STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

SSW 322: Software Engineering Design VI

Structural Design Patterns
---Decorator Pattern
2020 Spring

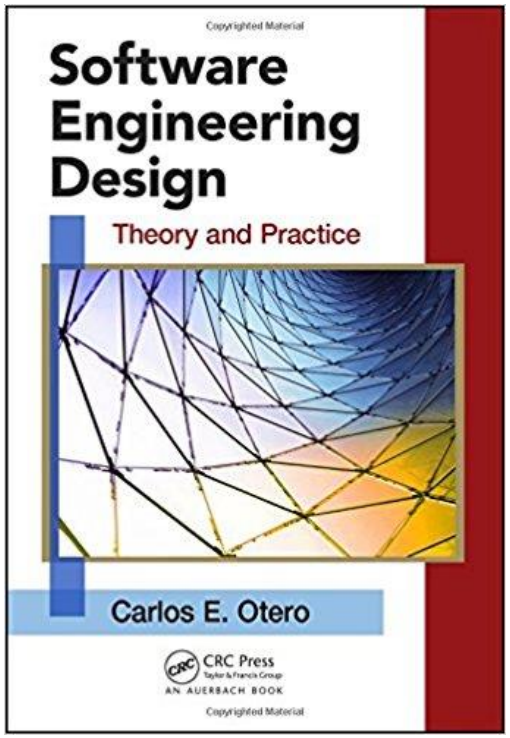
Prof. Lu Xiao
lxiao6@stevens.edu
Babbio 513
Office Hour: Monday/Wednesday 2 to 4 pm
Software Engineering
School of Systems and Enterprises

Today's Topics

- Brief review of last lecture
- Structural design patterns:
  - Decorator pattern
  - Real life example of decorator pattern-Java I/O
- Summary of bullet points

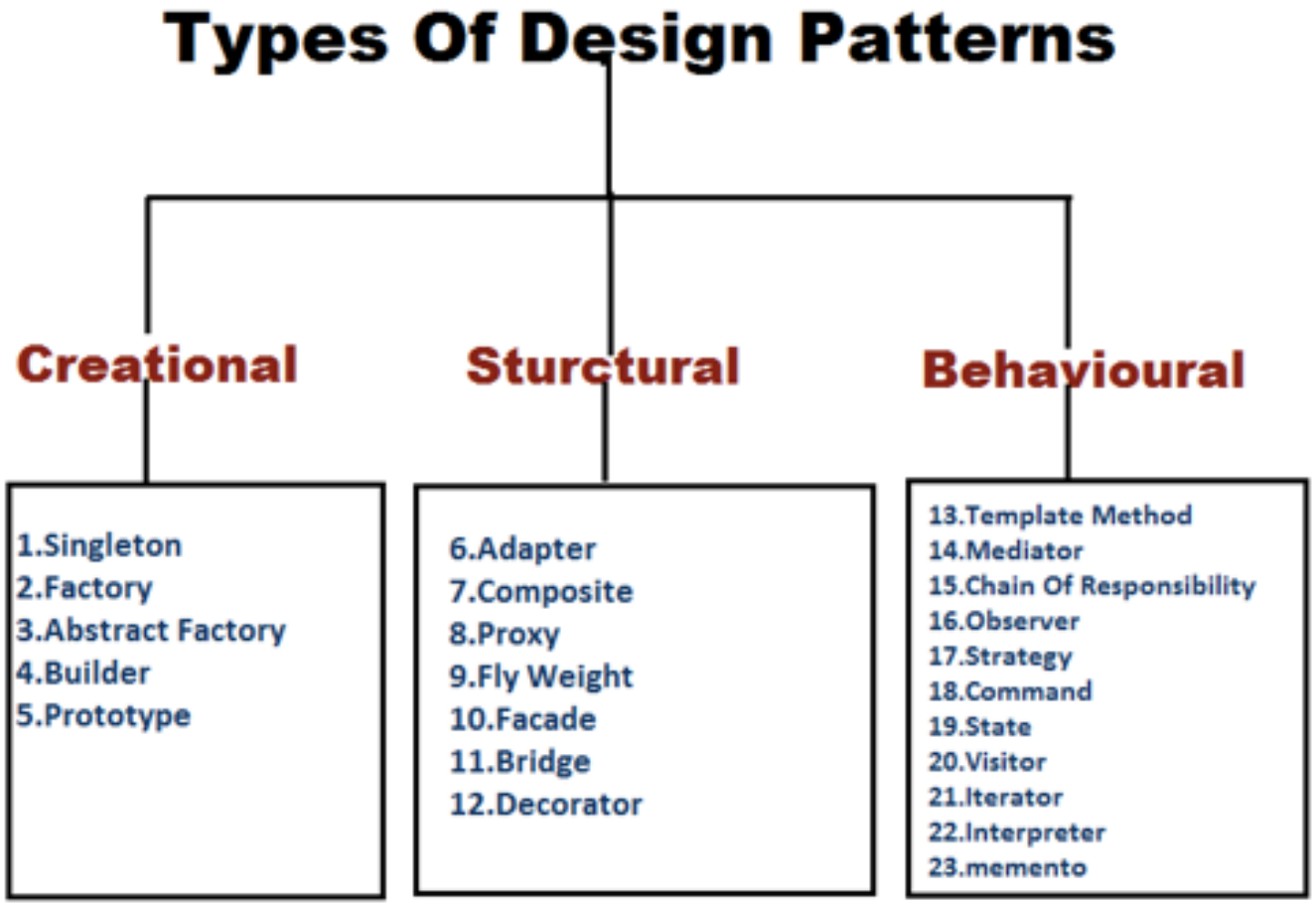
Acknowledgement

- https://sourcemaking.com/design_patterns
- **Software Engineering Design: Theory and Practice**



Design Pattern

- ***Common*** design solutions to problems that ***recur*** in different systems.



Last Lecture

- What type of design patterns we have learned?

- What are three design patterns

Last Lecture

- What type of design patterns we have learned?
  - Creational design pattern
- What are three design patterns
  - Abstract factory pattern
  - Factory method pattern
  - Singleton pattern

Structural Design Patterns

- In Software Engineering, Structural Design Patterns are Design Patterns that ease the design by identifying a simple way to realize relationships between entities.
- Deal with designing *larger* structure from *existing* classes or objects at *run time*.
  - *What are the possible ways to do so in OO design?*
- Significantly impact the reusability and modifiability of systems.
  - E.g. adapter, ***decorator***, composite, and facade

Structural Design Patterns

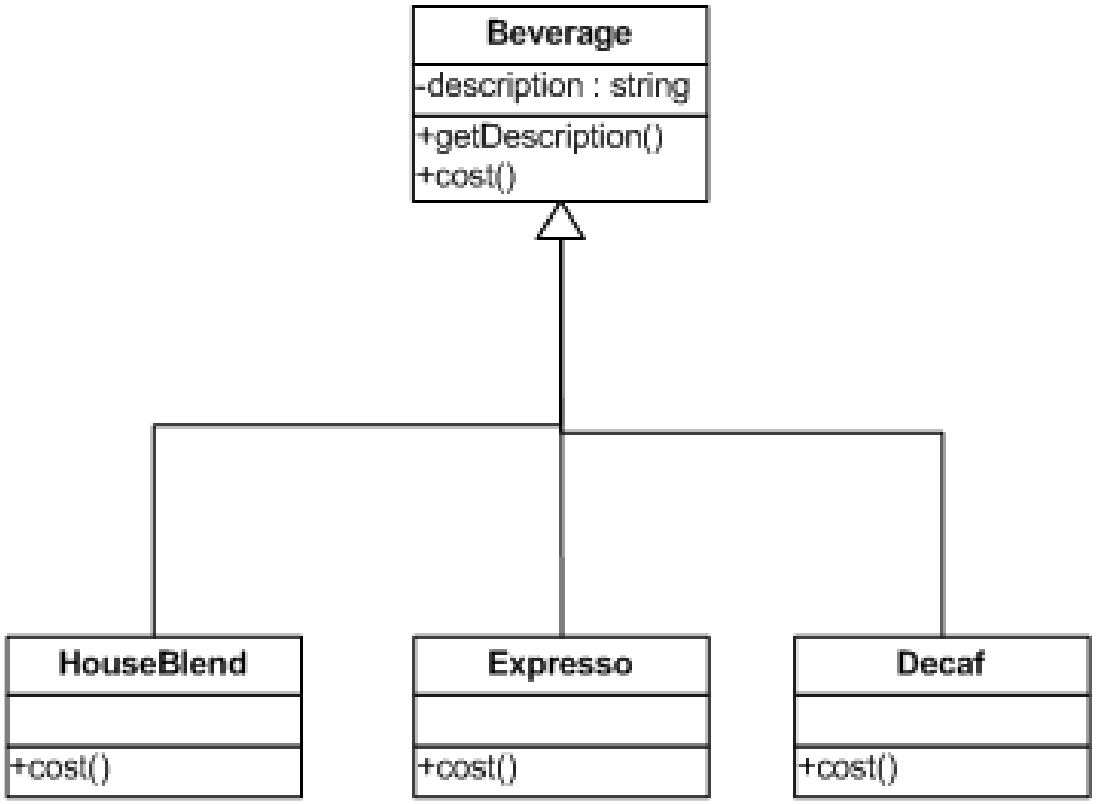
- In Software Engineering, Structural Design Patterns are Design Patterns that ease the design by identifying a simple way to realize relationships between entities.
- Deal with designing *larger* structure from *existing* classes or objects at *run time*.
  - *What are the possible ways to do so in OO design?*
    - ***Composition and Inheritance***
- Significantly impact the reusability and modifiability of systems.
  - E.g. adapter, ***decorator***, composite, and facade

Decorator Pattern
---add responsibilities to objects dynamically

The Coffee Shop-An Exercise in OO Problem Solving

- You have been hired to help with an ordering and accounting system for a the fastest growing coffee shops chain, ***StarBuzz***
- Their locations and their beverage offerings are growing more quickly that you can say “A tall latte”!
- So quickly they have problems keeping their ordering systems up-to-date

Original Design



- Beverage is an abstract super-class for all types of beverages offered
- The cost() method must be implemented by each Beverage sub-class

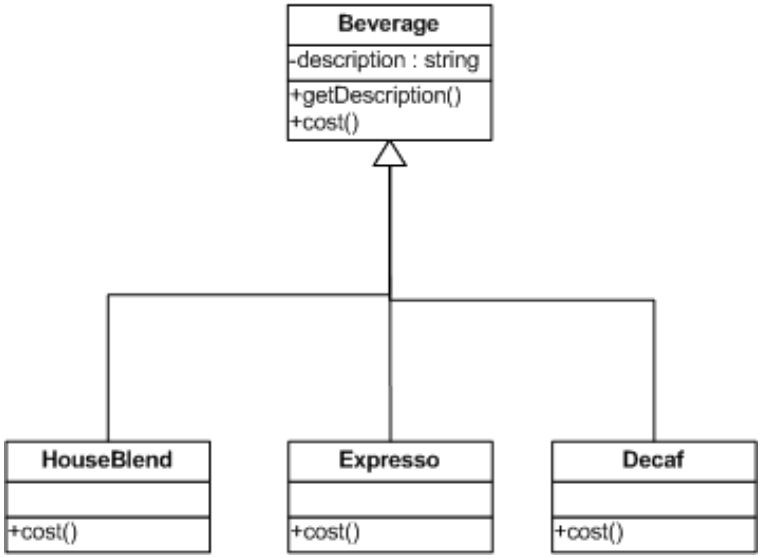
What is software made for?

What is software made for?

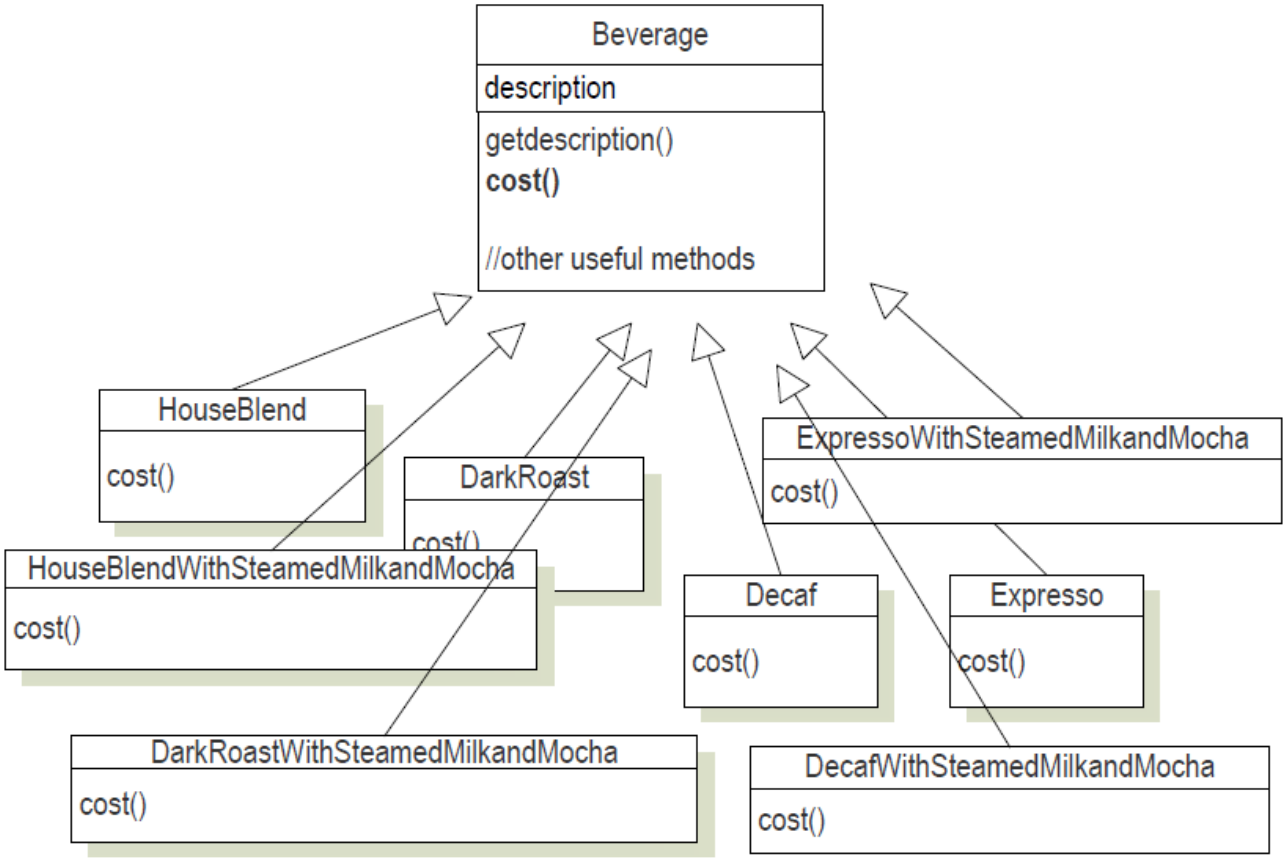
Change!!

A Change

- StarBuzz will offer different types of ***coffee-based*** beverage
  - Mocha
  - Latte
  - Cappuccino
  - …

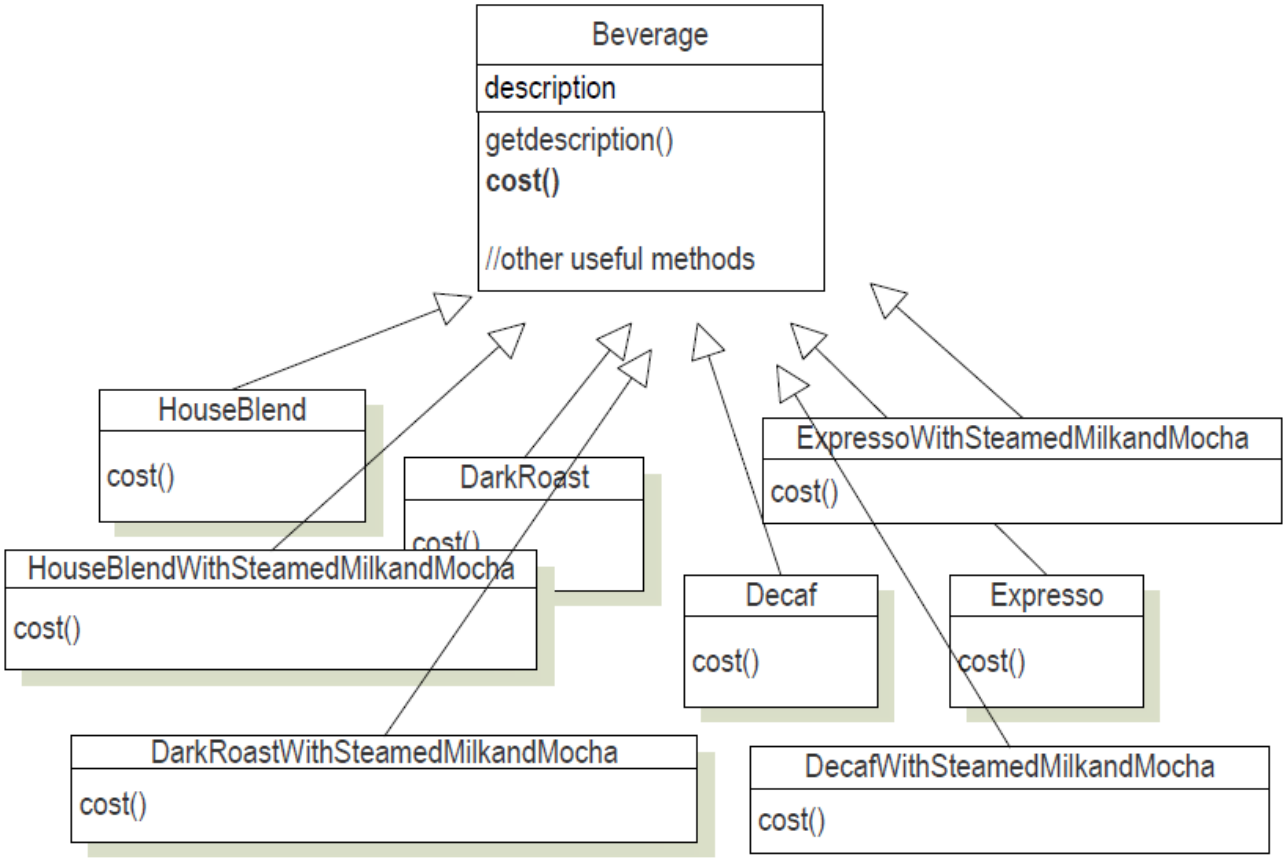


Solution 1



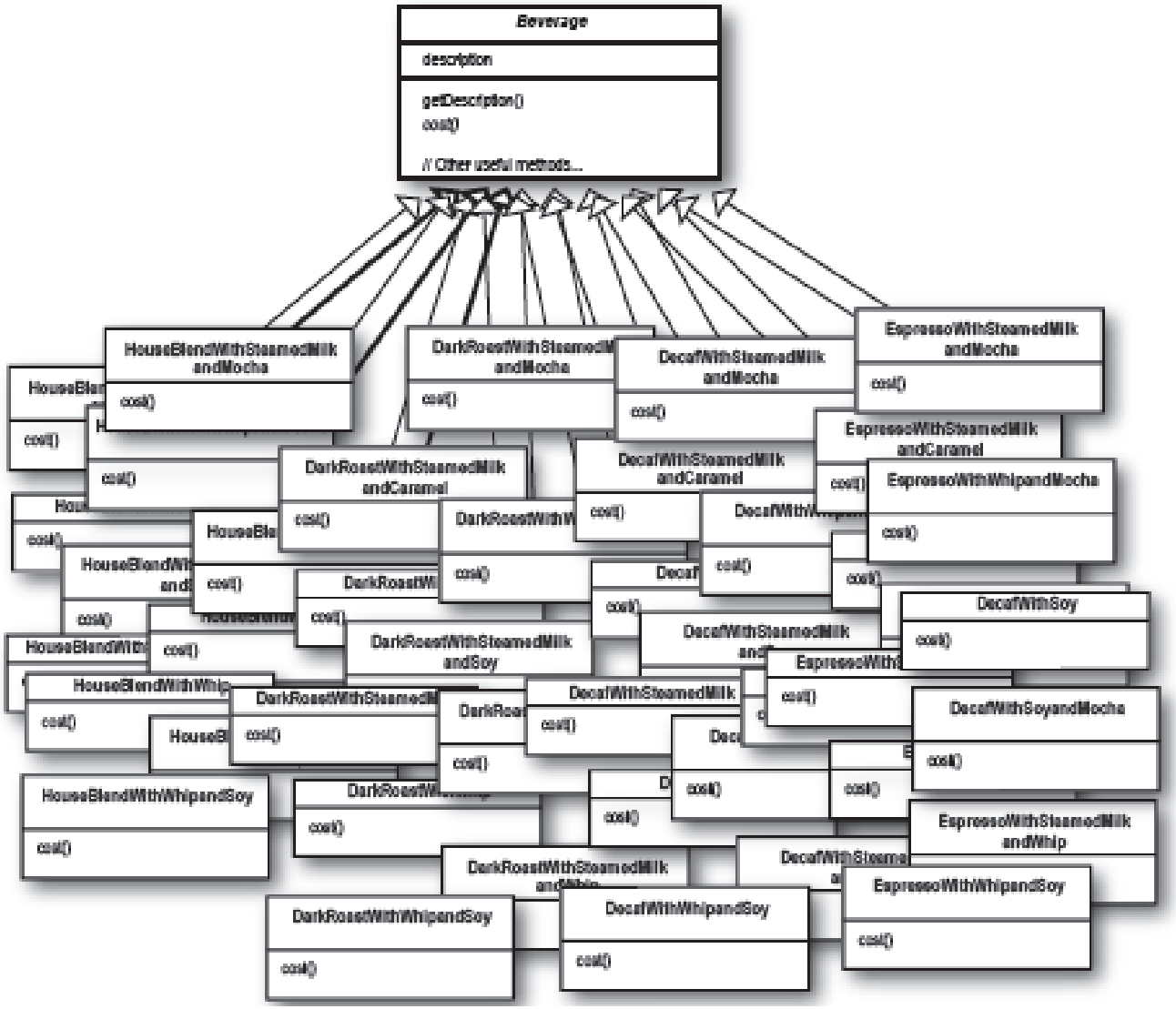
- Each cost() implementation must account for all ingredients of that Beverage

Solution 1

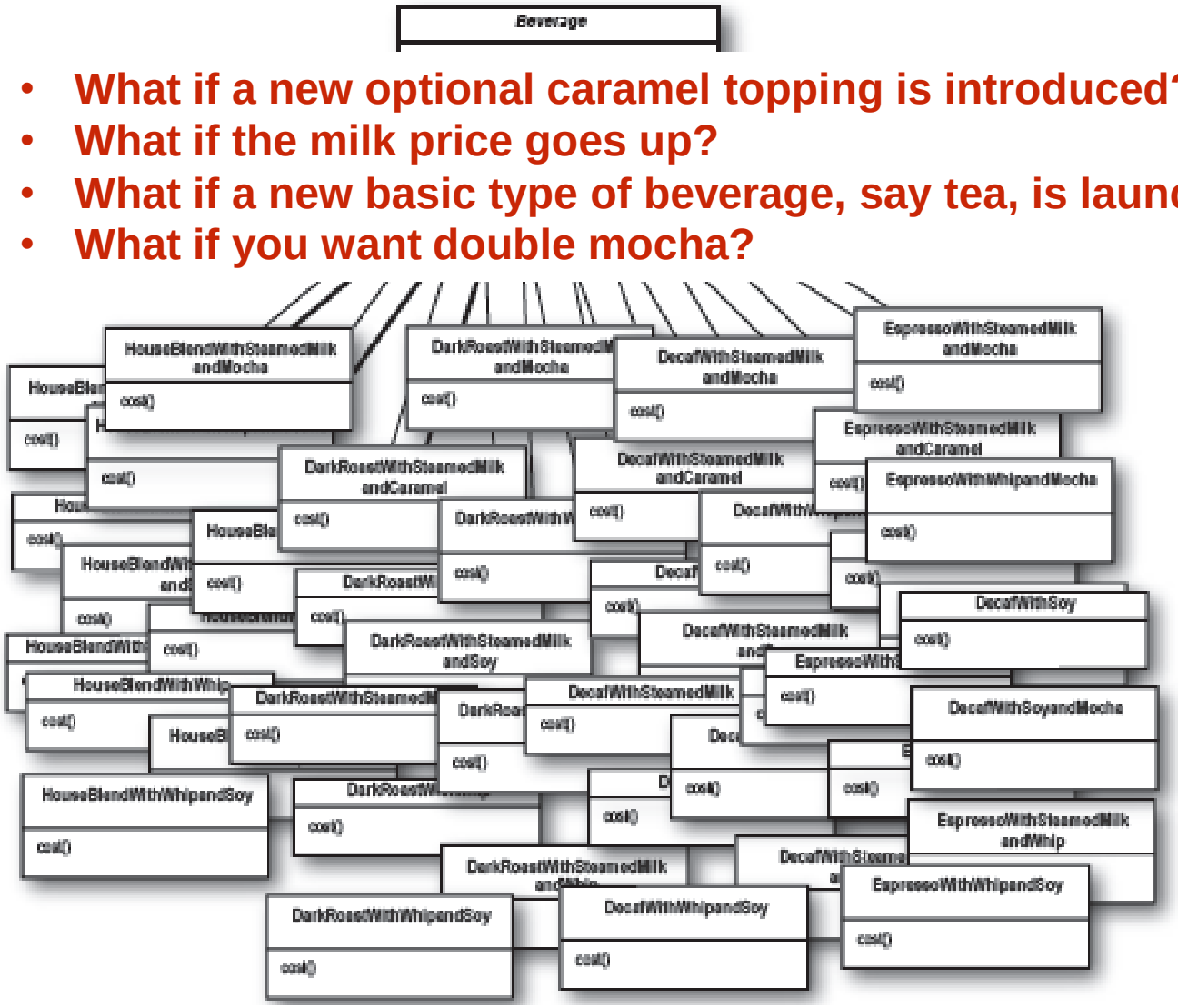


Any comments on solution 1? Is it good?

A Nightmare…

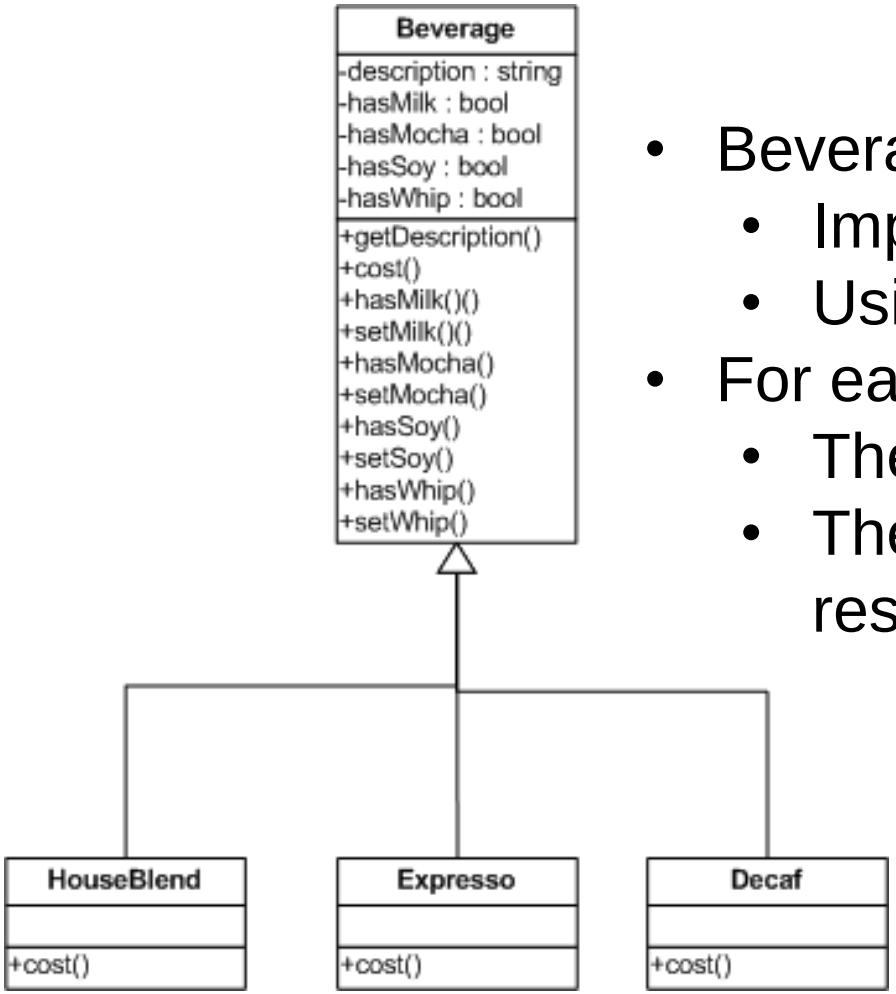

Beverage
description
getDescription()
cost()
// Other useful methods...
HouseBlendWithSteamedMilk andMocha
cost()
DarkRoastWithSteamedMilk andMocha
cost()
DecafWithSteamedMilk andMocha
cost()
EspressoWithSteamedMilk andMocha
cost()
EspressoWithSteamedMilk andCaramel
cost()
DarkRoastWithSteamedMilk andCaramel
cost()
DecafWithSteamedMilk andCaramel
cost()
EspressoWithWhipandMocha
cost()
DecafWithSoy
cost()
DarkRoastWithSteamedMilk andSoy
DecafWithSteamedMilk
cost()
DecafWithSoyandMocha
cost()
HouseBlendWithWhipandSoy
cost()
EspressoWithSteamedMilk andWhip
DarkRoastWithWhipandSoy
cost()
DecafWithWhipandSoy
cost()
EspressoWithWhipandSoy
cost()

A Nightmare…



- What if a new optional caramel topping is introduced?
- What if the milk price goes up?
- What if a new basic type of beverage, say tea, is launched?
- What if you want double mocha?

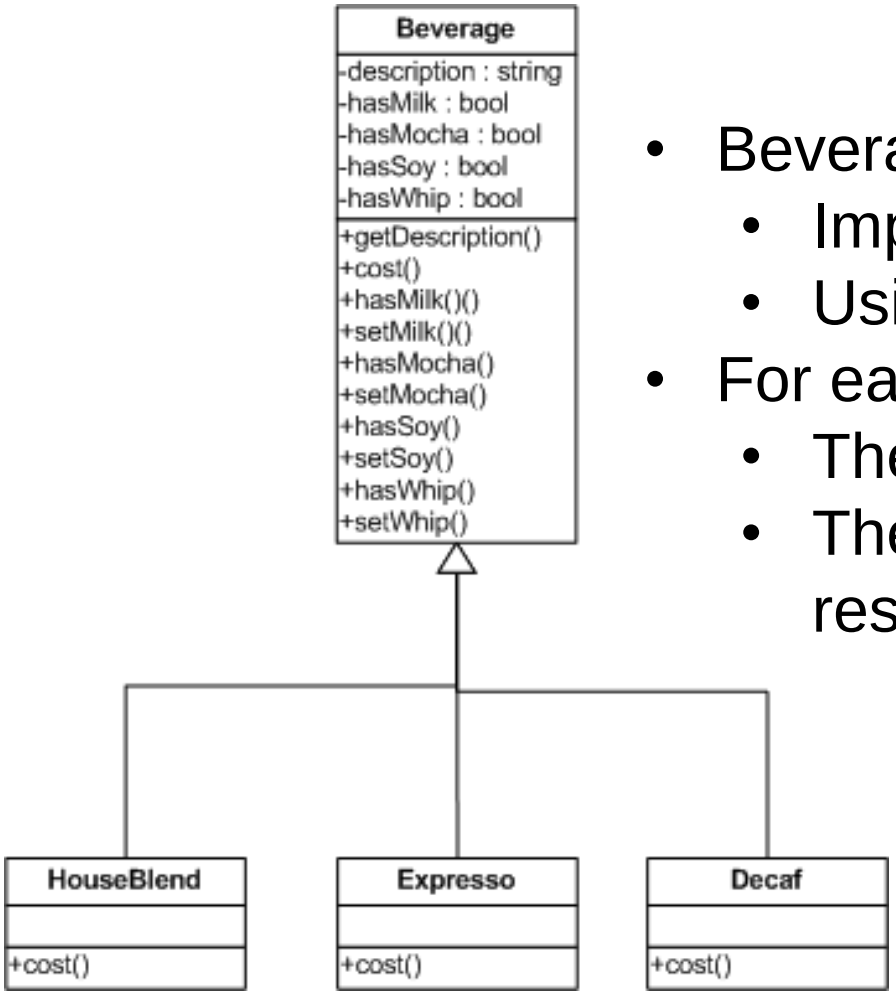
Solution 2



- Beverage is no more abstract
  - Implements the cost() method
  - Using info in its instance variables
- For each actual beverage:
  - The cost() method calls super.cost()
  - Then adds its own pricing to that result

Solution 2

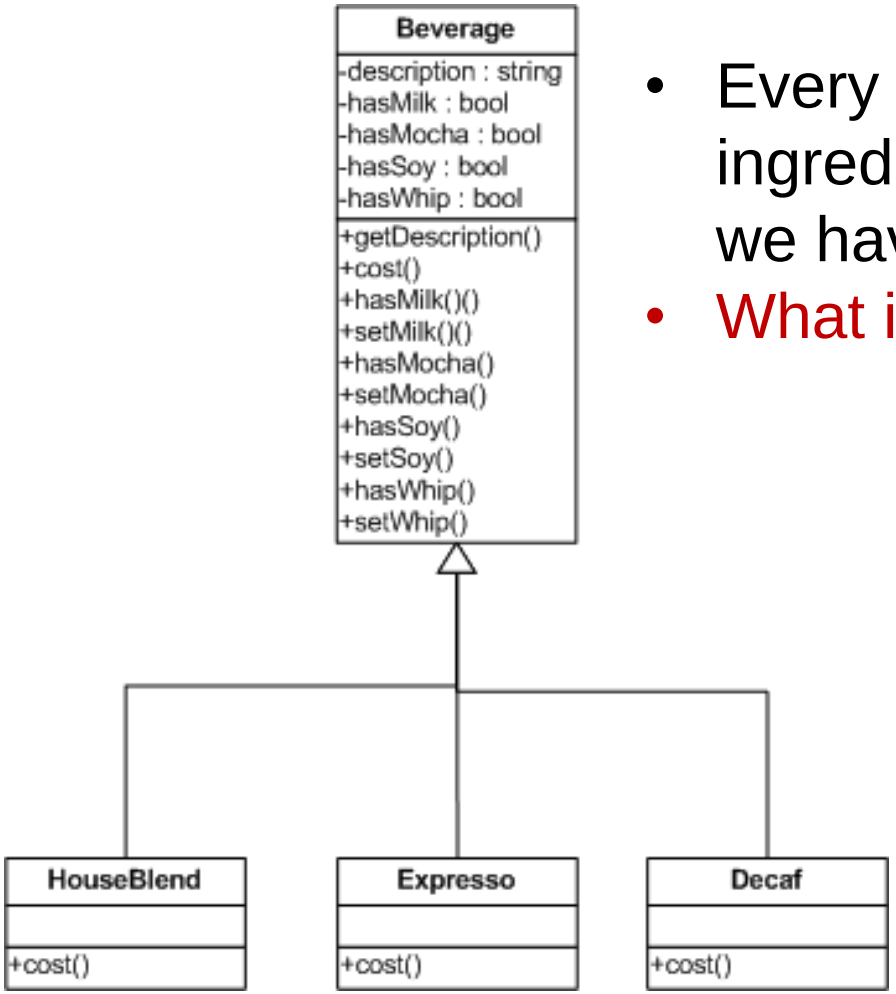
Any comments on solution 2? Is it good?



- Beverage is no more abstract
  - Implements the cost() method
  - Using info in its instance variables
- For each actual beverage:
  - The cost() method calls super.cost()
  - Then adds its own pricing to that result

Solution 2

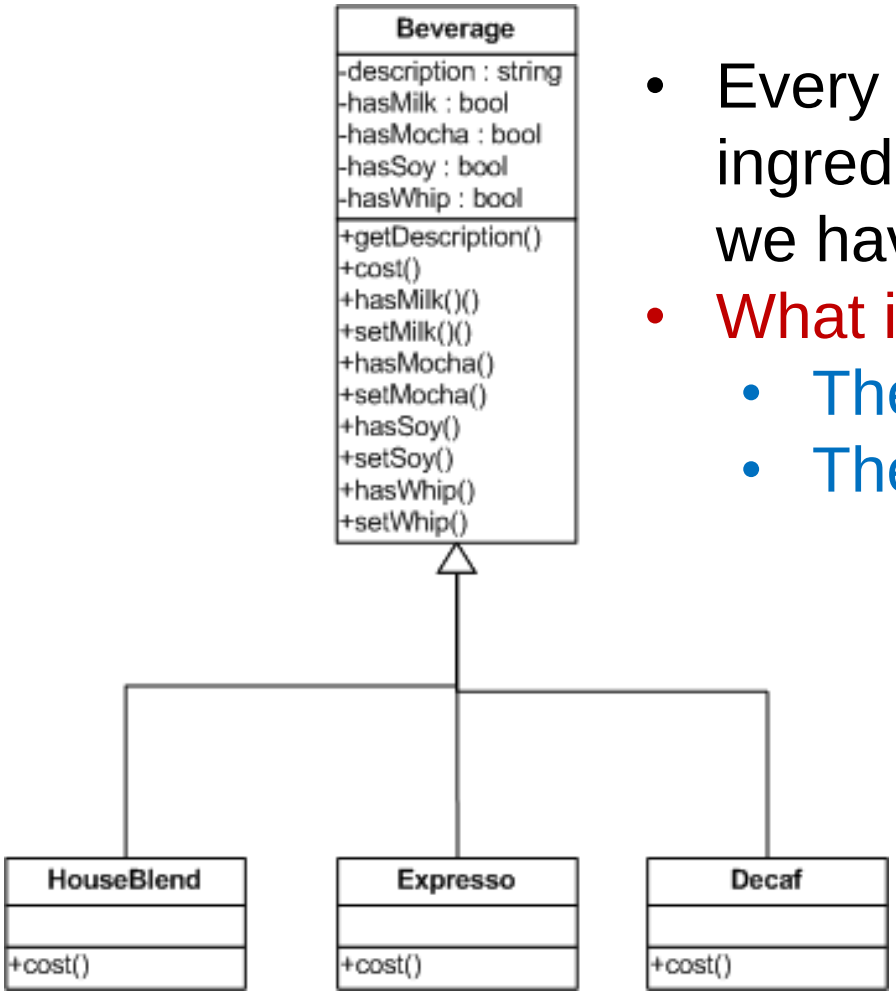
Any comments on solution 2? Is it good?



- Every time a new condiment or other ingredient is included in the offering we have some code change.
- What is the impact?

Solution 2

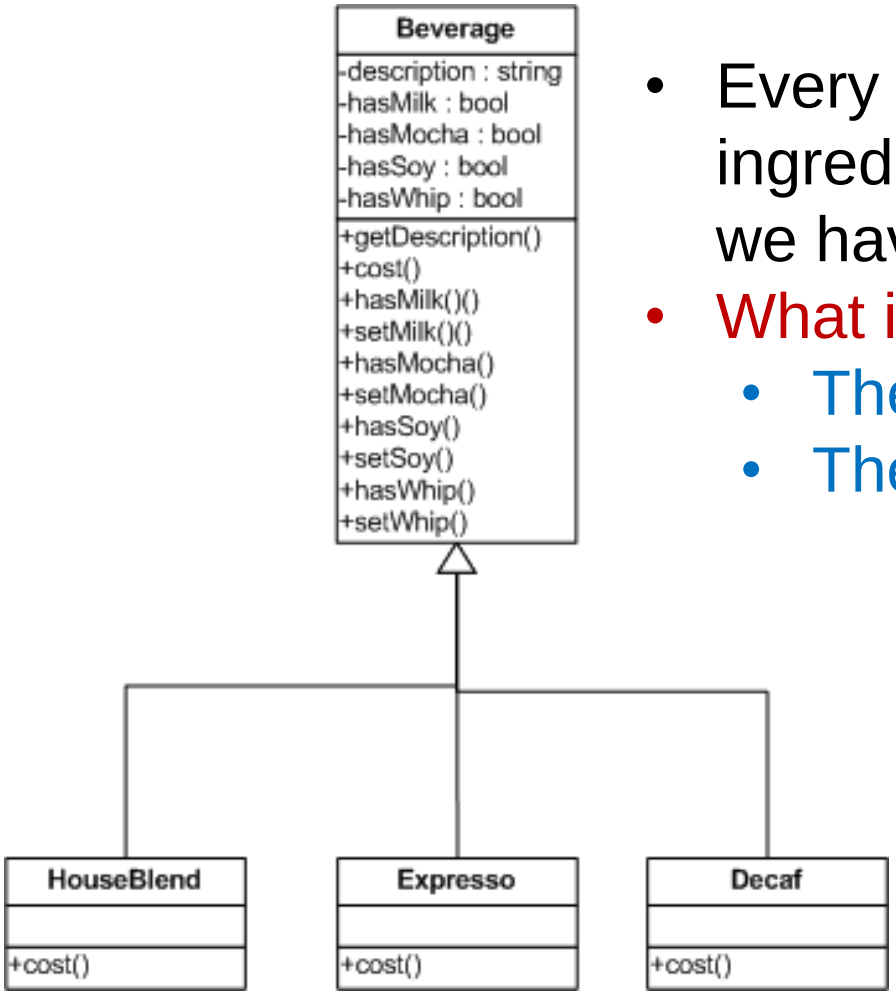
Any comments on solution 2? Is it good?



- Every time a new condiment or other ingredient is included in the offering we have some code change.
- What is the impact?
  - The structure of the Beverage
  - The algorithm of Beverage.cost()

Solution 2

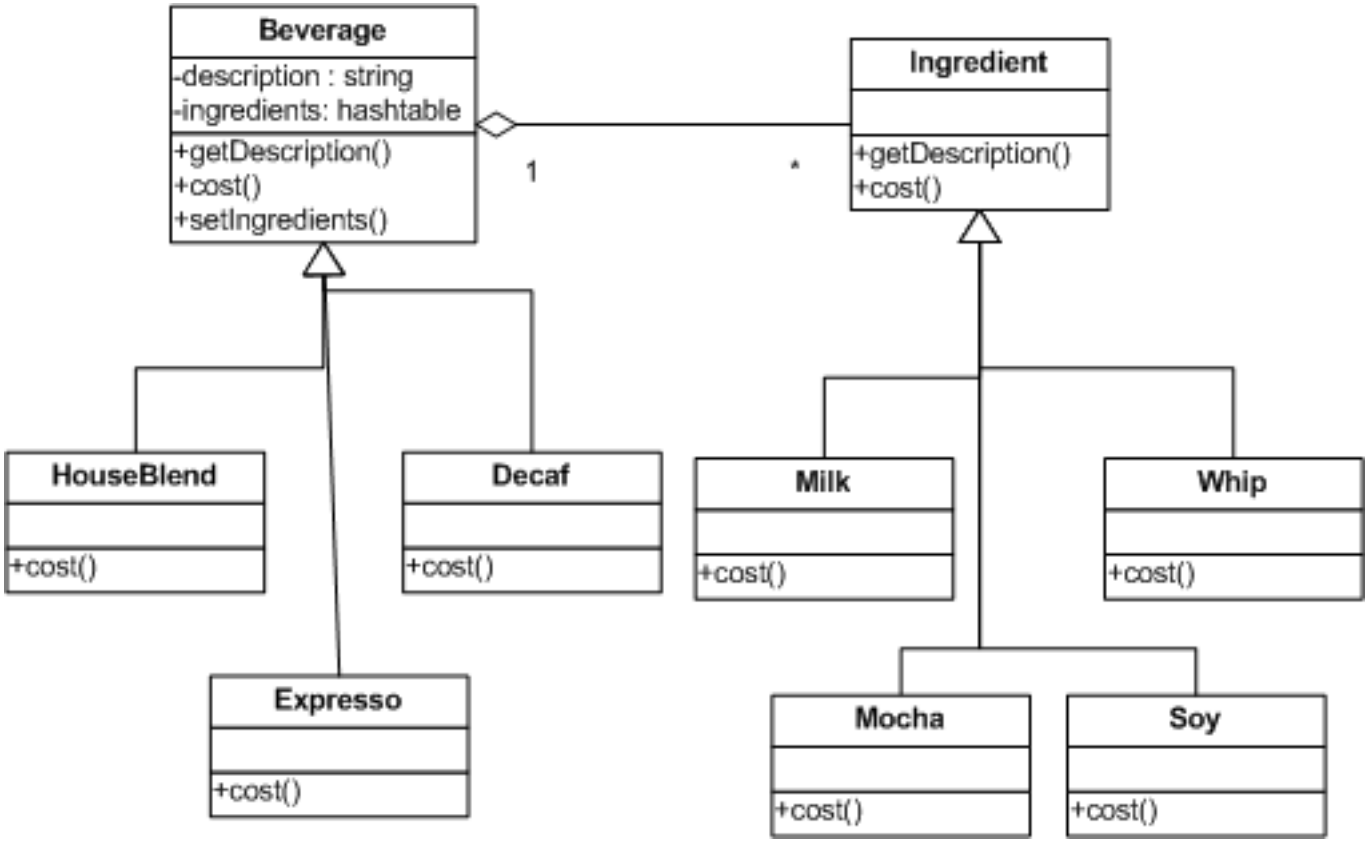
Any comments on solution 2? Is it good?



- Every time a new condiment or other ingredient is included in the offering we have some code change.
- What is the impact?
  - The structure of the Beverage
  - The algorithm of Beverage.cost()

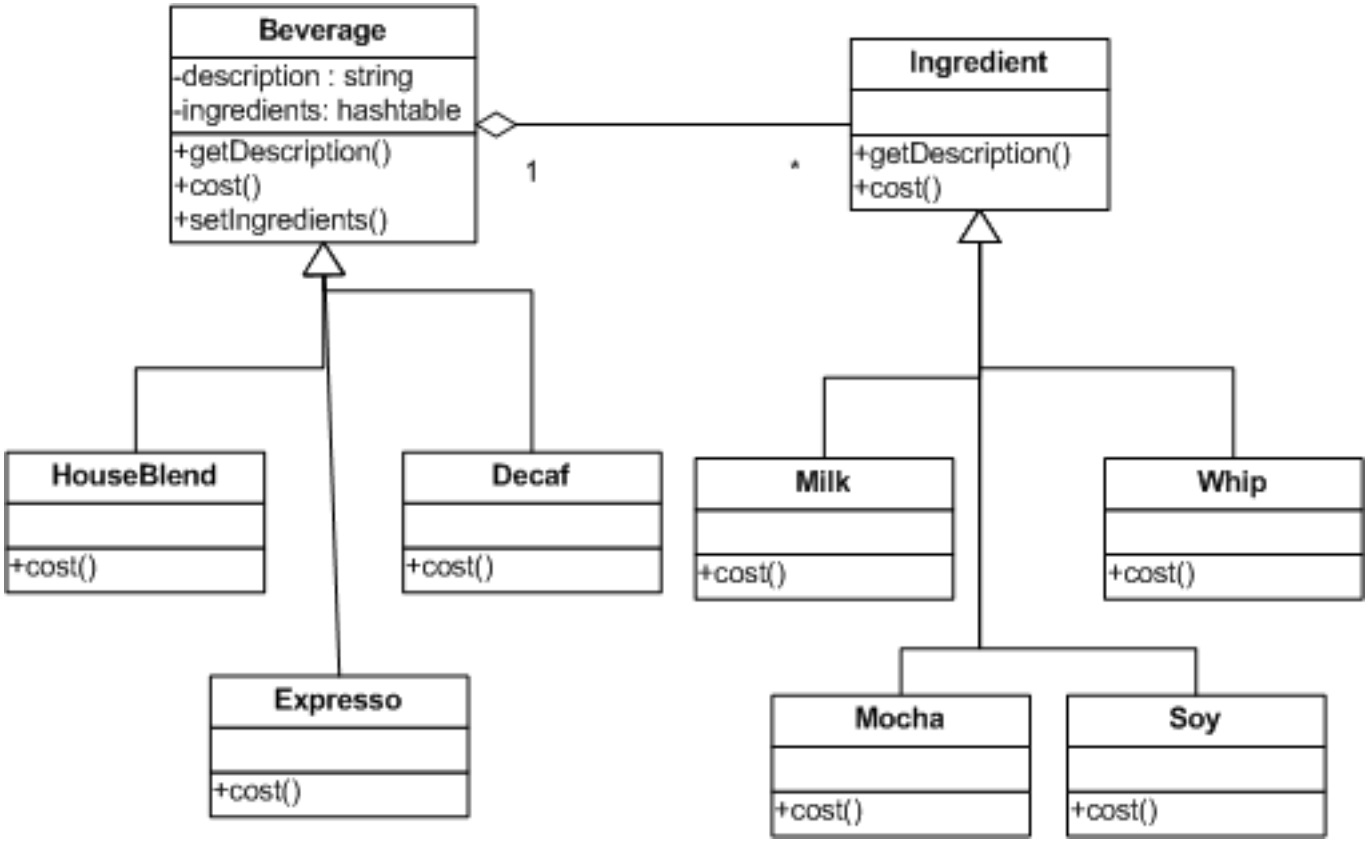
These recurring changes to existing code increase the chances of introducing bugs or bad side effects.

Solution 3


Beverage
-description : string
-ingredients: hashtable
+getDescription()
+cost()
+setIngredients()
1
*
Ingredient
+getDescription()
+cost()
HouseBlend
+cost()
Decaf
+cost()
Expresso
+cost()
Milk
+cost()
Whip
+cost()
Mocha
+cost()
Soy
+cost()

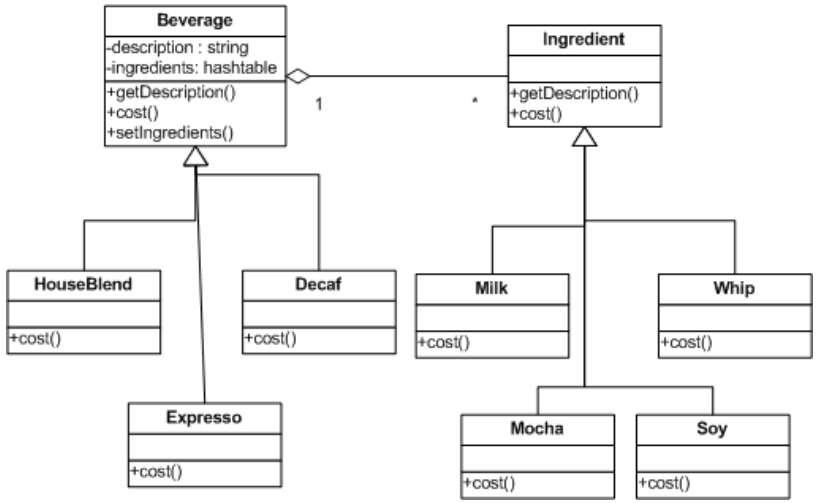
Solution 3

Any comments on solution 3? Is it good?



Solution 3

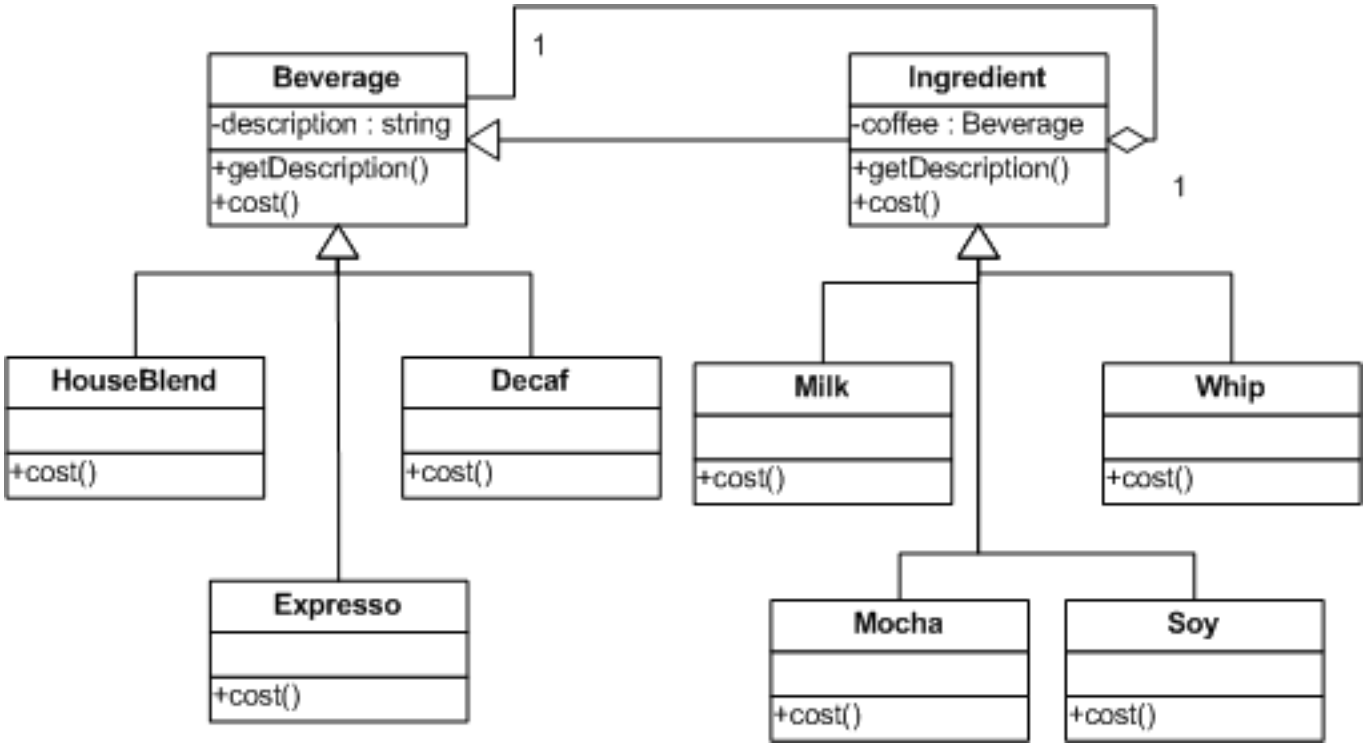
Any comments on solution 3? Is it good?



Still Not Flexible Enough…

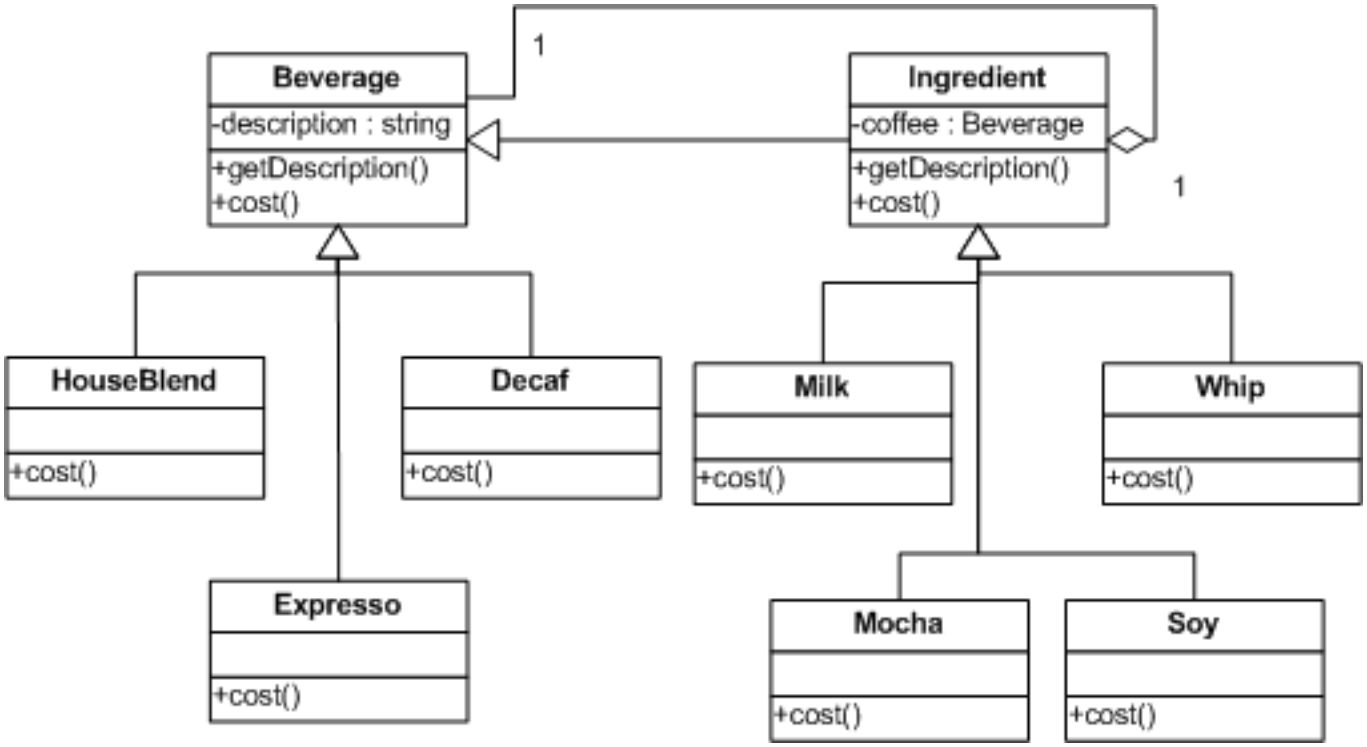
- In the implementation of cost(), it implies that the ingredient cost is added *before or after* the cost of the coffee
- In many cases, the order of these behaviors ***can not be determined beforehand***.

Finally, meet the decorator pattern…



What's the key feature of this solution?

Finally, meet the decorator pattern…



What's the key feature of this solution?

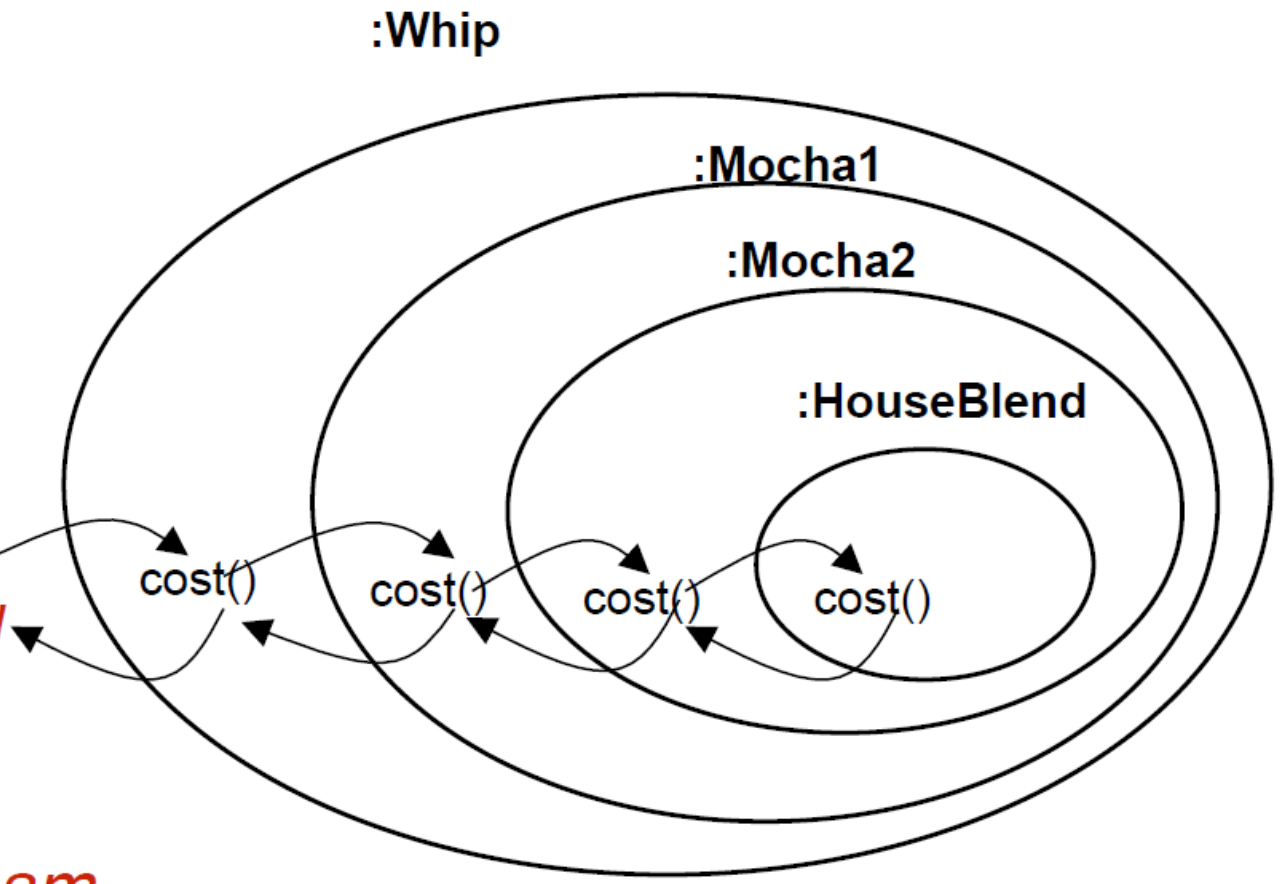
1. Ingredient implements (is-a) Beverage
2. Ingredient is composed of (contains) a Beverage

Decorator Basic Idea

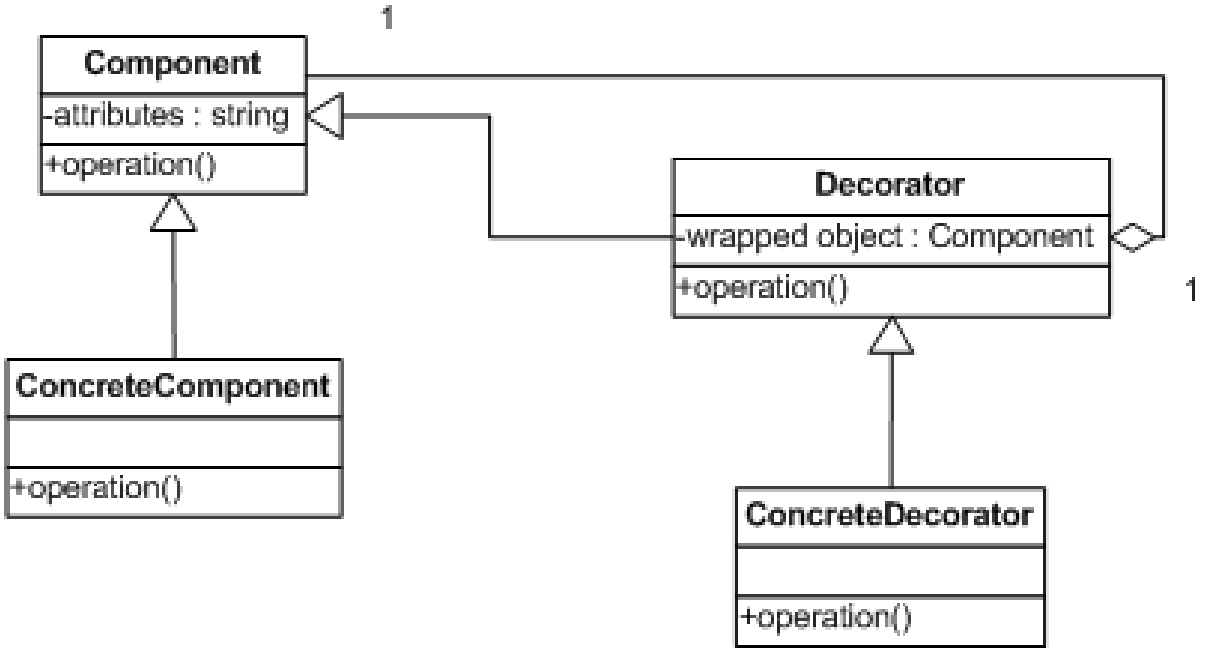
- Start with an instance (object) of some type of beverage
- "Decorate" that instance at run time with ingredients
  - Think of decorator objects as wrappers
  - Every instance of beverage served is a different collection of objects
    - ***Many decorators + 1 main beverage instance***
- *Example order: HouseBlend with double mocha and whipped cream*

Constructing Your Order

- *Order:*
  - *HouseBlend*
  - *mocha*
  - *mocha*
  - *whipped cream*



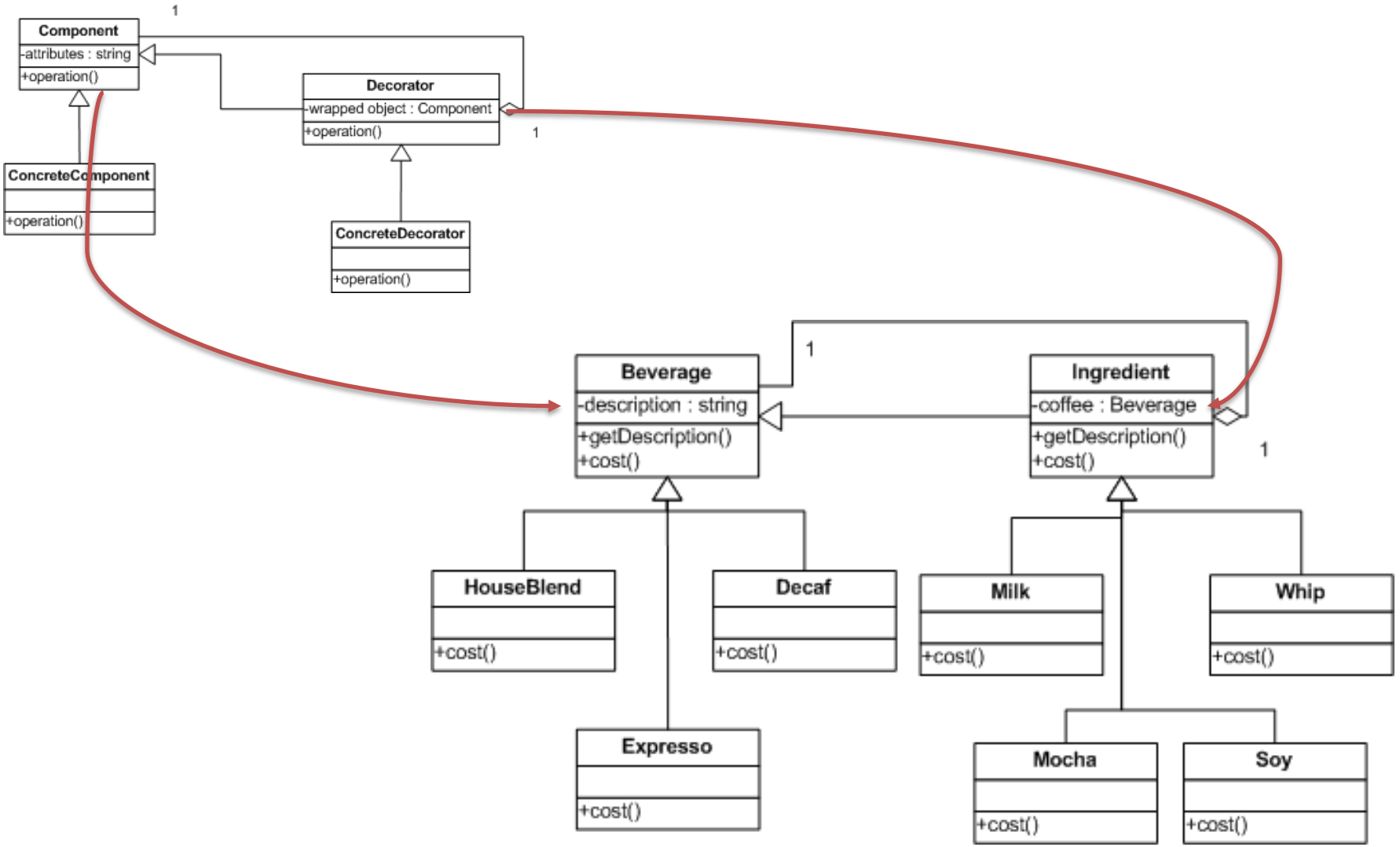
Decorator Pattern- Photo ID



- "The Decorator pattern dynamically attaches additional responsibilities to an object"
- Decorators provides an alternative to inheritance for extending functionality
  - Through wrapping, i.e. composition

Remember: ***Favor composition over inheritance***

Decorator Pattern- Photo ID



Why decorator pattern is good?

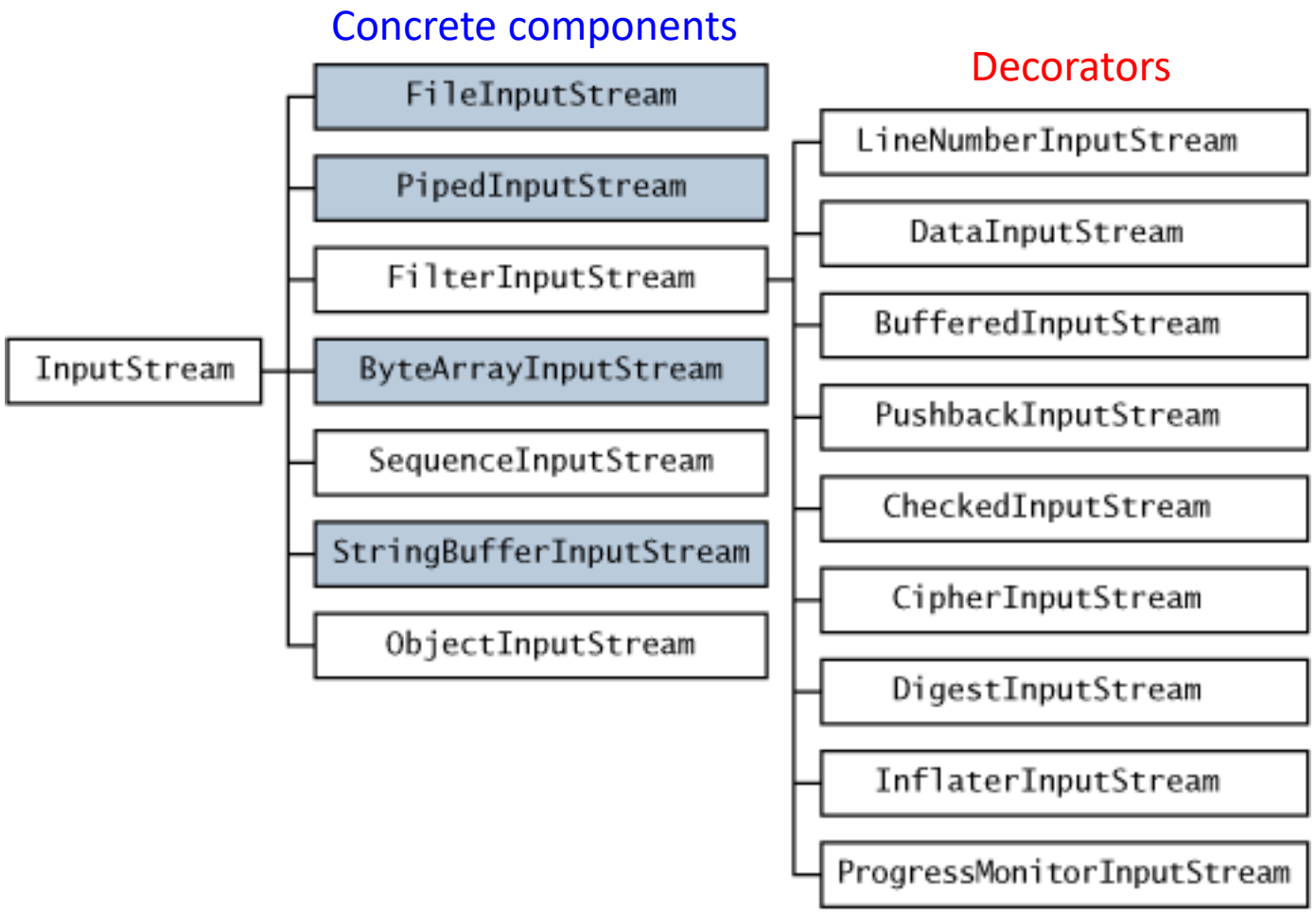
- All types of decorators have the same super-type as the object they decorate
  - You can pass around the wrapping decorator object instead of the “decorated” object
- Behavior: The decorator executes its own behavior either before or after invoking the same behavior on the object it wraps depending on business logic
  - ***Dynamism: decorator objects can be added at run time at any moment***

Real-world Decorators ---Java I/O

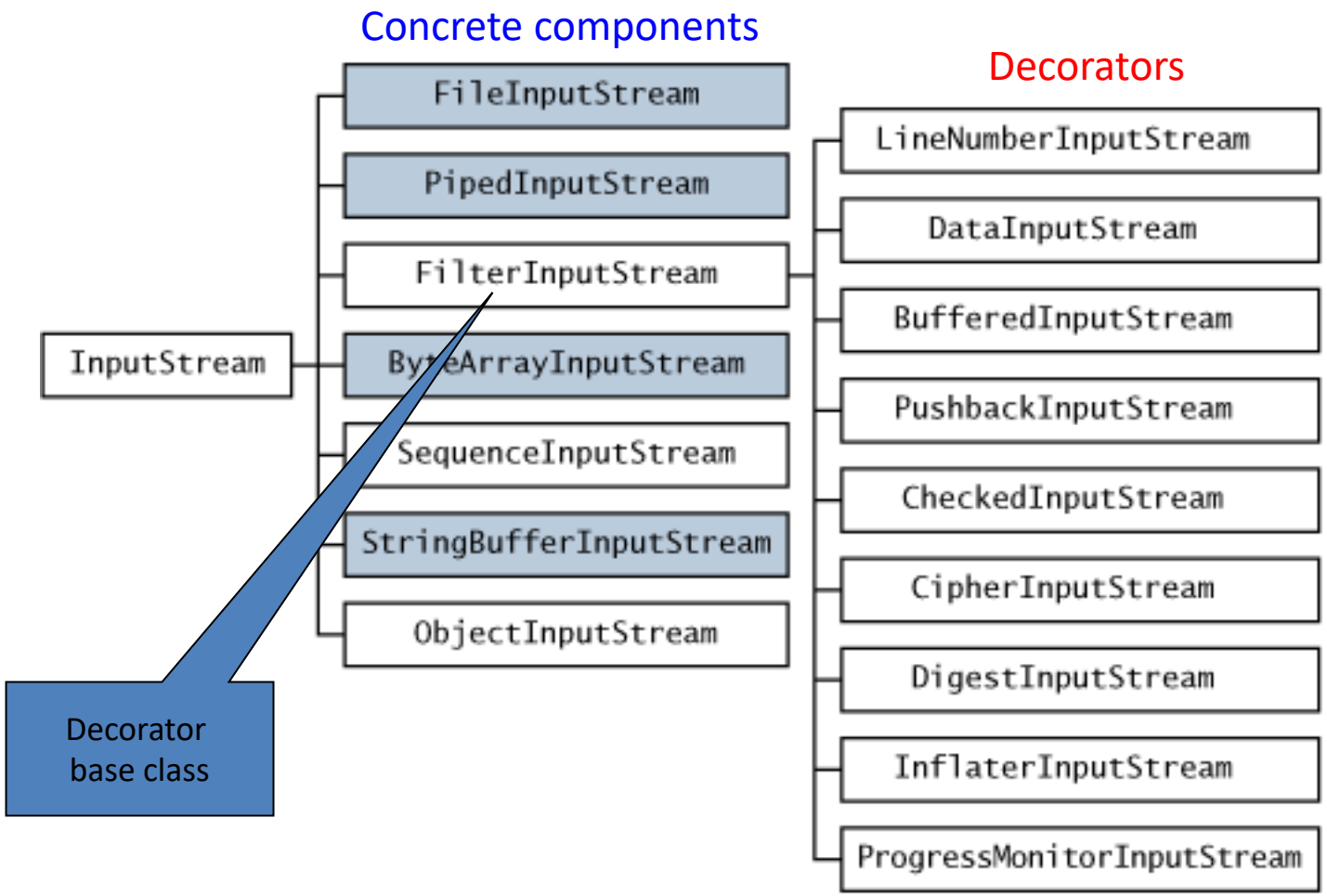
Decorator Usage: Java I/O

- The Java I/O library of “stream” classes is an example of the Decorator pattern
- java.io.InputStream is the abstract Component
  - Specialized by a bunch of concrete types of input stream classes
- java.io.FilterInputStream is the abstract Decorator
  - Specialized by a hunch of concrete types of decorators to provide additional functions at run-time

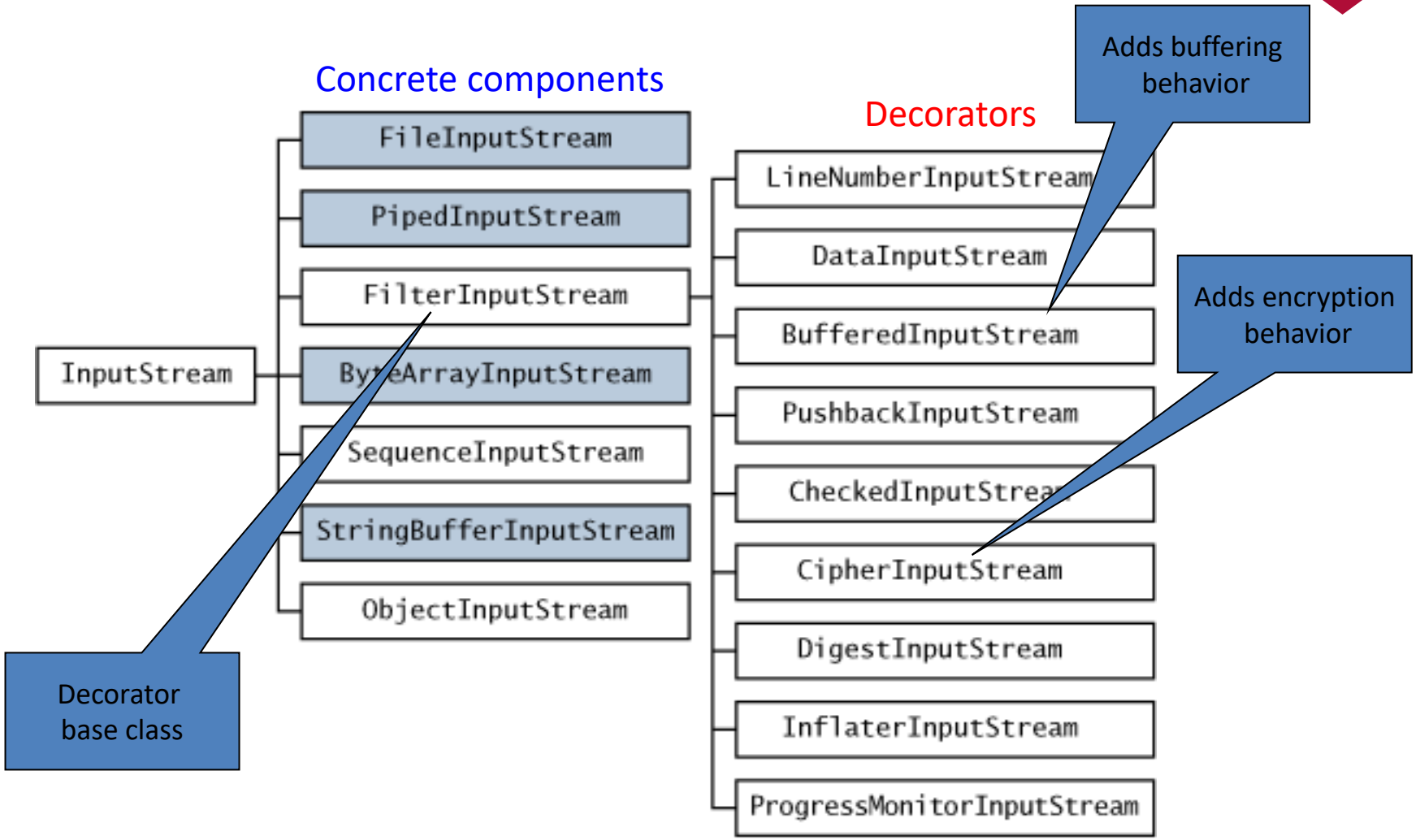
Java Stream Classes



Java Stream Classes



Java Stream Classes



Decorator Pattern- Summary

- Classification: Structural Pattern
- Context: *dynamically* add functionality
- Problem: you want to use the basic functionality of an object, but you may need to invoke either *before or after* that some extra functionality
- Solution: extension of basic functionality without subclassing
- Consequences:
  - Small objects that host extra functionality only
  - Instantiates possibly a lot of objects

Bullet Points

- Inheritance is one form of extension, but not necessarily the best way to achieve flexibility in our designs.
- In our designs we should allow behavior to be extended without the need to modify existing code.
- Composition and delegation can often be used to add new behaviors at runtime.

Remember: ***Favor composition over inheritance***

Bullet Points

- The Decorator Pattern provides an alternative to subclassing for extending behavior.
- The Decorator Pattern involves a set of decorator classes that are used to wrap concrete components.
- Decorator classes mirror the type of the components they decorate.
  - In fact, they are the same type as the components they decorate, either through inheritance or interface implementation.

Bullet Points

- 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.
- You can wrap a component with any number of decorators.
- Decorators are typically transparent to the client of the component unless the client is relying on the component's concrete type.


thank you