Decorator Pattern

© Dr. B. Fraser

Image: Rachel Claire on pexels.com

Topics

1) How can we easily modify existing classes with new behaviours?
2) How can we design our code to support changes during maintenance?

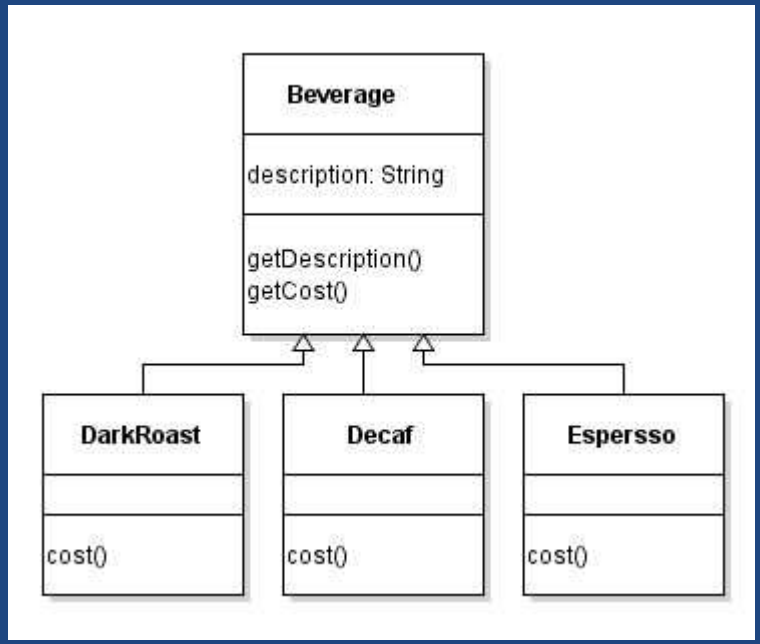
The Coffee Shop

Trying to Add Extras

Image: Chevanon Photography on pexels.com

Base Coffee Shop System

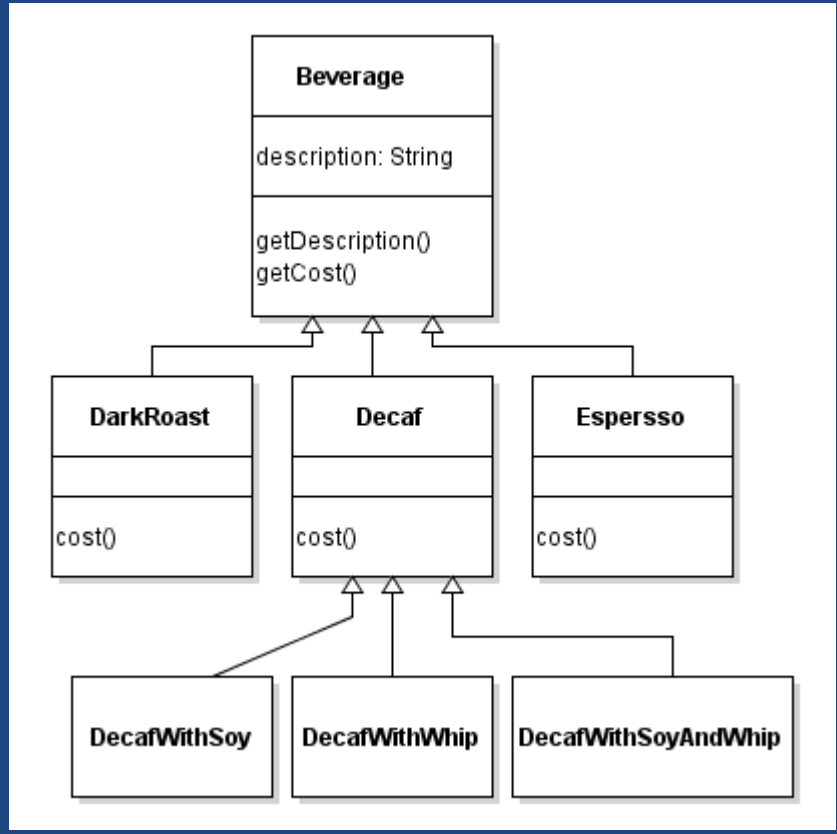
- What does inheritance buy us?
  - ..
    Useful for collections of beverages or a function to operate on any Beverage
- Are separate classes useful?
  - Upfront Recommendation: Don't use inheritance for your entities, but rather your policies.



- Our work today is to add *extras* to our drinks:
  Decaf with Mocha and Soy

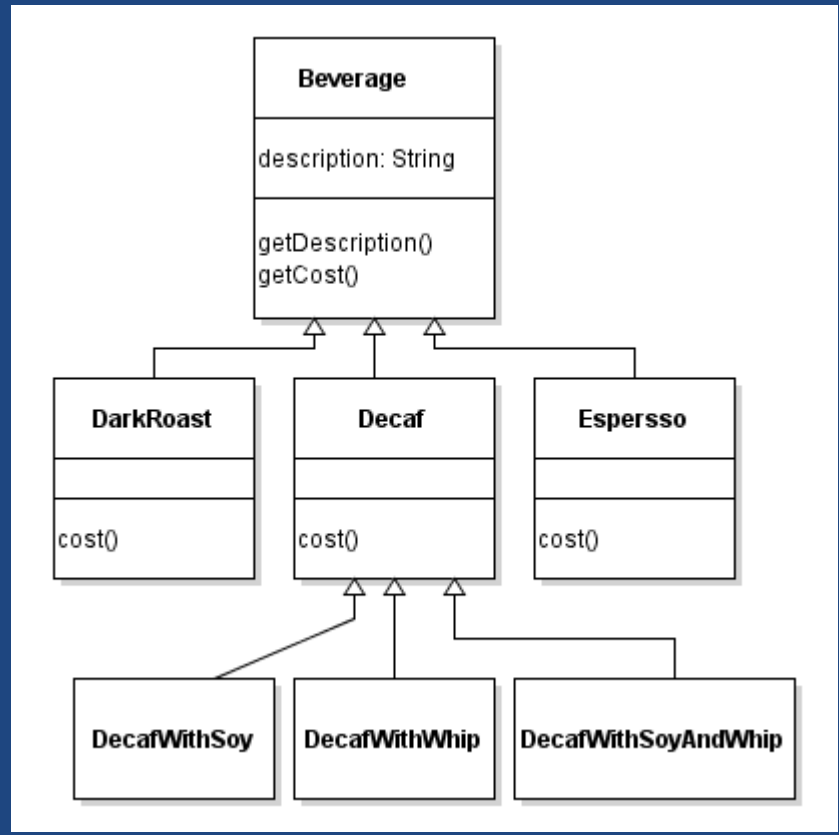
Try 1: Derived Classes

- Each extra will:
  - Modify drink cost
  - Modify drink description
- Discussion
  What happens when adding:
  - New Caramel Extra
  - New HotChocolate



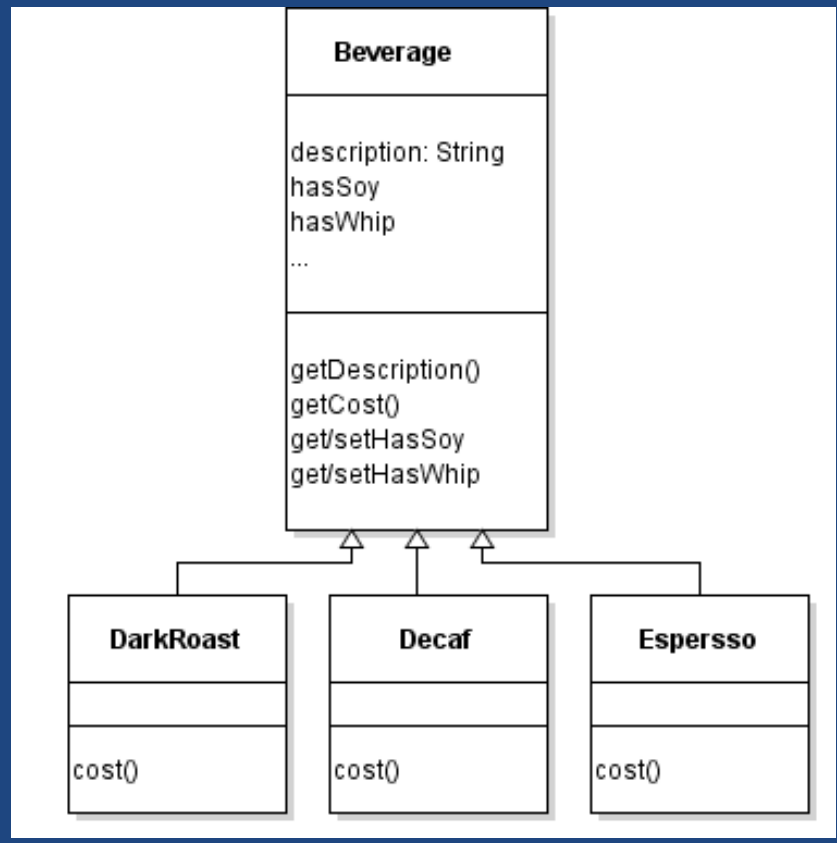
Try 1: Derived Classes (cont)

- Critique what is wrong with the OOD using software design principles
- Violates OOD principles:
  - encapsulate what changes
  - favour composition over inheritance
  - Don’t Repeat Yourself (multiple classes for whip)
- Cannot change an object’s type at runtime
  - Cannot add Whip to an already instantiated beverage



Try 2: Extras in Base Class

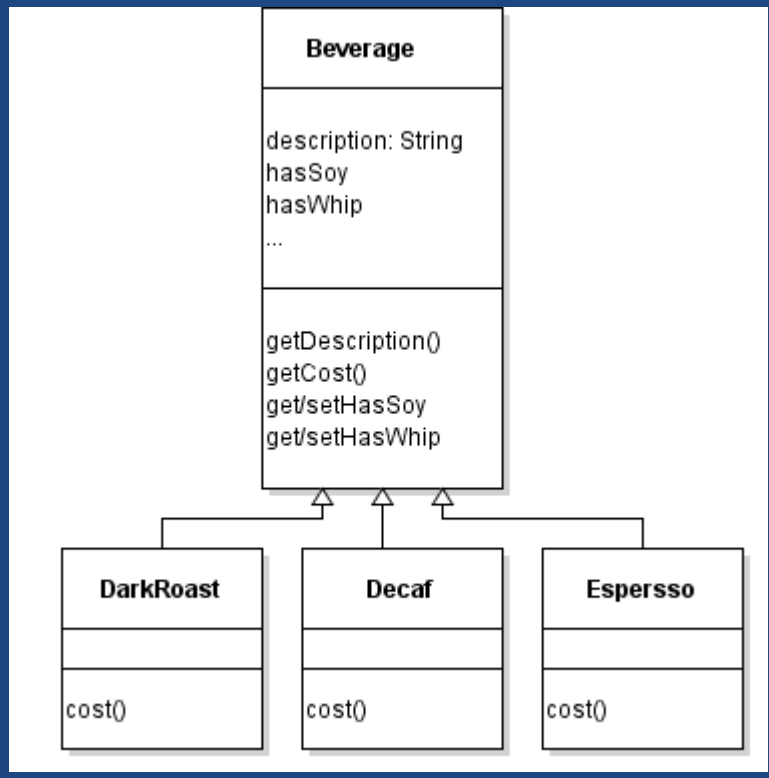
- Put the extras in the base class
  - Base class computes cost of extras
  - Derived classes provide cost of plain drink
- What's better about this OOD?
  - No class explosion!
  - Can dynamically add/remove extras



Try 2: Extras in Base Class (cont)

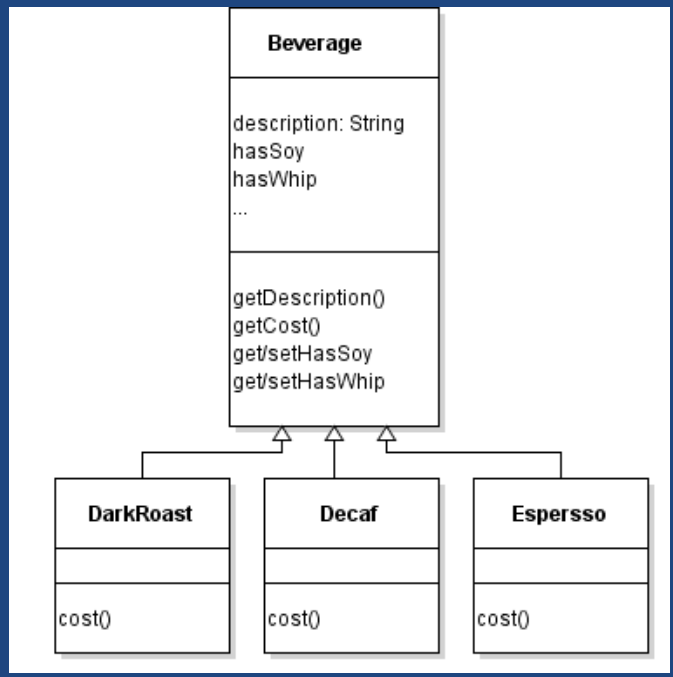
- Problems
  - To add a new extra requires
    ..
    Likely to introduce bugs and non-local changes (derived classes)
  - Inherited behaviour (extras) may not make sense in some drinks

    Ex: ice-tea with whip cream?
  - Creating a double-mocha
- Ideas to use to enhance our OOD
  - ..



Open-Closed Principle

- Design Principle:
  Open-Closed Principle
  - Classes should be
    open for..
    closed for..
  - Ex: adding a new extra should not require re-coding existing classes
- New requirements should result in new classes, not changing existing code at the root of the inheritance tree



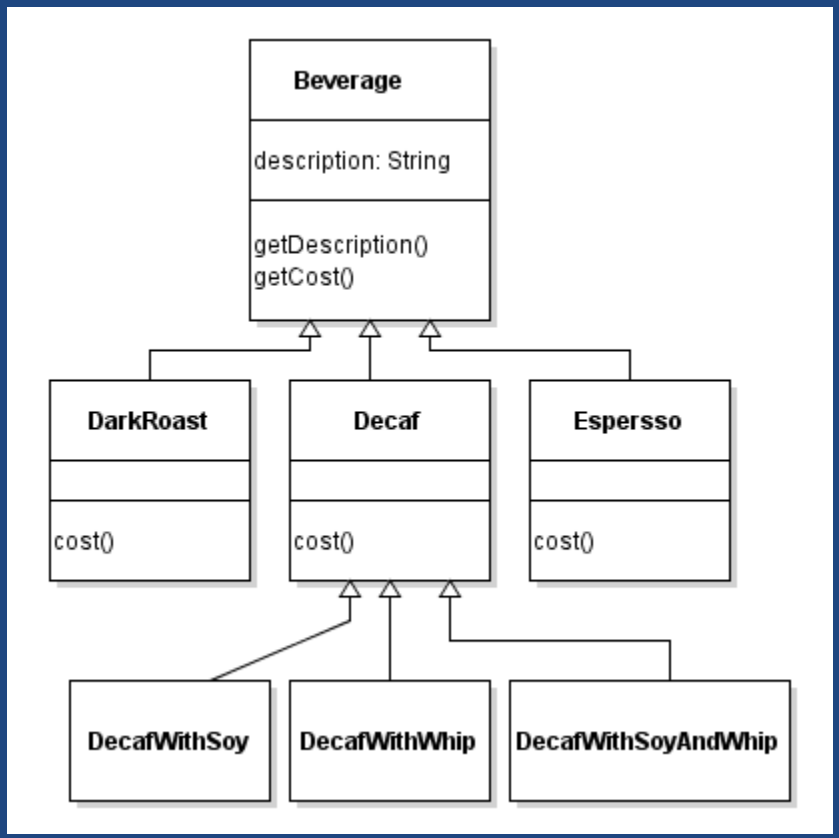
Adds a level of abstraction (complexity);
use in areas expected to change

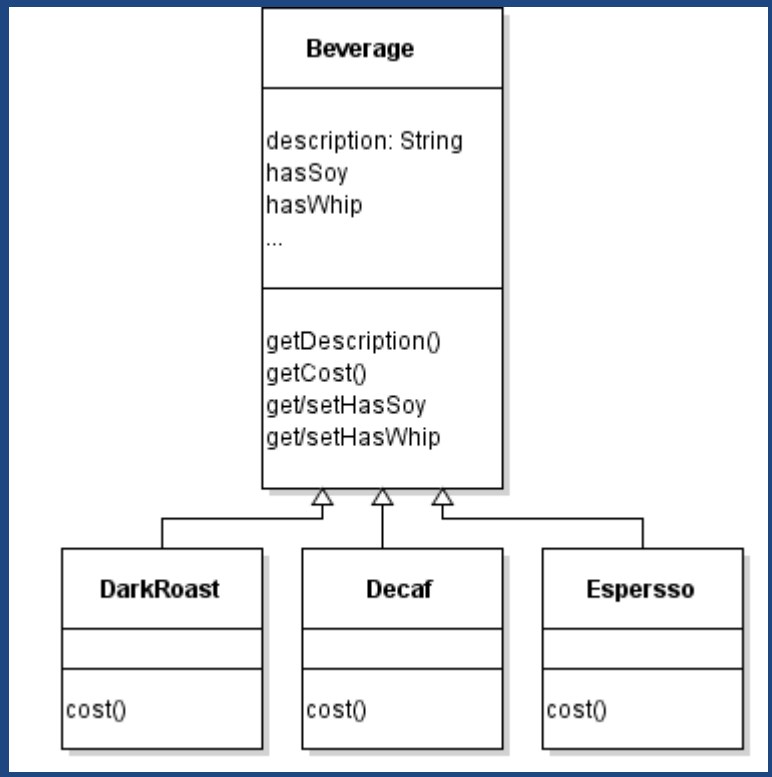
Decorators
to add Extras

Image: https://chercher.tech/kotlin/decorator-design-pattern-kotlin

The Ideas So Far

- What was wrong with each of these?

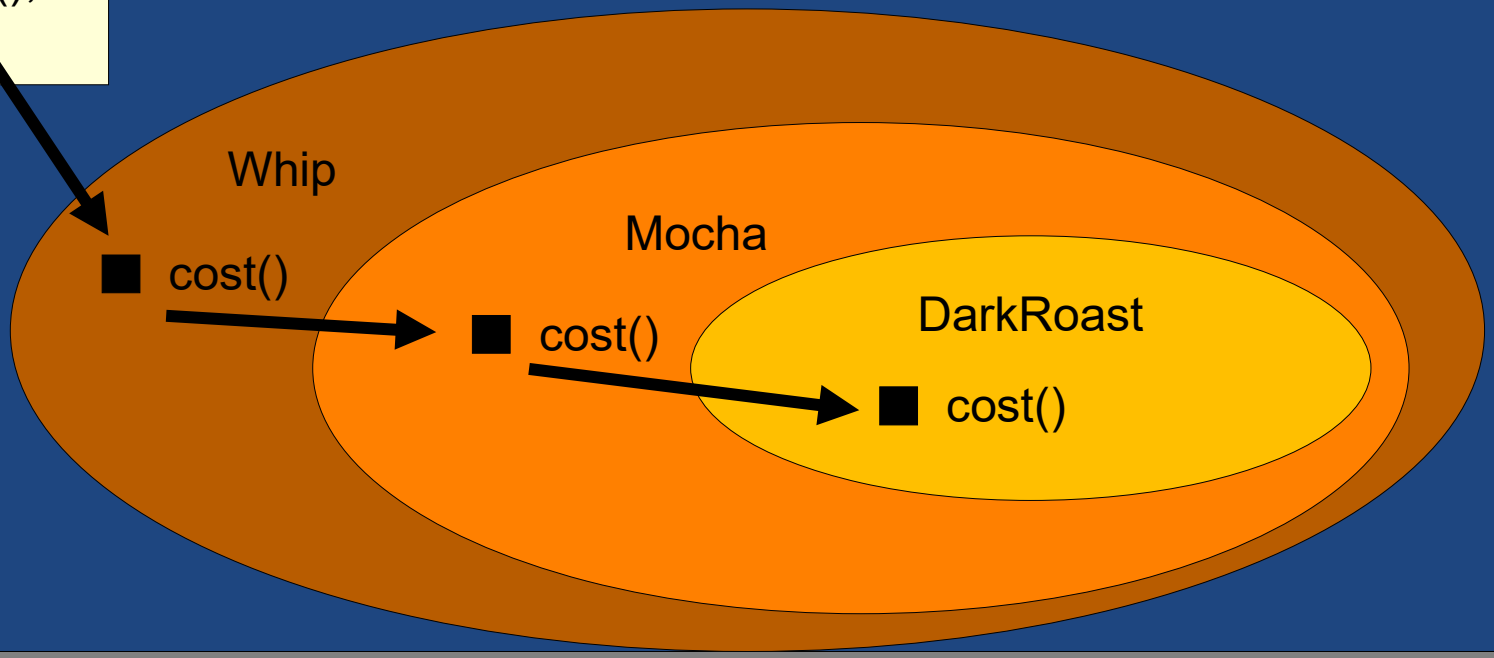




Try 3: Decorator

- An Example:
  Make a Mocha DarkRoast with Whip

```
main() {
        Beverage b = ....
        int total = b.cost();
}
```

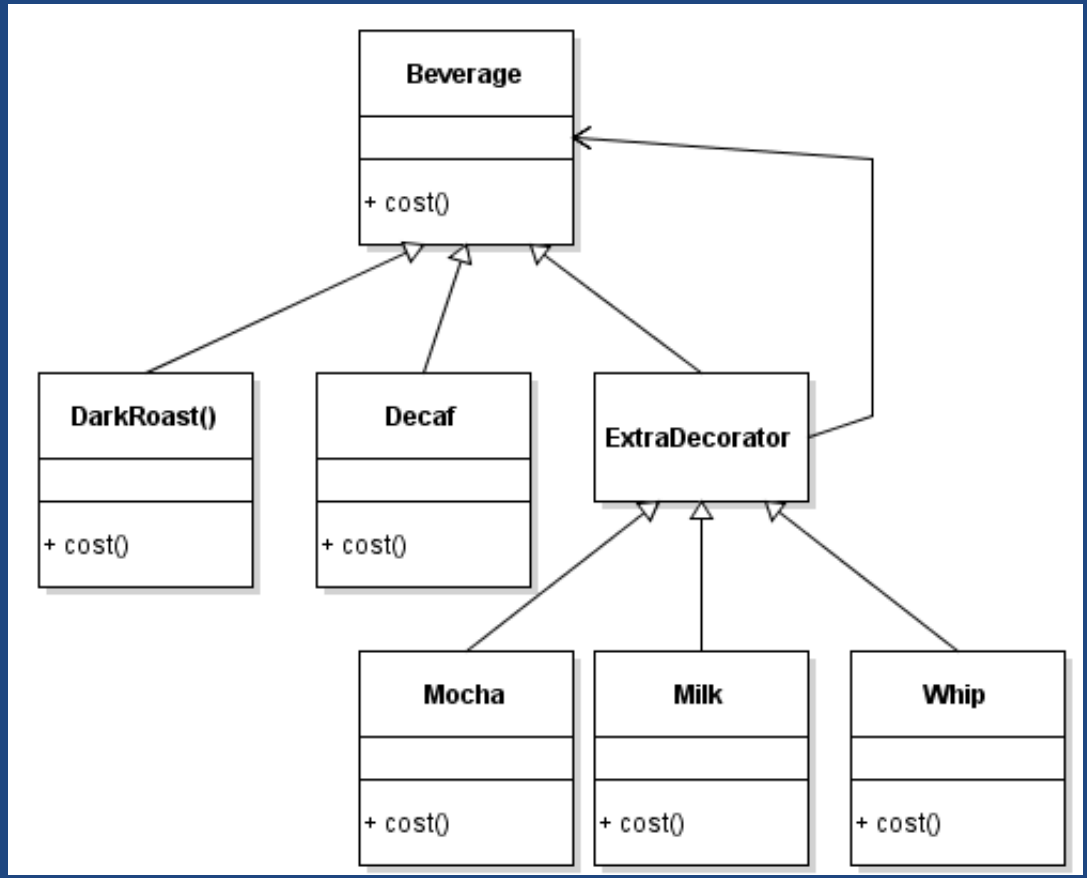


Try 3: Decorator (cont)

Decorator Pattern

- ..

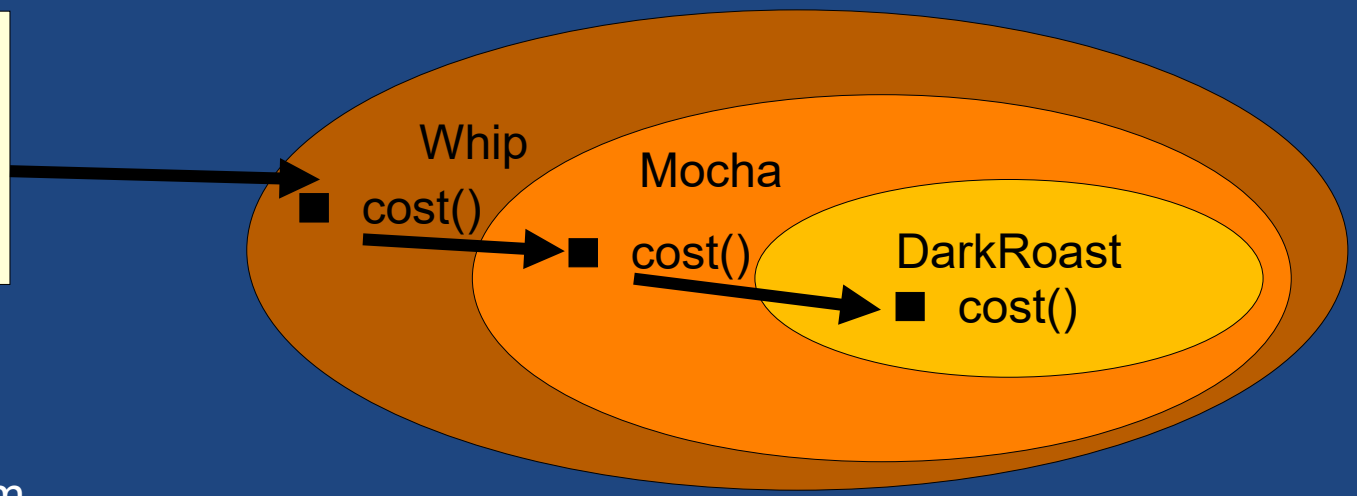
- Decorators provide a flexible alternative to..



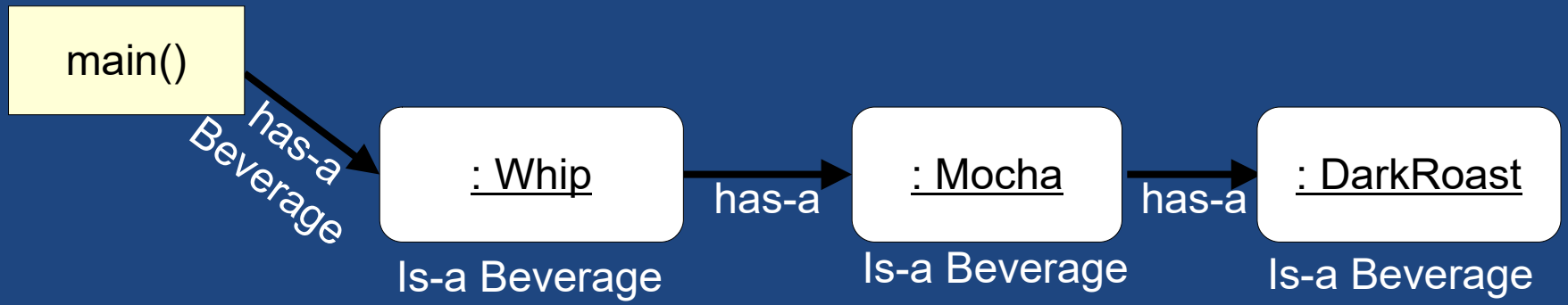
Decorator w/ Mocha DarkRoast with Whip

Decorator Drawing (idea)

```
main() {
        Beverage b = ....
        int total = b.cost();
}
```



Decorator UML Object Diagram



In-class Exercise

- Draw a UML object diagram for:
  Double mocha, soy, decaf

Decorator Code

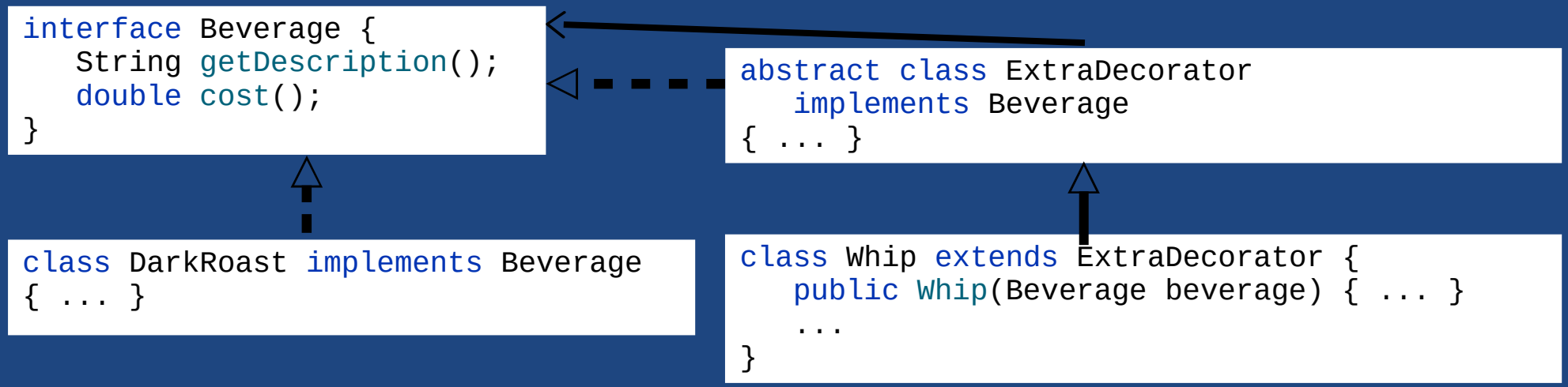
```
interface Beverage {
   String getDescription();
   double cost();
}
```

```
abstract class ExtraDecorator
   implements Beverage
{
   protected Beverage beverage;
   public ExtraDecorator(Beverage beverage) {
      this.beverage = beverage;
   }
}
```

```
class DarkRoast implements Beverage
{
   public String getDescription() {
      return "Dark Roast Coffee";
   }
   public double cost() {
      return .99;
   }
}
```

```
class Whip extends ExtraDecorator {
   public Whip(Beverage beverage) {
      super(beverage);
   }

   public String getDescription() {
      return beverage.getDescription()
             + ", Whip";
   }

   public double cost() {
      return .10 + beverage.cost();
   }
}
```

Decorator Code (client)



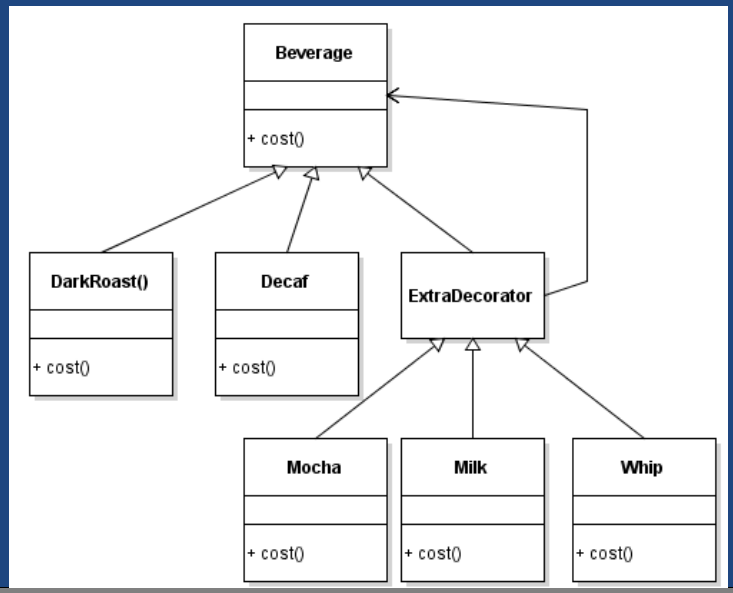

```
class ClientCode {
    void foo() {
        Beverage b = new DarkRoast();
        b = new Mocha(b);
        b = new Mocha(b);
        b = new Whip(b);
        System.out.println(b.getDescription() + " $" + b.cost());
    }

    void bar() {
        Beverage b = new Whip(new Mocha(new Mocha(new DarkRoast())));
        System.out.println(b.getDescription() + " $" + b.cost());
    }
}
```

Decorator OOD

- Design Principle:
  Favour composition over inheritance
  - Decorator still uses inheritance: gives us runtime polymorphism
  - Behaviours (methods) are modified via composition
- Design Principle:
  Open-Closed Principle
  - New “extras” added by adding new class; existing code unchanged

Decorator Features

- Decorators have..
- Can use..
  to wrap an object
  - Decorator can wrap a decorator (same supertype)
- Decorator can add behaviour before/after delegating to object it wraps to do the rest of the job
- Uses composition, so can decorate objects dynamically at runtime



Decorator Drawbacks

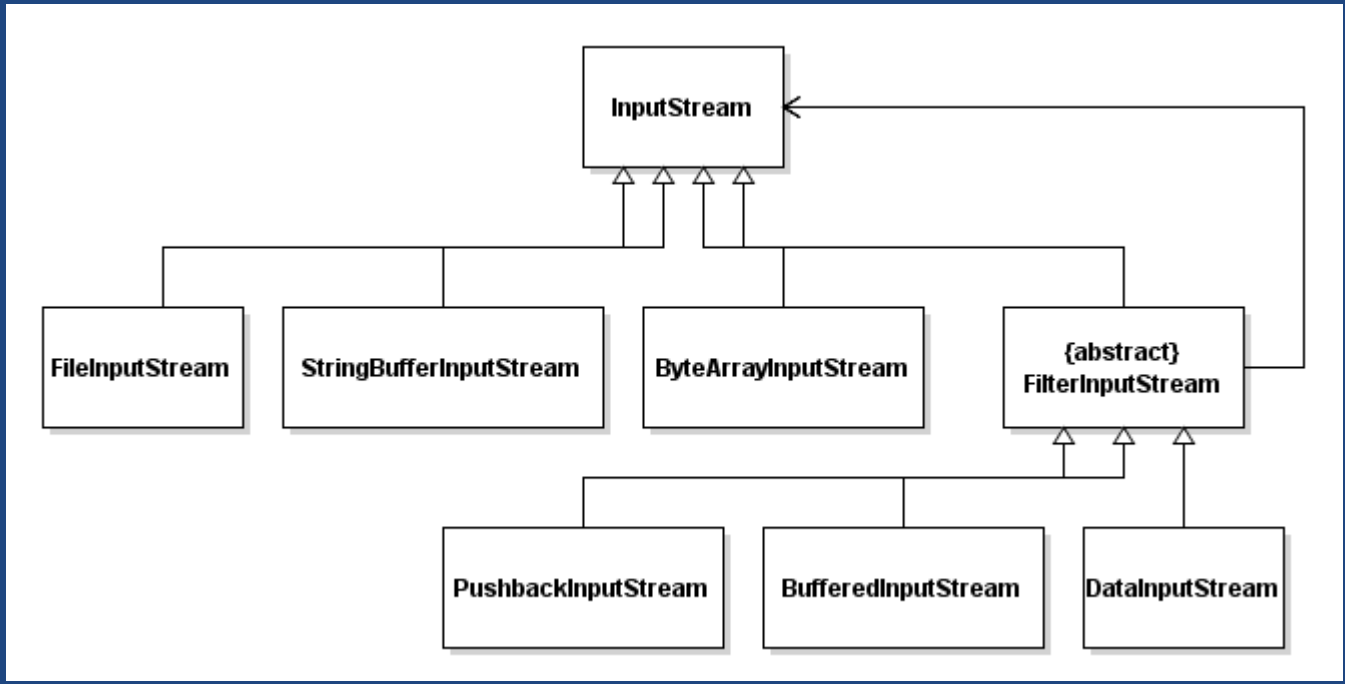
- Adds many small classes to a project
  - increased complexity for learning the OOD.
- Instantiation is more complex
  - Must instantiate the base object, and then wrap it with each decorator (can use the builder pattern)
- Client code cannot interrogate the object type to find out its inner-object's concrete type
  - But! If code depends on concrete types, it's likely bad code! Code to an interface

Coding Exercise

- In the patterns project (see sample code from lecture):
  - Add “Iced” extra (+$0.75)
  - Add “SuperSize” extra (twice the price) (Problematic – Why?)

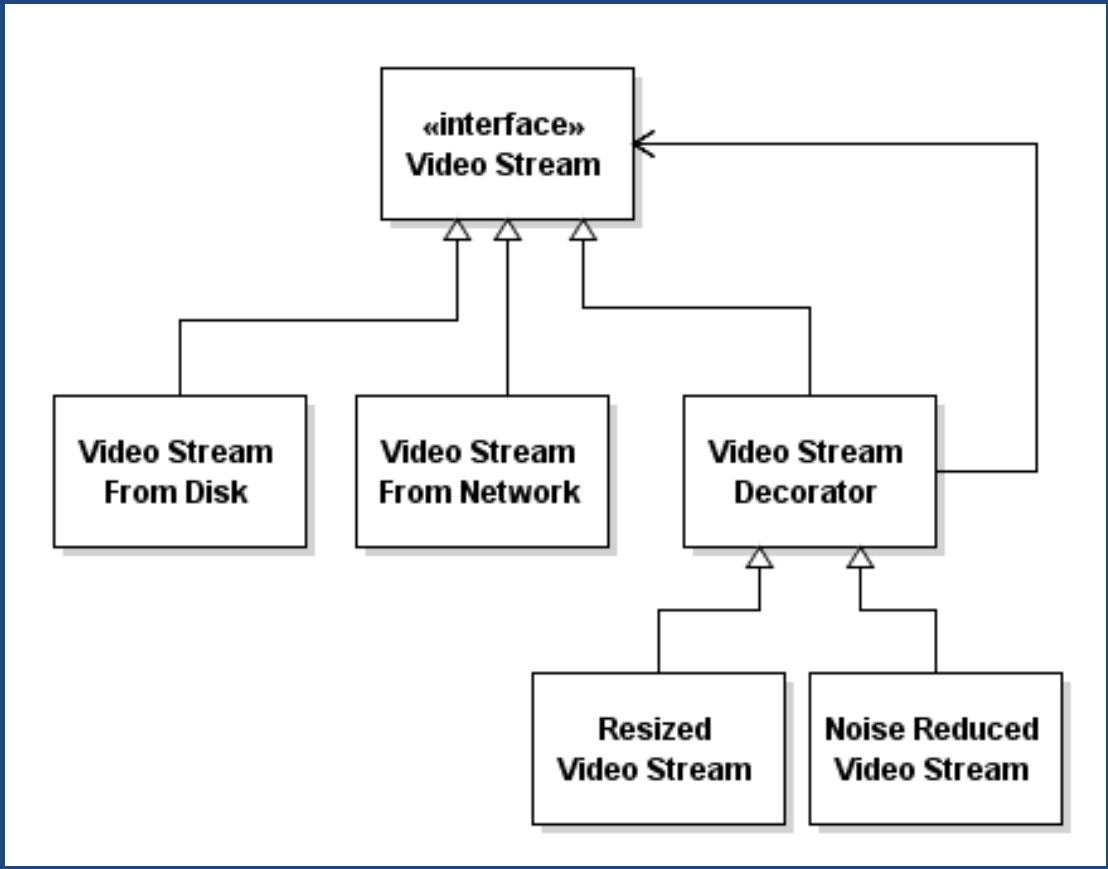
Java I/O Decorators

- Java I/O uses a few streams,
  and numerous decorators
  - Complex initially; *easier* once you know decorators



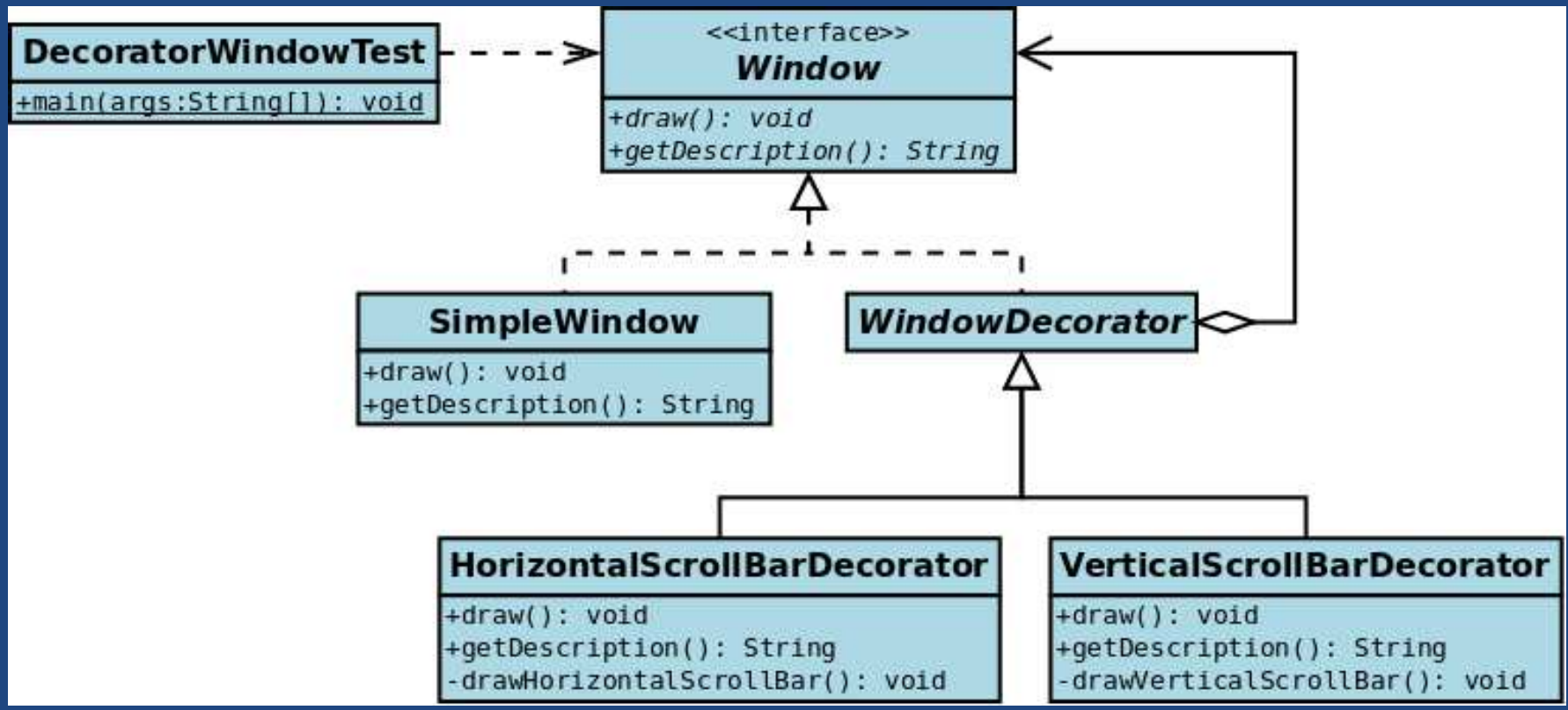
Video Stream Decorators

- Use decorators to add processing to a video stream



Scroll Bar Window Decorators

- Use decorators to add scroll bars to a window



By Nheirbaut at English Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=18612968

Summary

- Design Principles
  - Encapsulate what varies
  - Favour composition over inheritance
  - Open for extension, closed for modification
- Decorator Pattern
  - The decorator is-a base class
  - The decorator has-a base class
  - Attaches additional responsibilities to an object dynamically at runtime
- Plain coffee is so much easier!