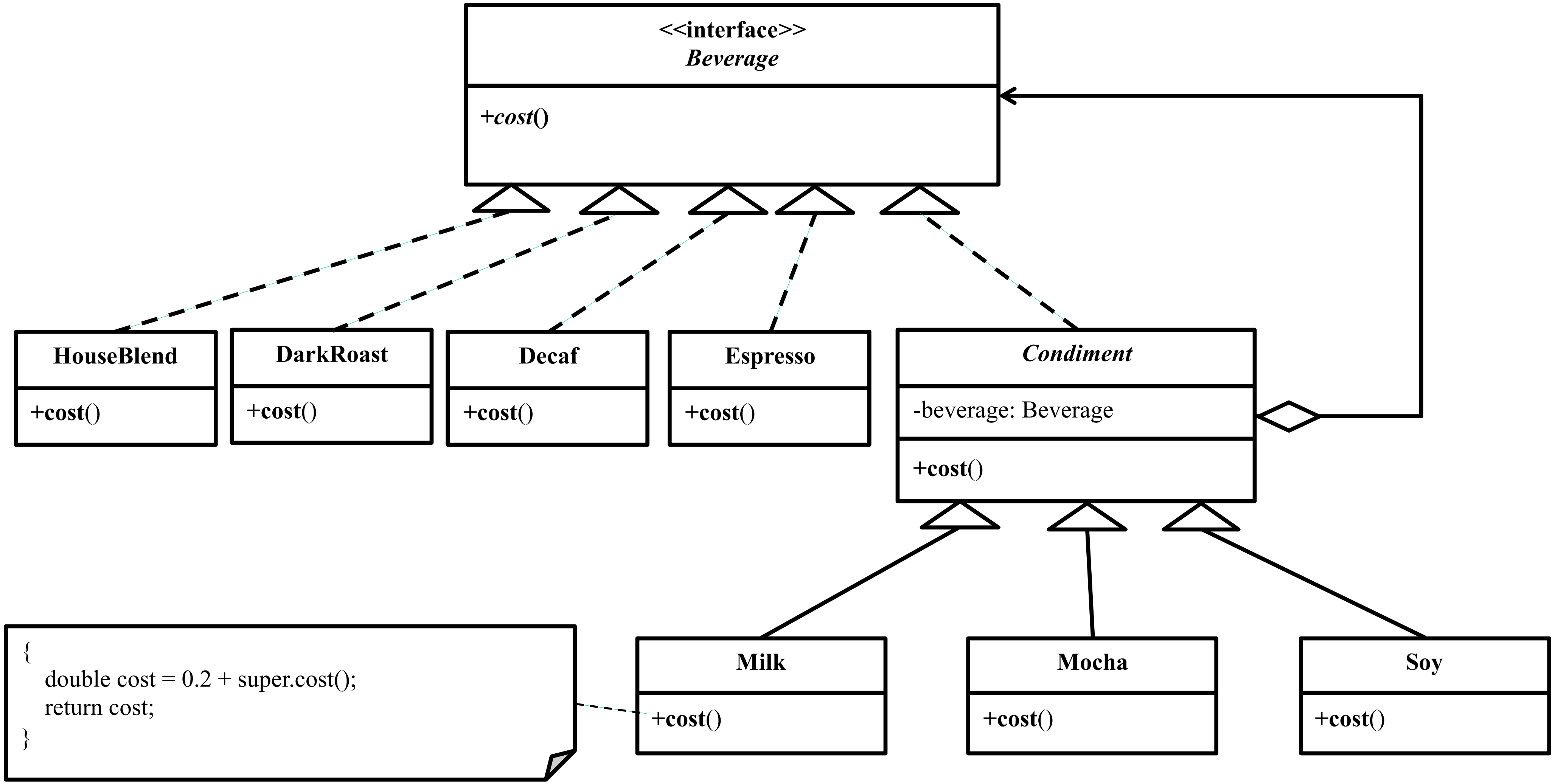
```
{
   double cost = 0.2 + super.cost();
   return cost;
}
```

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

Enable composing behavior recursively

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

Refactored Design after Design Process

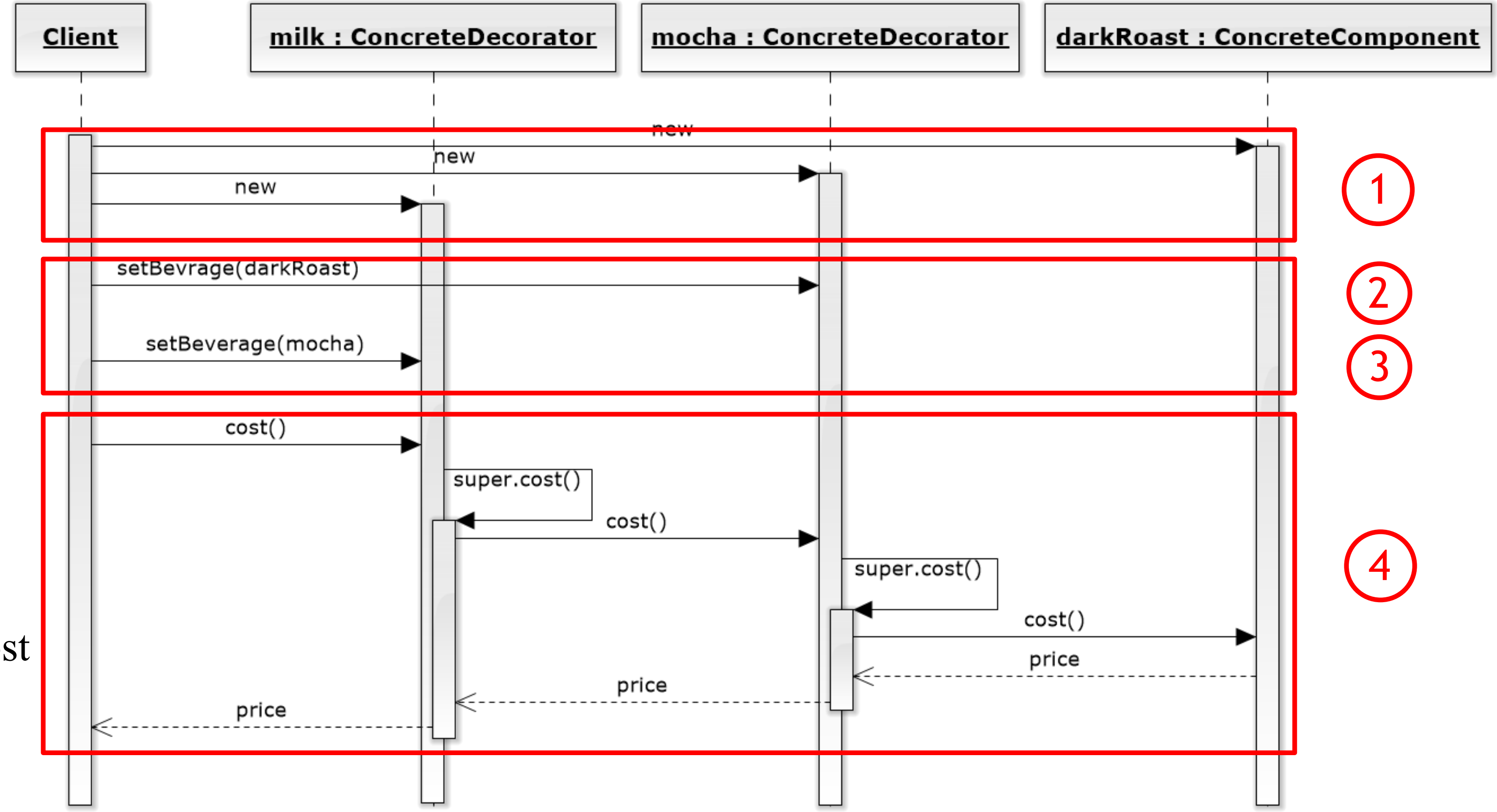


```
{
   double cost = 0.2 + super.cost();
   return cost;
}
```

Sequence Diagram

1. Client creates three objects: DarkRoast, Mocha, and Milk.
2. Client decorates the DarkRoast object with the Mocha object
3. Client decorates the Mocha object with the Milk object

4. Client calculates the cost by invoking the cost() of the top decorator (the Milk object). It will then recursively call the cost methods of all the other components.


Client
milk : ConcreteDecorator
mocha : ConcreteDecorator
darkRoast : ConcreteComponent
new
new
new
setBevrage(darkRoast)
setBeverage(mocha)
cost()
super.cost()
cost()
super.cost()
cost()
price
price
price
1
2
3
4