CS427a: Object-Oriented Programming

Design Patterns for Flexible and Reusable design

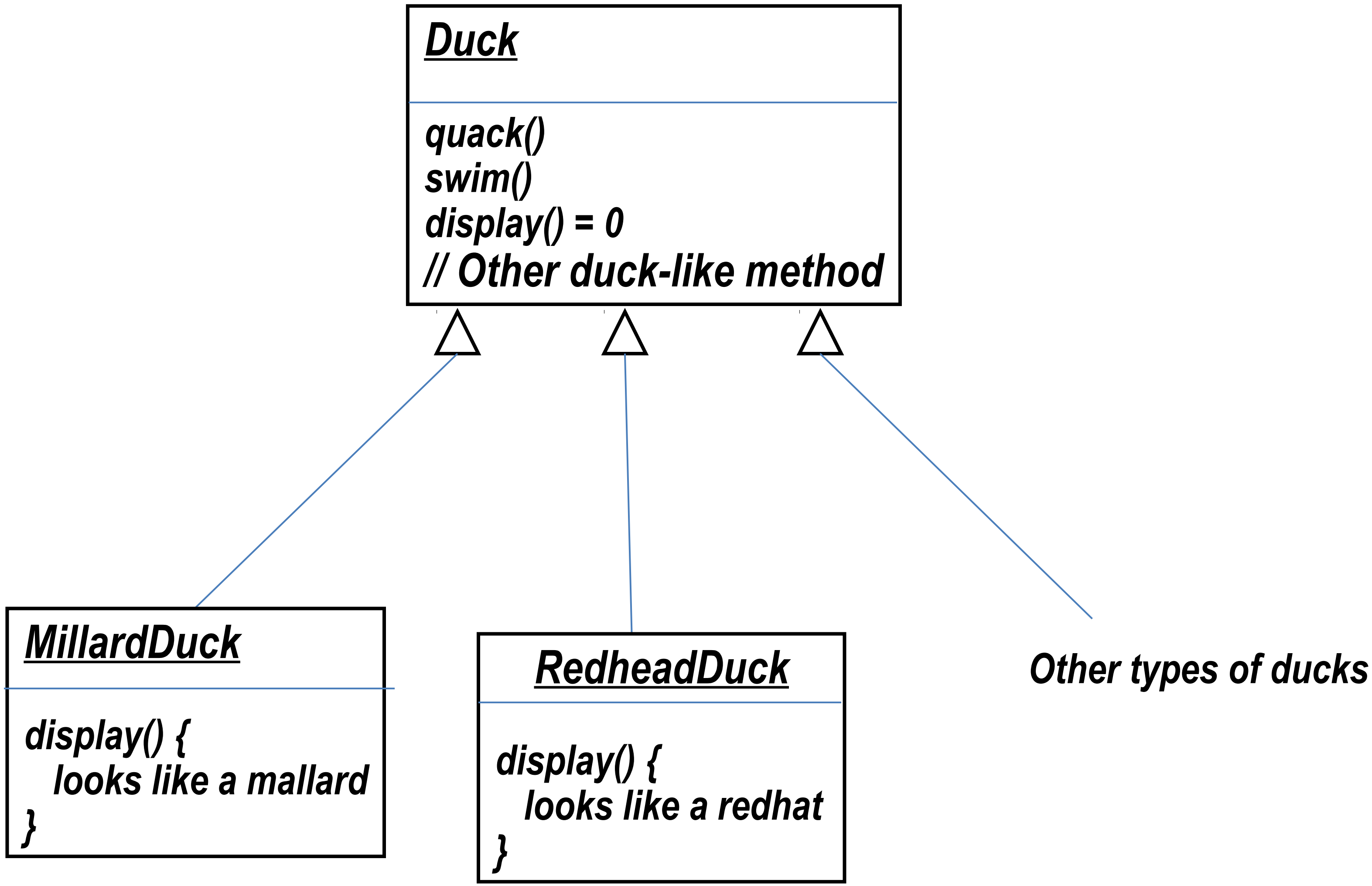
Michael J. Fischer
(from slides by Y. Richard Yang)

Lecture 23b
November 29, 2011

Example: Duck Game

- A startup produces a duck-pond simulation game
- The game shows a large variety of duck species swimming and making quacking sounds

Initial Design



Design Change: add fly()

Duck

quack()
swim()
display() = 0
fly()
// Other duck-like method

MillardDuck

display() {
looks like a mallard
}

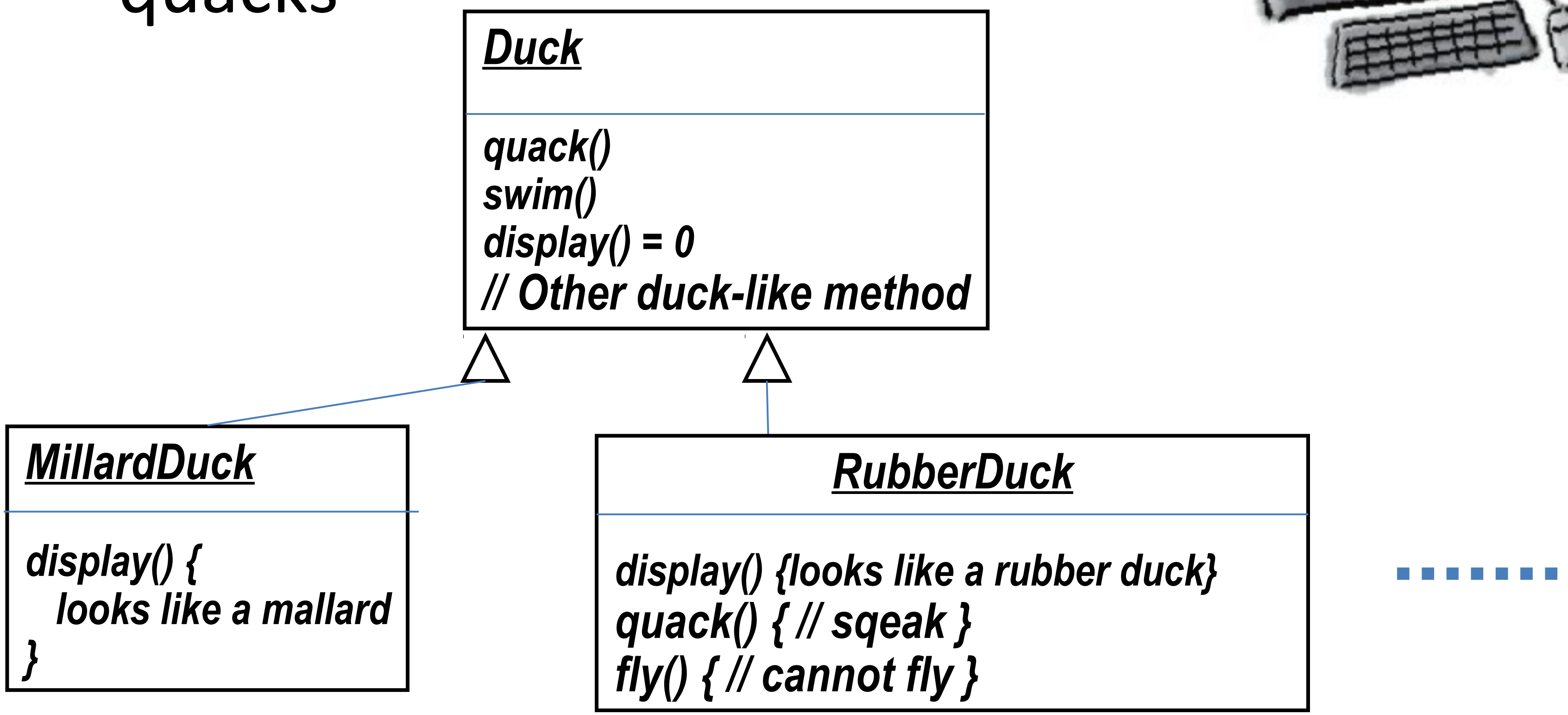
RedheadDuck

display() {
looks like a redhat
}

Other types of ducks

Problem

- Generalization may lead to unintended behaviors:
  a rubber duck is flying and quacks



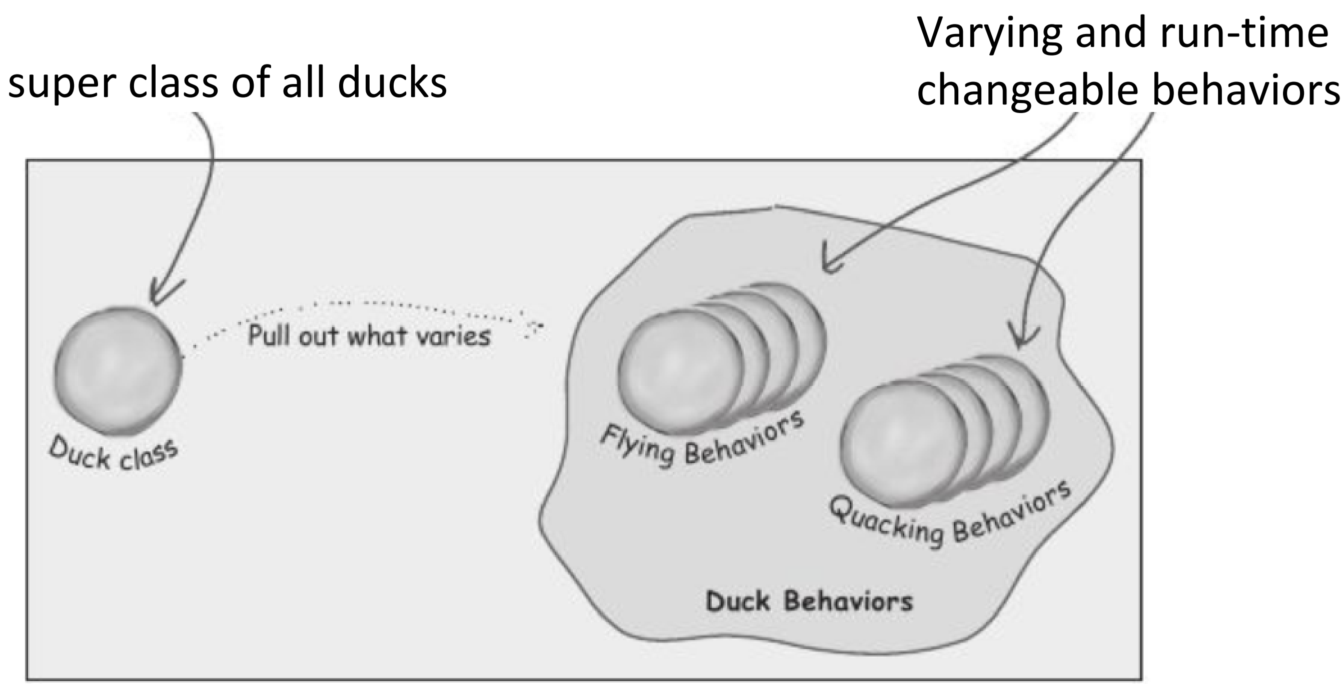
Anticipating Changes

- Identify the aspects of your application that may vary
  - What may change?

- Anticipate that
  - new types of ducks may appear and
  - behaviors (quack, swimming, and flying) may change, even change at run time (swirl flying, circular flying, …)

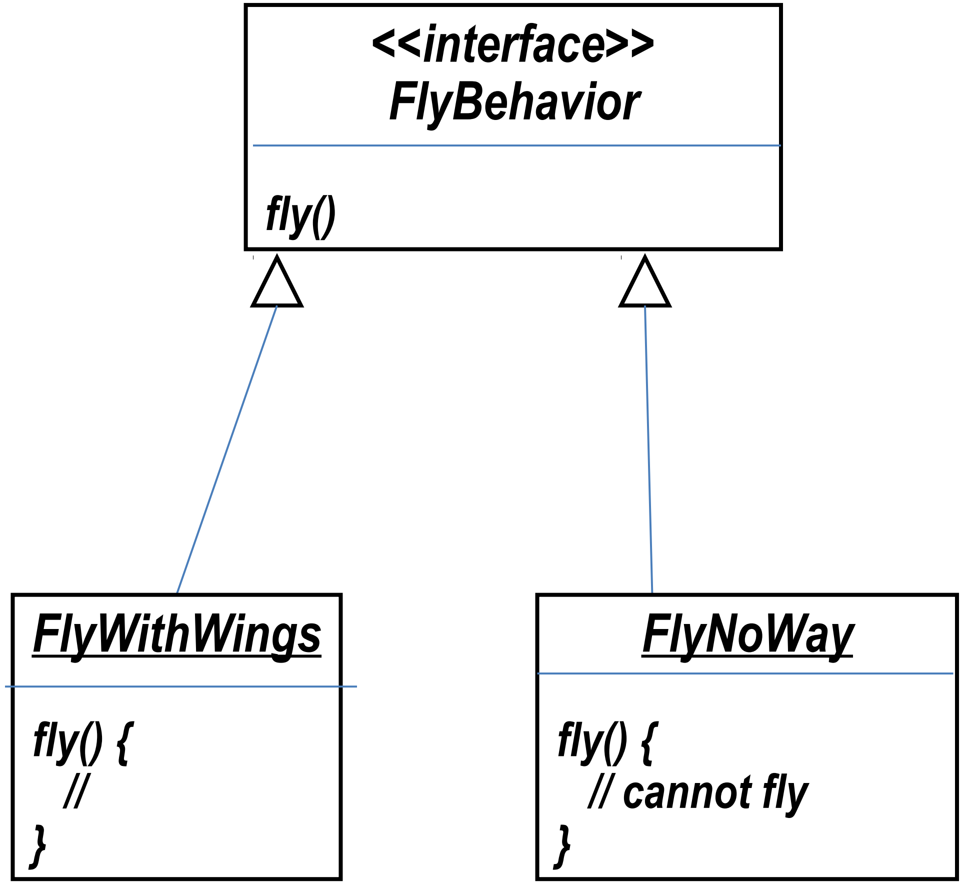
Handling Varying Behaviors

- Solution: take what varies and "encapsulate" it
  - Since fly() and quack() vary across ducks, separate these behaviors from the Duck class and create a new set of classes to represent each behavior



Design

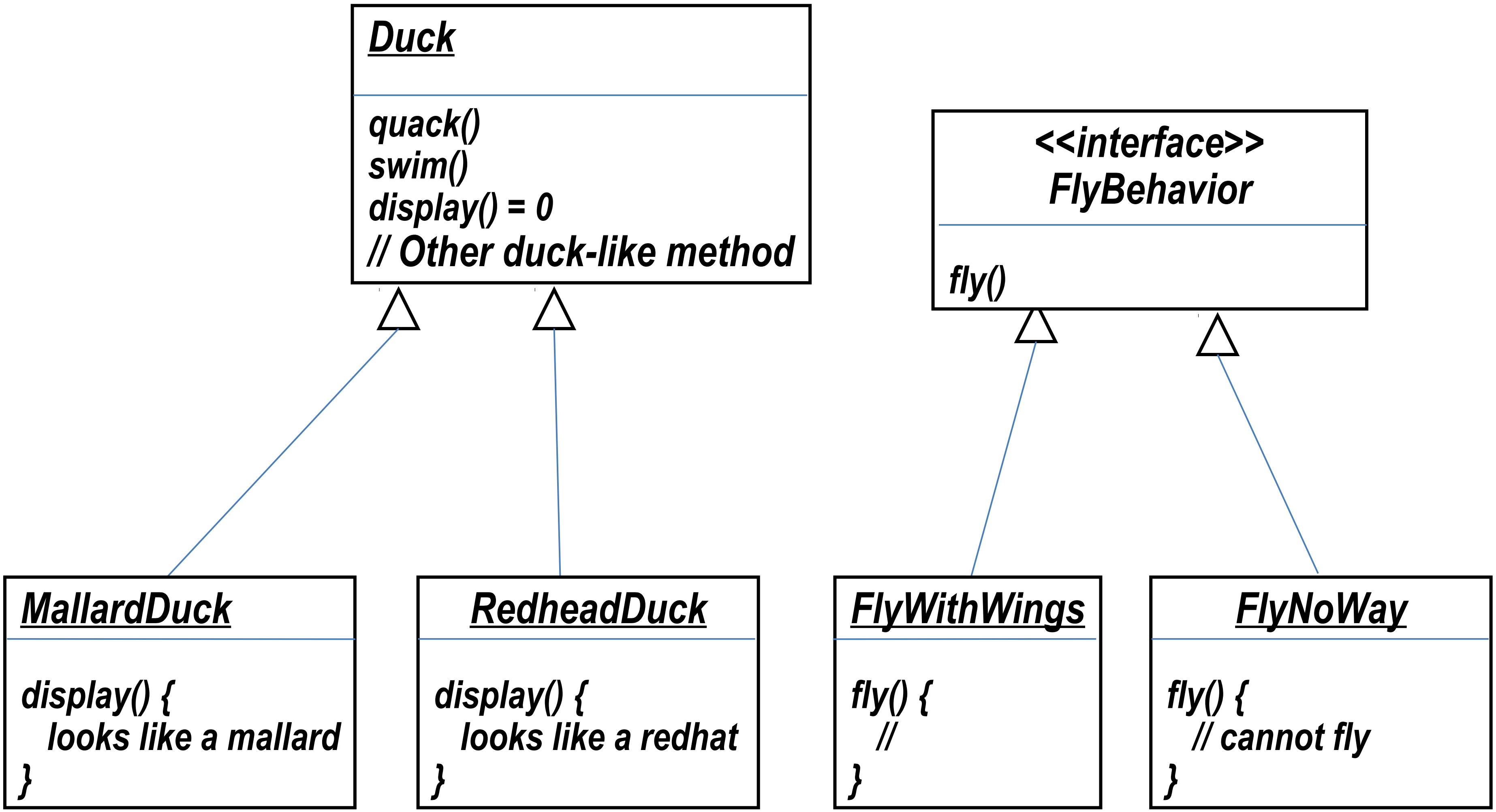
- Each duck object has a fly behavior



Programming to implementation vs. interface/supertype

- Programming to an implementation
  - Dog d = new Dog();
  - d.bark();
- Programming to an interface/supertype
  - Animal a = new Dog();
  - a.makeSound();

Implementation



Exercise

- Add rocket-powered flying?

The Strategy Pattern

- Defines a set of algorithms, encapsulates each one, and makes them interchangeable by defining a common interface

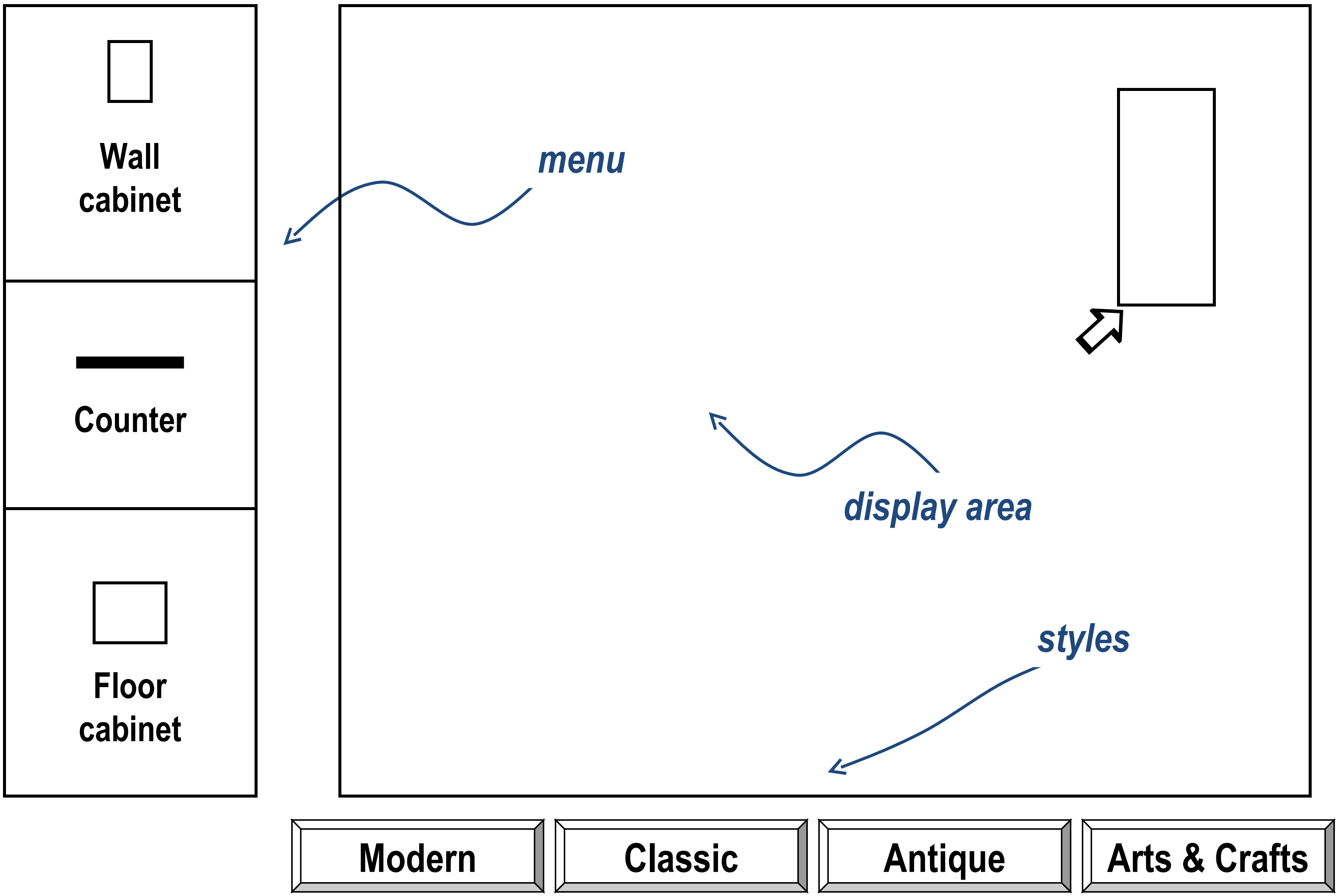
Exercise


Character
WeaponBehavior weapon;
fight();
Queen
fight() { ... }
King
fight() { ... }
Troll
fight() { ... }
Knight
fight() { ... }
KnifeBehavior
useWeapon() { // implements cutting with a knife }
BowAndArrowBehavior
<<interface>>
WeaponBehavior
useWeapon();
AxeBehavior
useWeapon() { // implements chopping with an axe }
SwordBehavior
useWeapon() { // implements swinging a sword }
setWeapon(WeaponBehavior w) {
this.weapon = w;
}

Summary: Design Principles

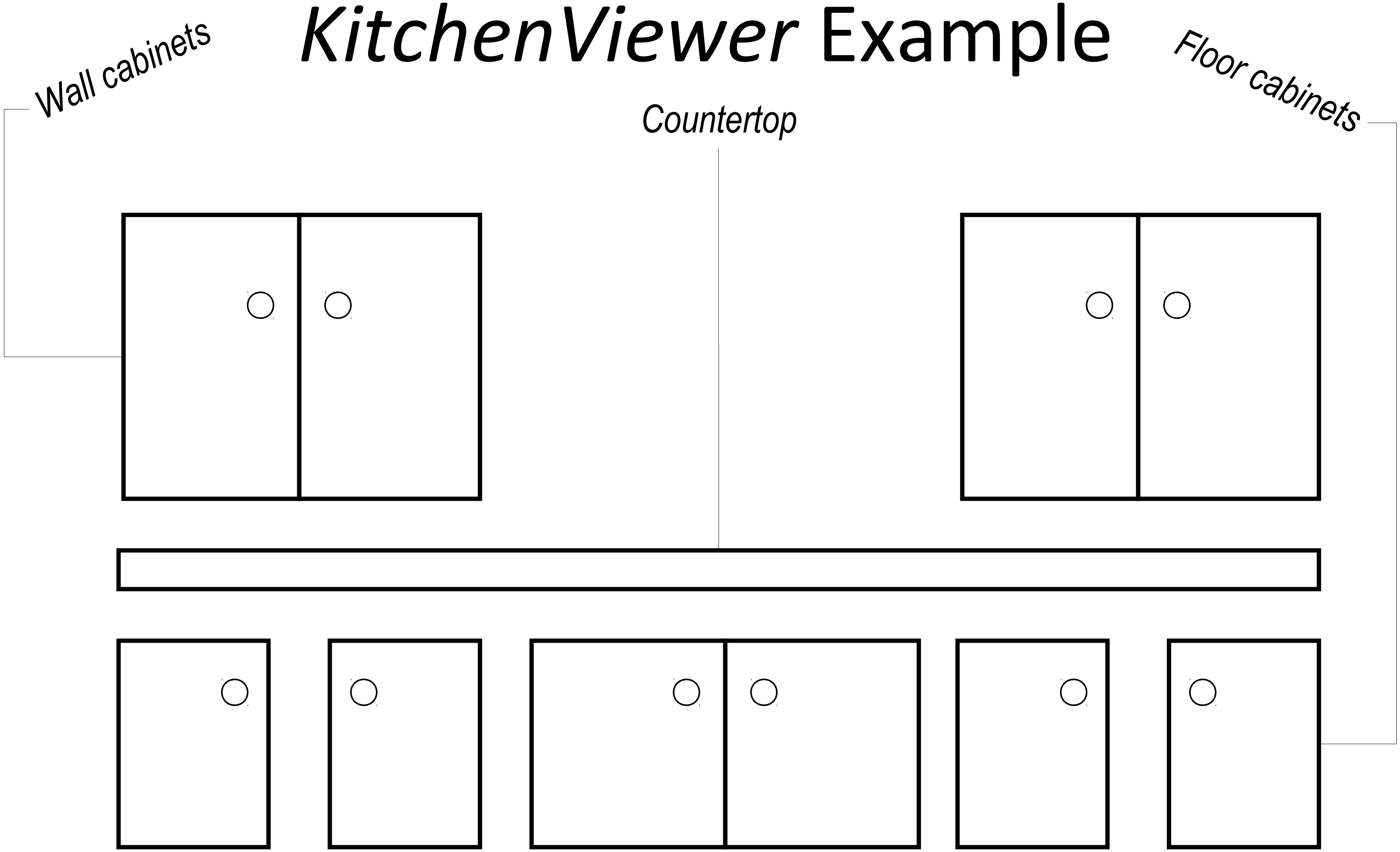
- Identify the aspects of your application that vary and separate them from what stay the same
- Program to an interface not implementation
- Favor composition over inheritance

Example: KitchenViewer Interface



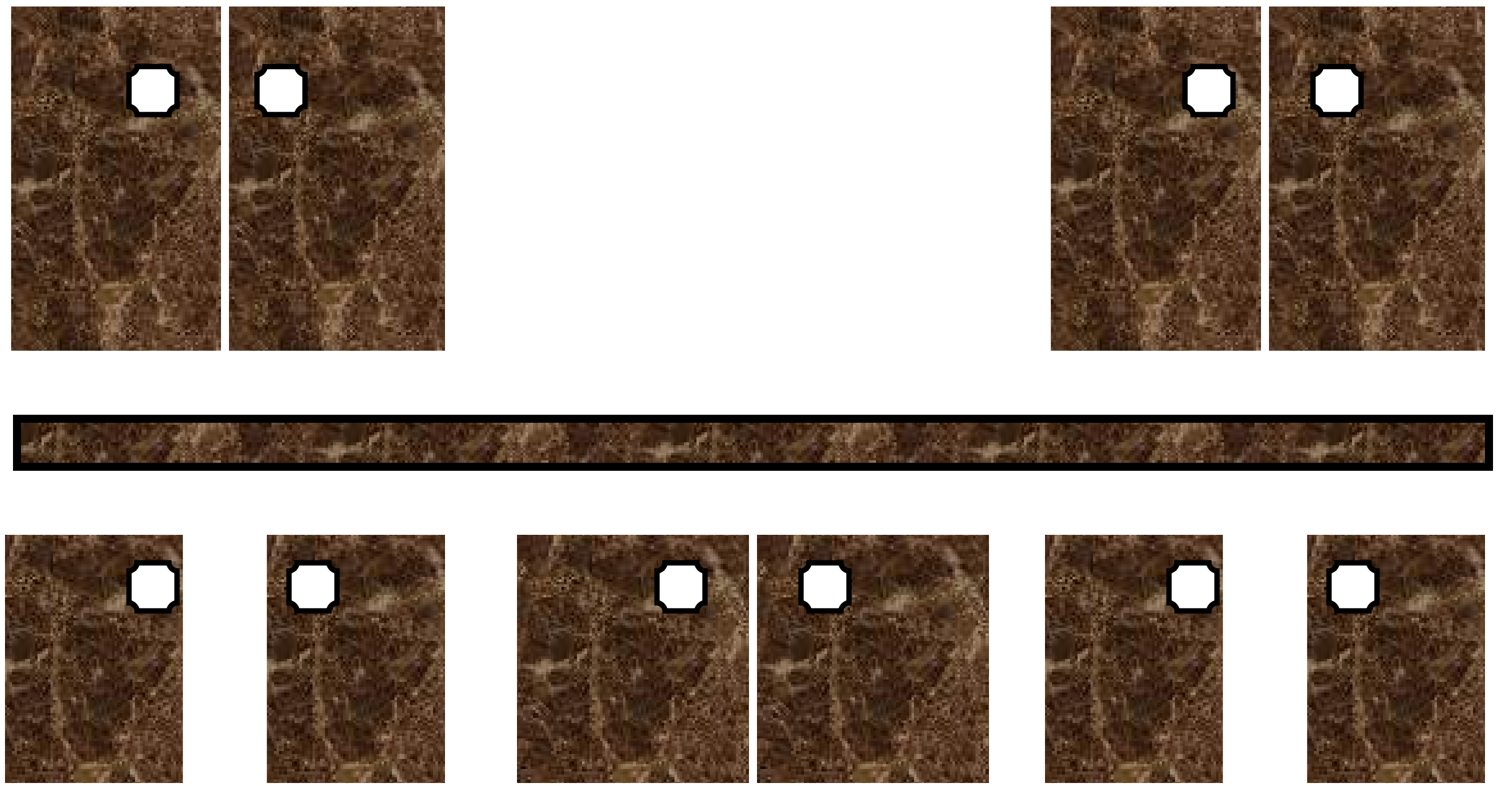
Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

KitchenViewer Example



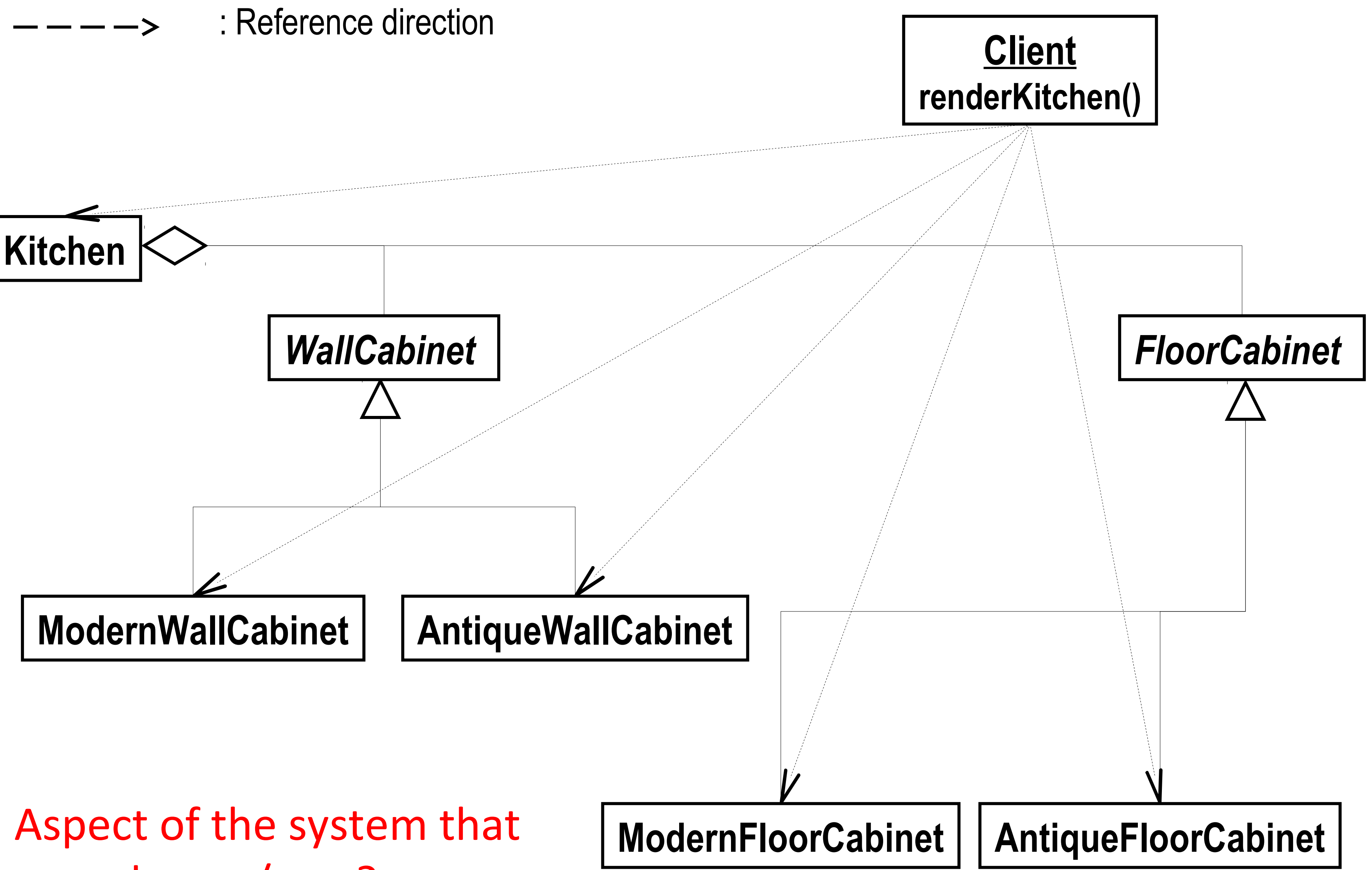
Modern | Classic | Antique | Arts & Crafts

Selecting *Antique* Style



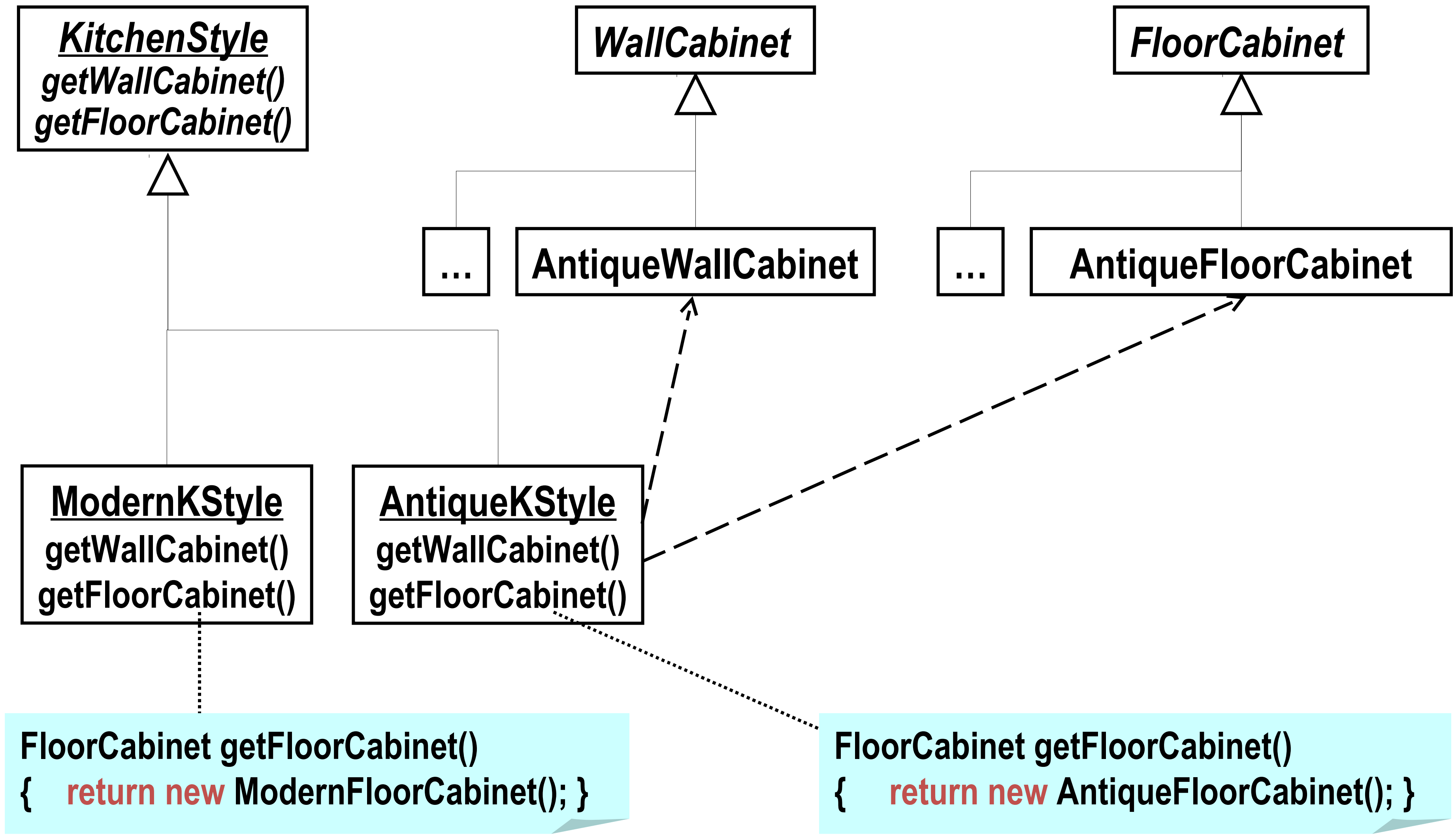
Modern | Classic | **Antique** | Arts & Crafts

KitchenViewer Using Standard Inheritance

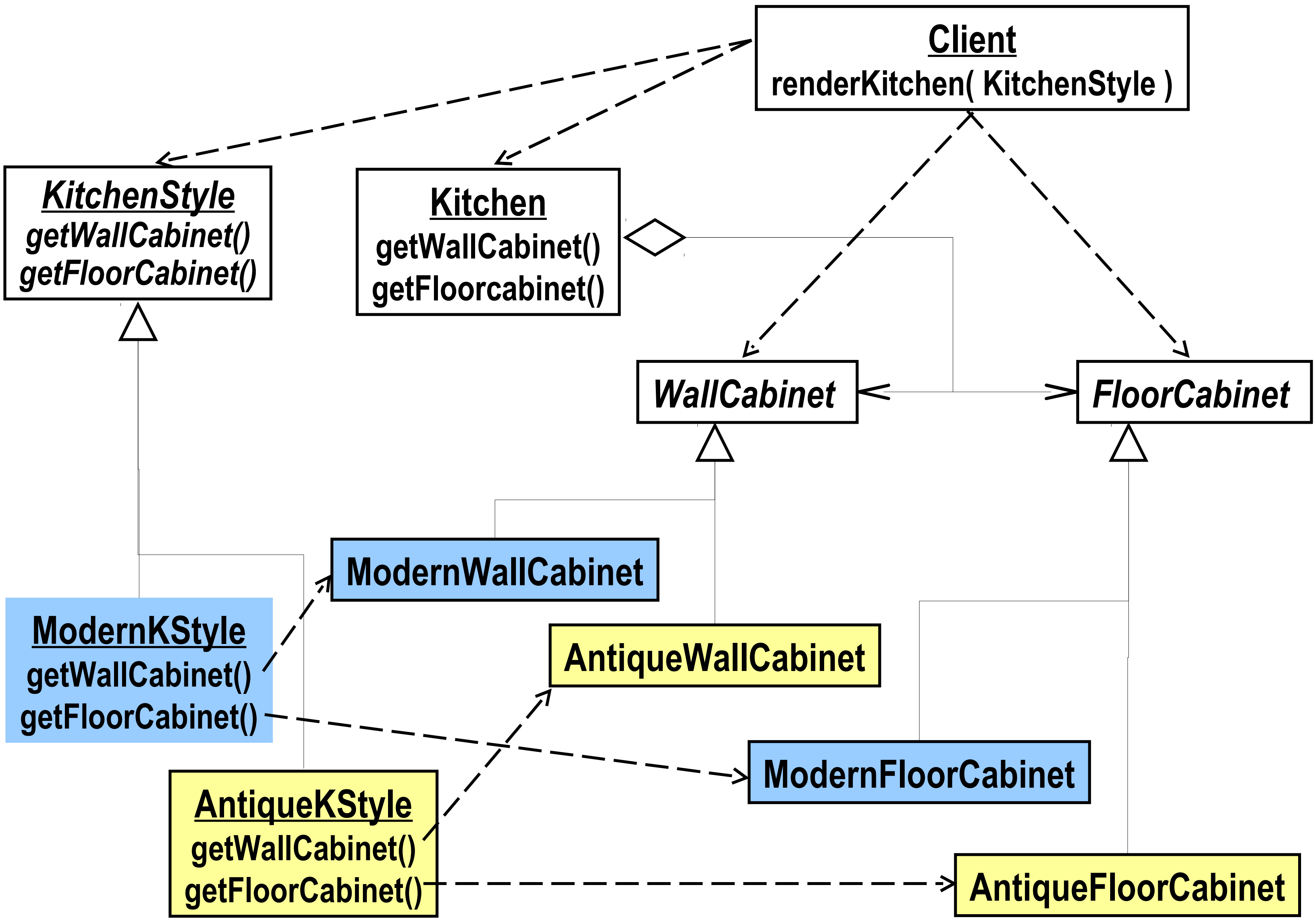


Aspect of the system that may change/vary?

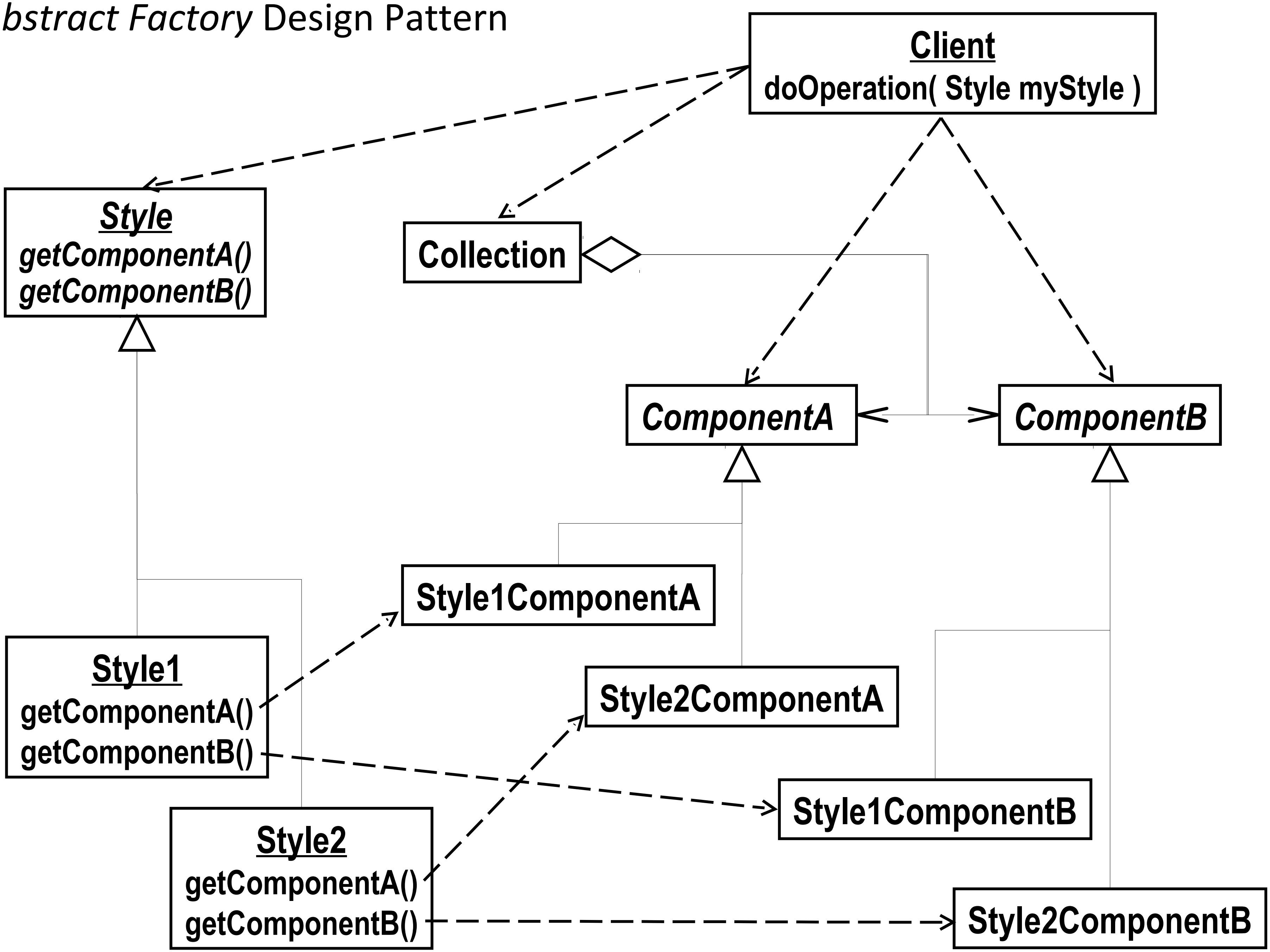
The Abstract Factory Idea



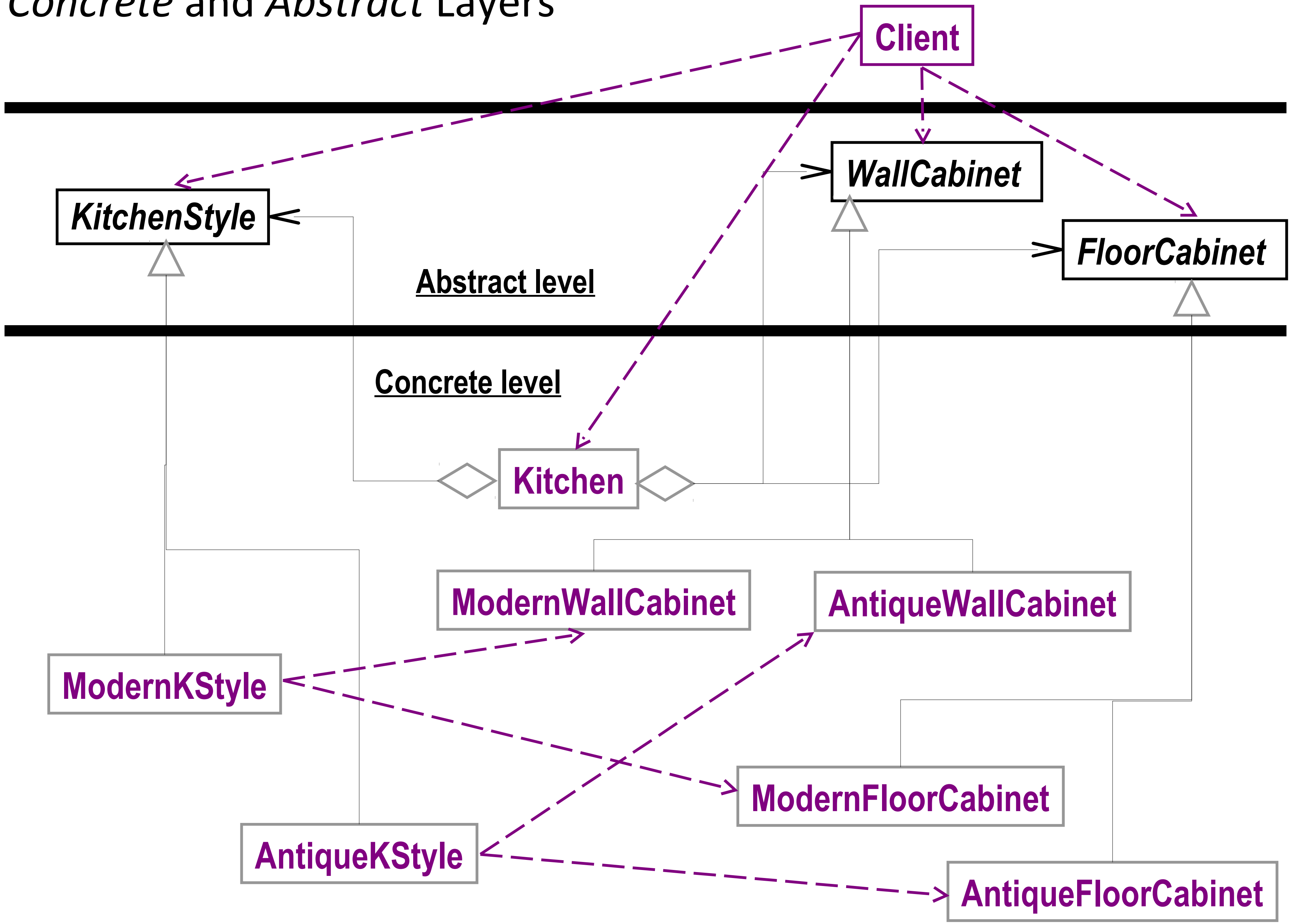
Abstract Factory Design Pattern Applied to *KitchenViewer*



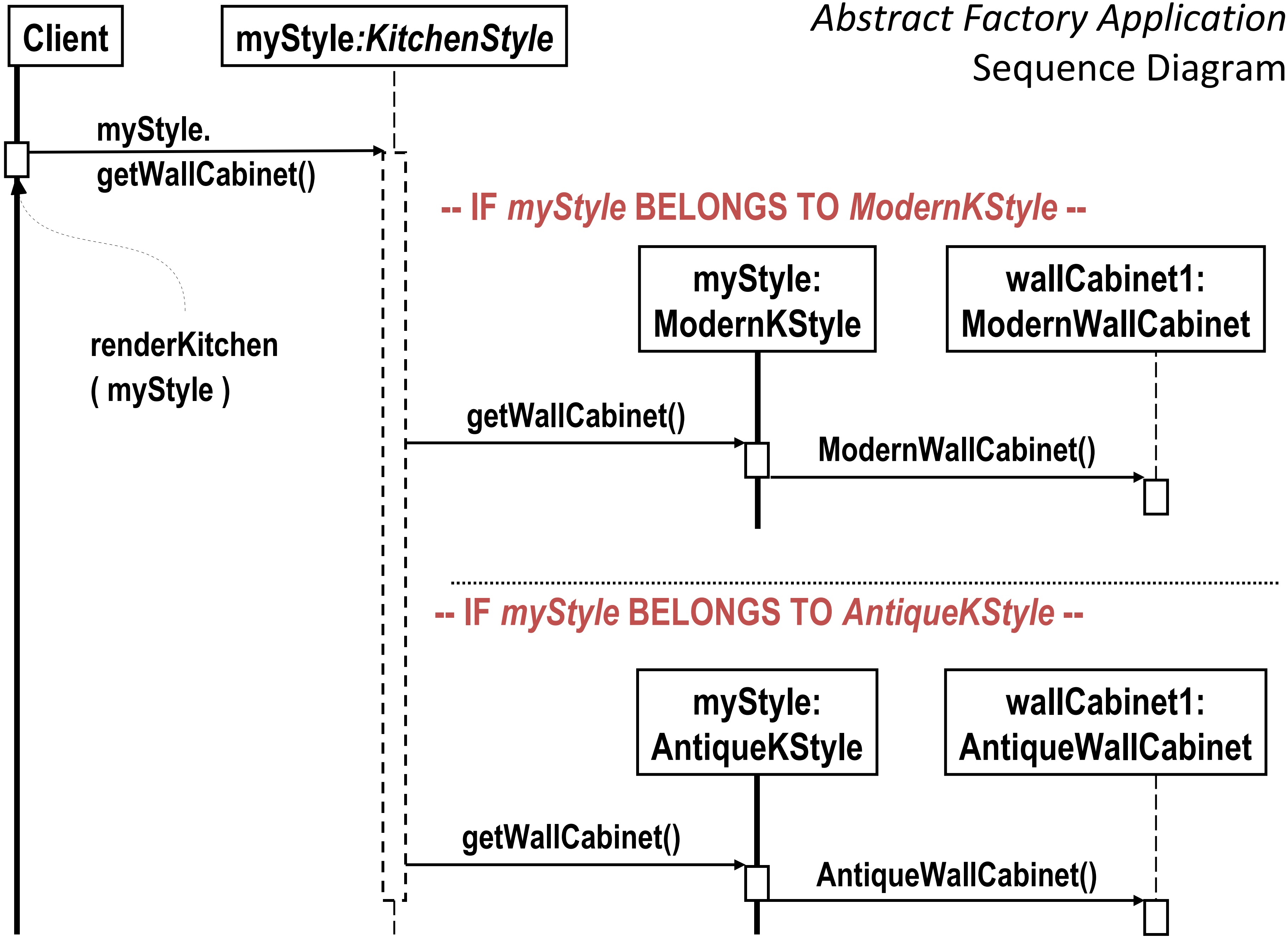
Abstract Factory Design Pattern



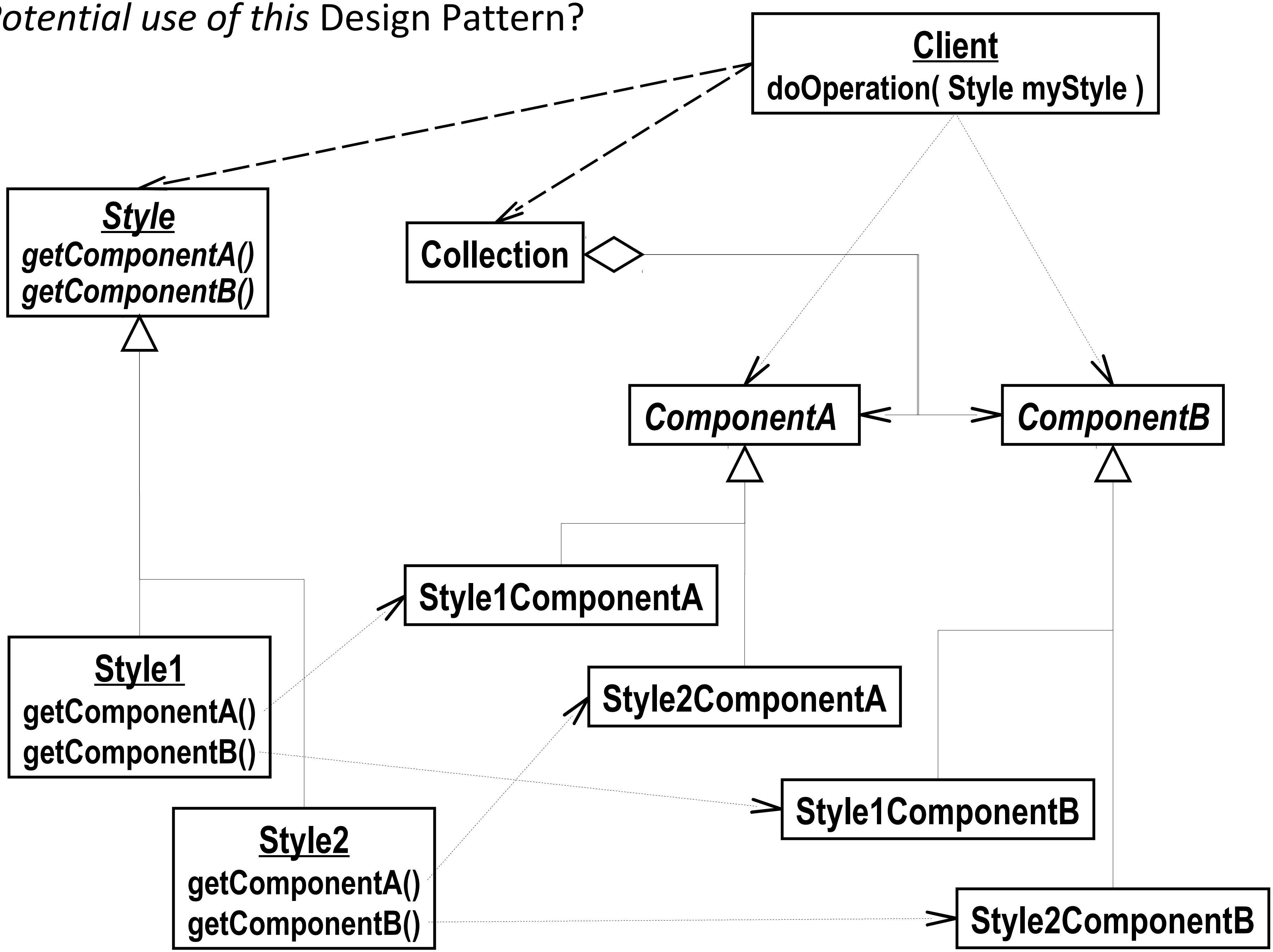
Concrete and *Abstract* Layers



Abstract Factory Application
Sequence Diagram

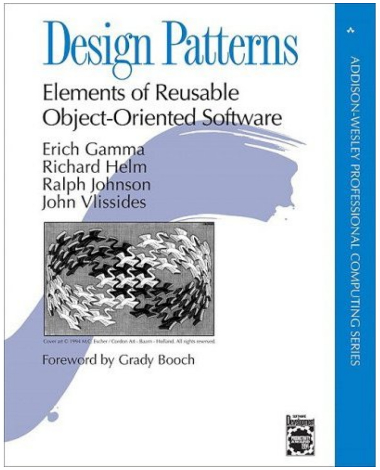


Potential use of this Design Pattern?



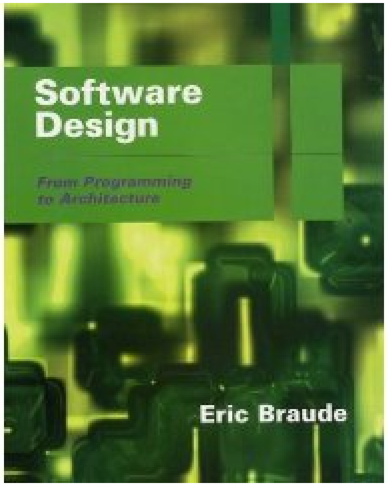
References

- Design Patterns



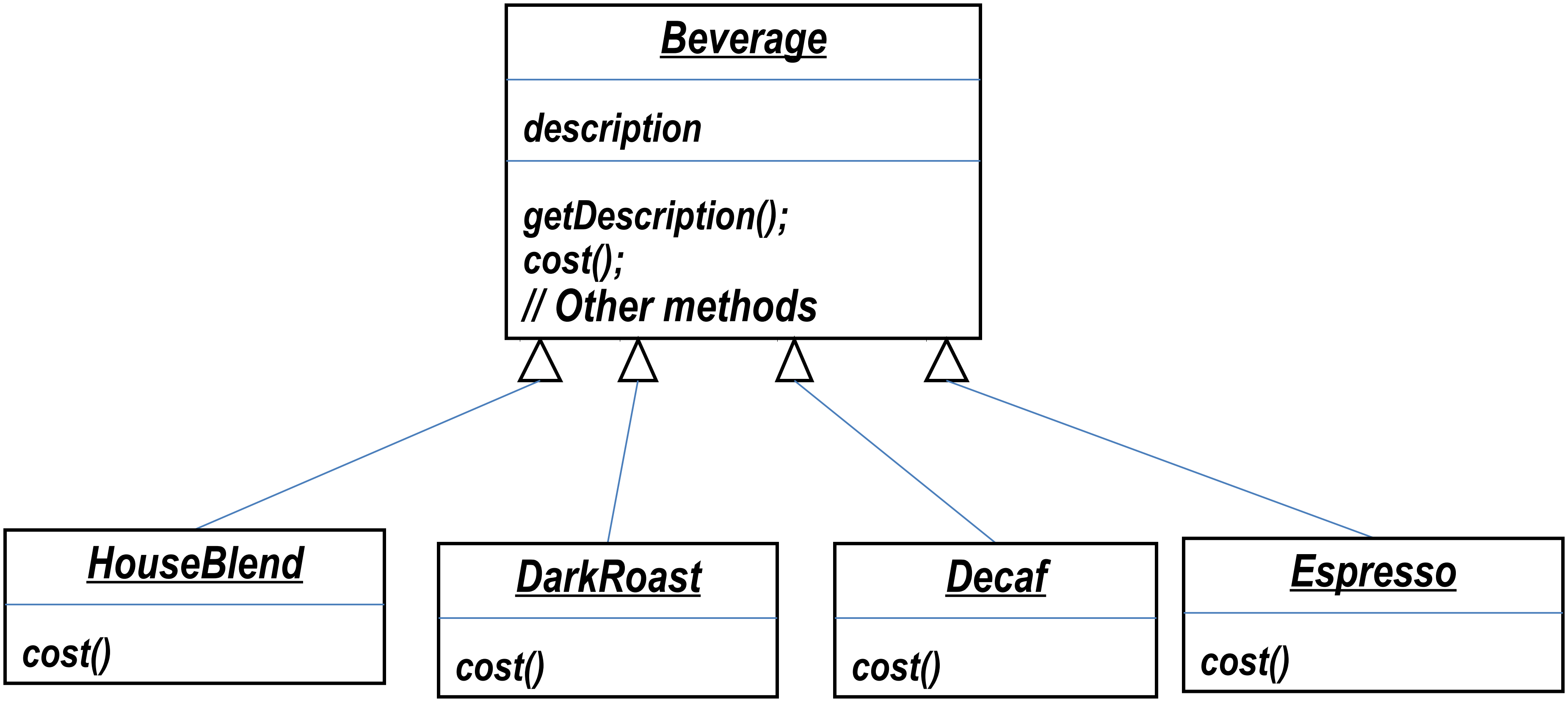
- Headfirst Design Patterns

- Software Design

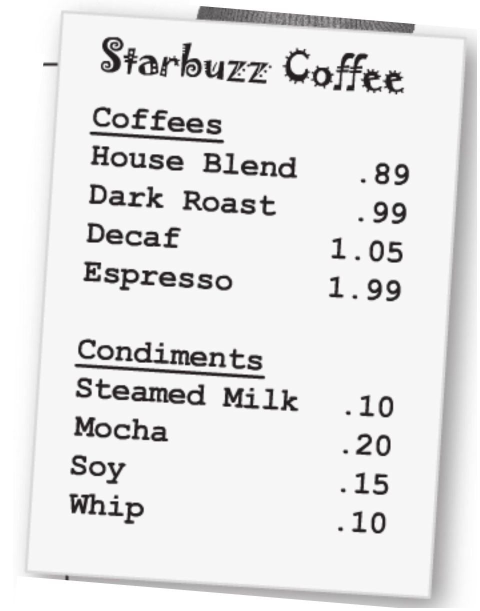


Example: Starbuzz Coffee

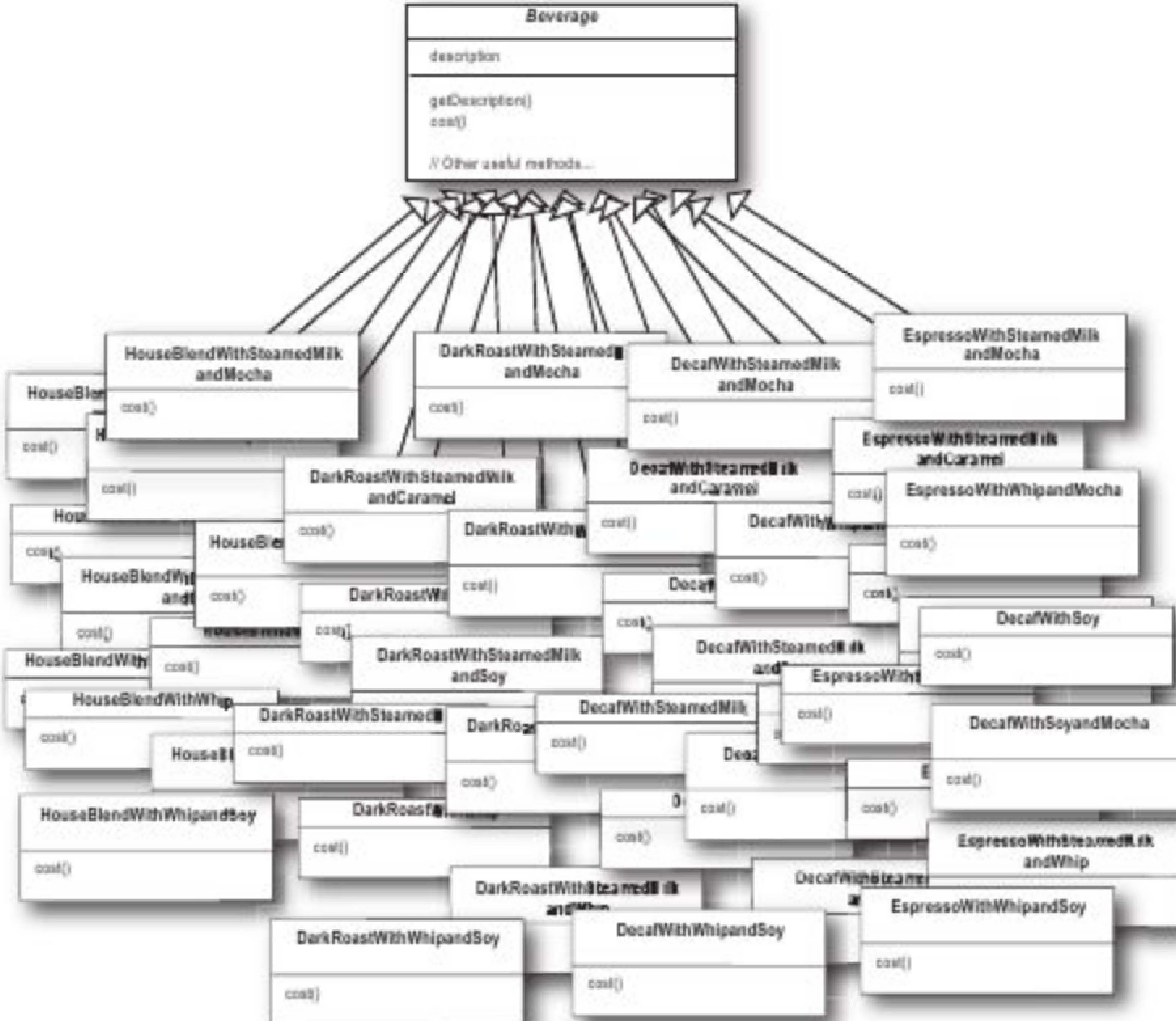
- The coffee shop offers a variety of beverages



Problem

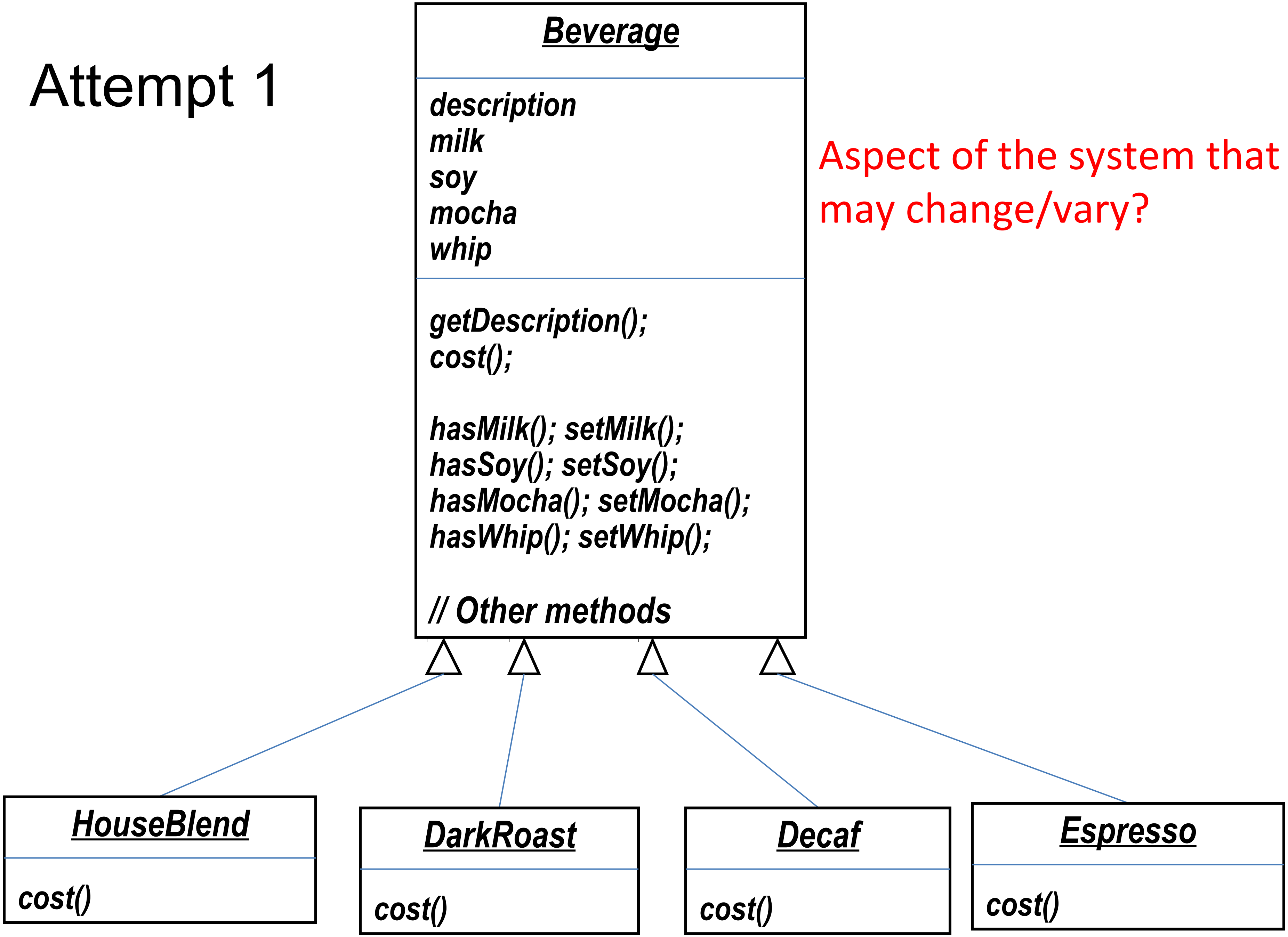


- A customer may also ask for condiments
  - steamed milk
  - soy
  - mocha (otherwise known as chocolate)
  - whipped milk
- Starbuzz charges a bit for each of these



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

Attempt 1



Aspect of the system that may change/vary?

Potential Changes

- Potential changes:
  - Price change to condiments
  - New condiments
  - Double moca
  - …

Design idea

- Basic idea: extension at run time, not compile time

- Definition: The Decorator pattern attaches additional features to an object dynamically. It provides a flexible alternative to subclassing for extending functionality

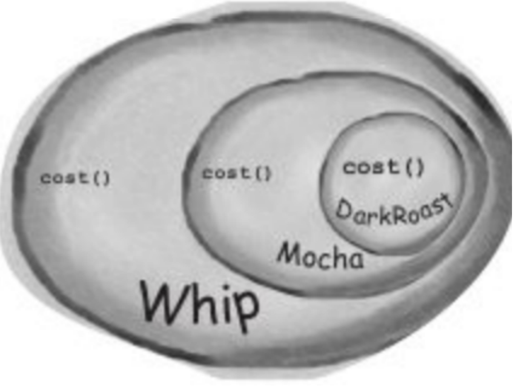
Design approach 1

- Each beverage contains a dynamic list of condiments
- Example
  - Take a DarkRoast object
  - Decorate it with a Mocha object
  - Decorate it with a Whip object

UML class model?

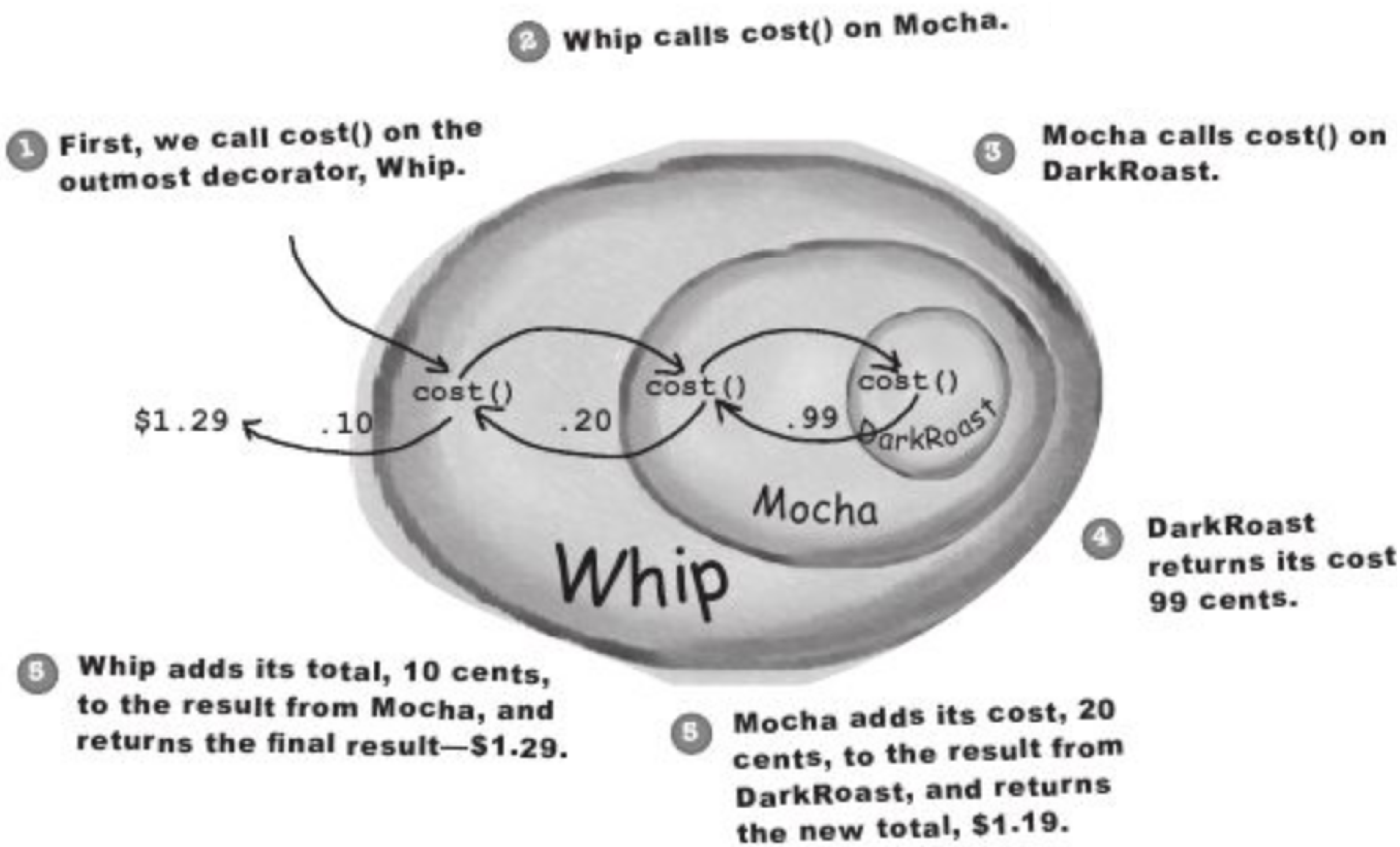
Decorator design

- Example
  - Take a DarkRoast object
  - Decorate it with a Mocha object
  - Decorate it with a Whip object
  - Call the cost() method and rely on delegation to add on the condiment cost

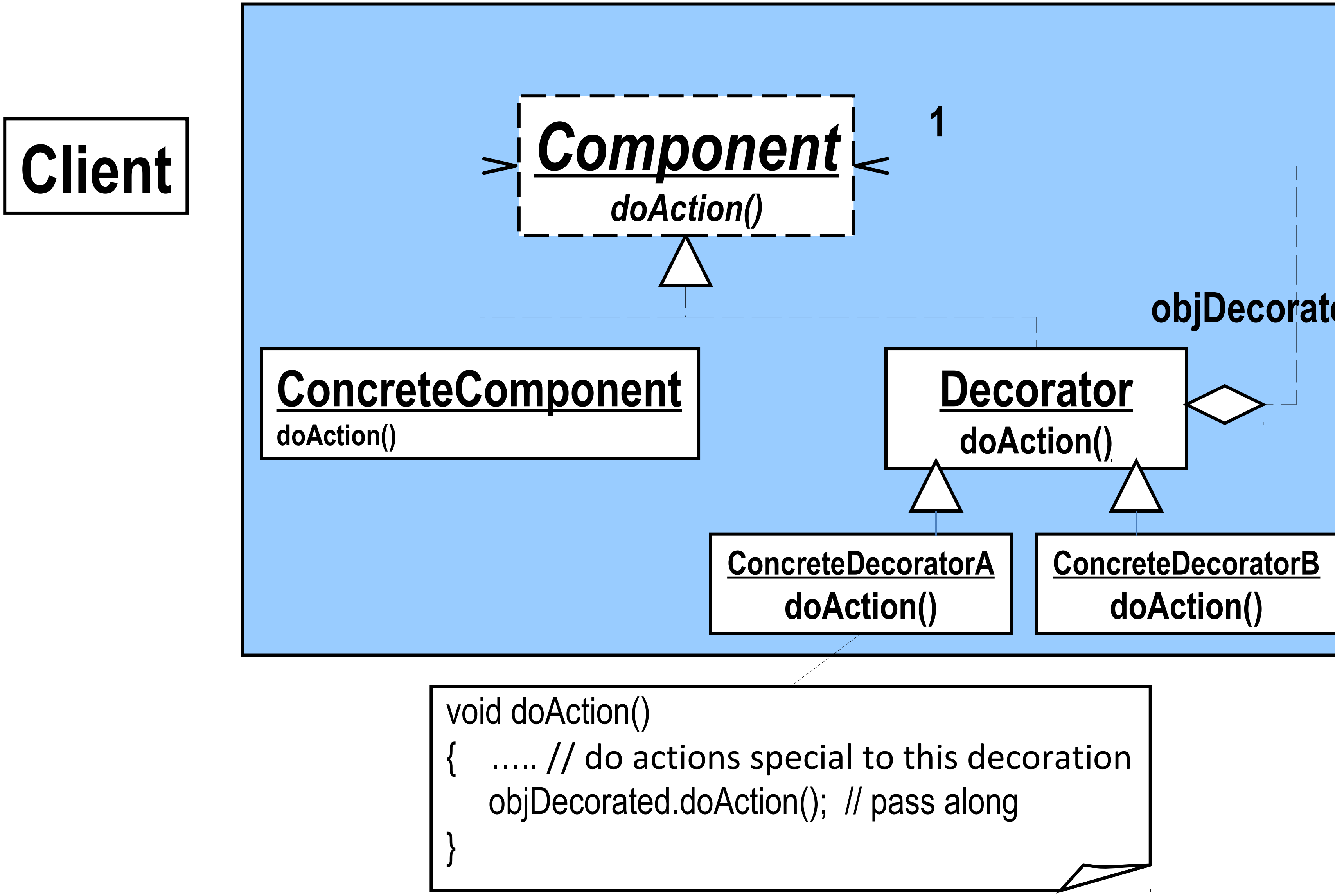


- Decorator adds its own behavior before or after calling the decorated object

Decoration Delegation Process

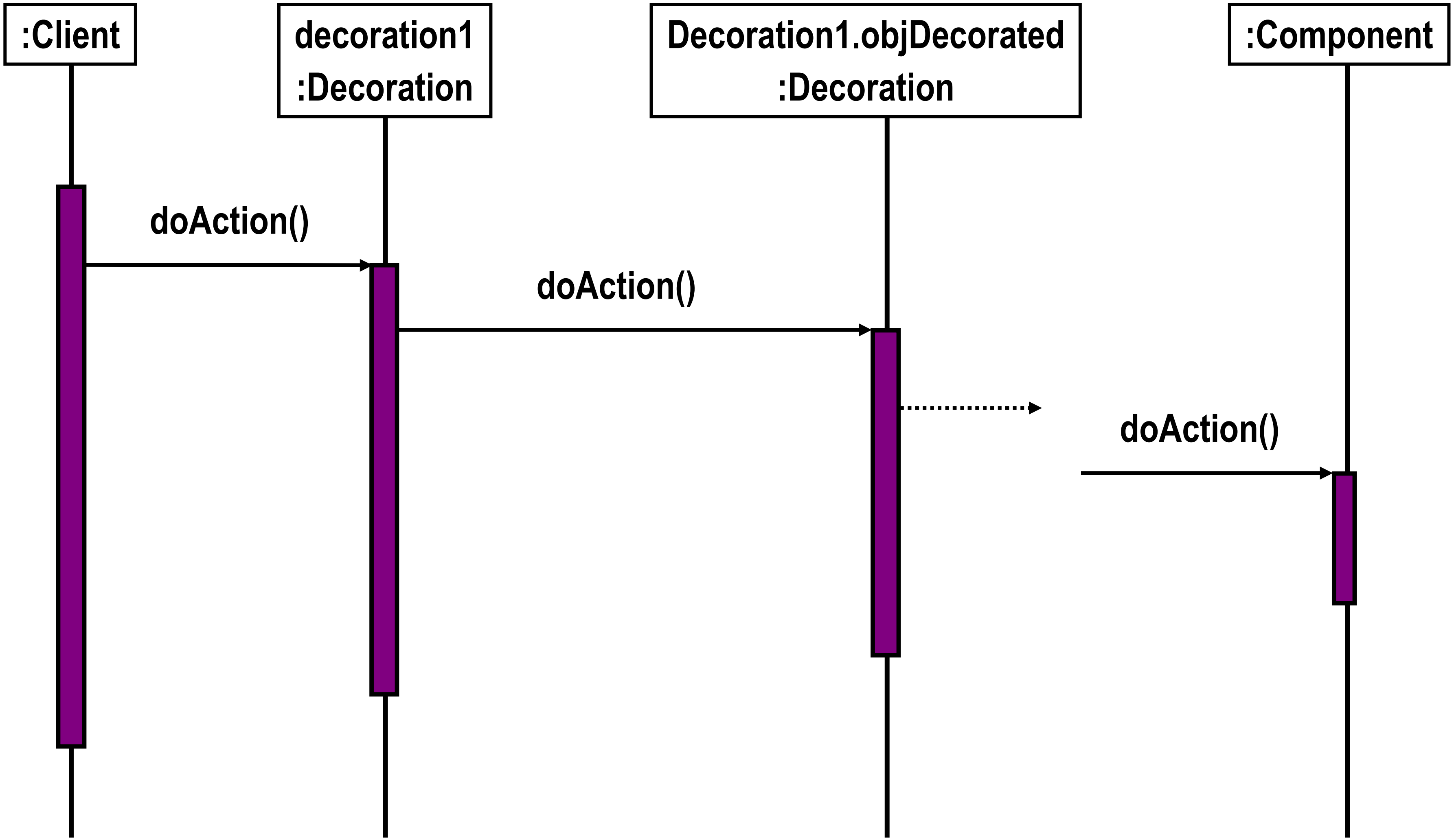


Decorator Class



Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Sequence Diagram for *Decorator*



Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Decoration Features

- Decorators have the same supertype as the objects they decorate
- You can use one or more decorators to wrap an object
  - Thus, you can pass decorated object in place of original (wrapped) object
- The decorator adds its own behavior either before or after delegating to the object it decorates to
- Objects can be decorated at any time, including run-time, with as many decorators as possible

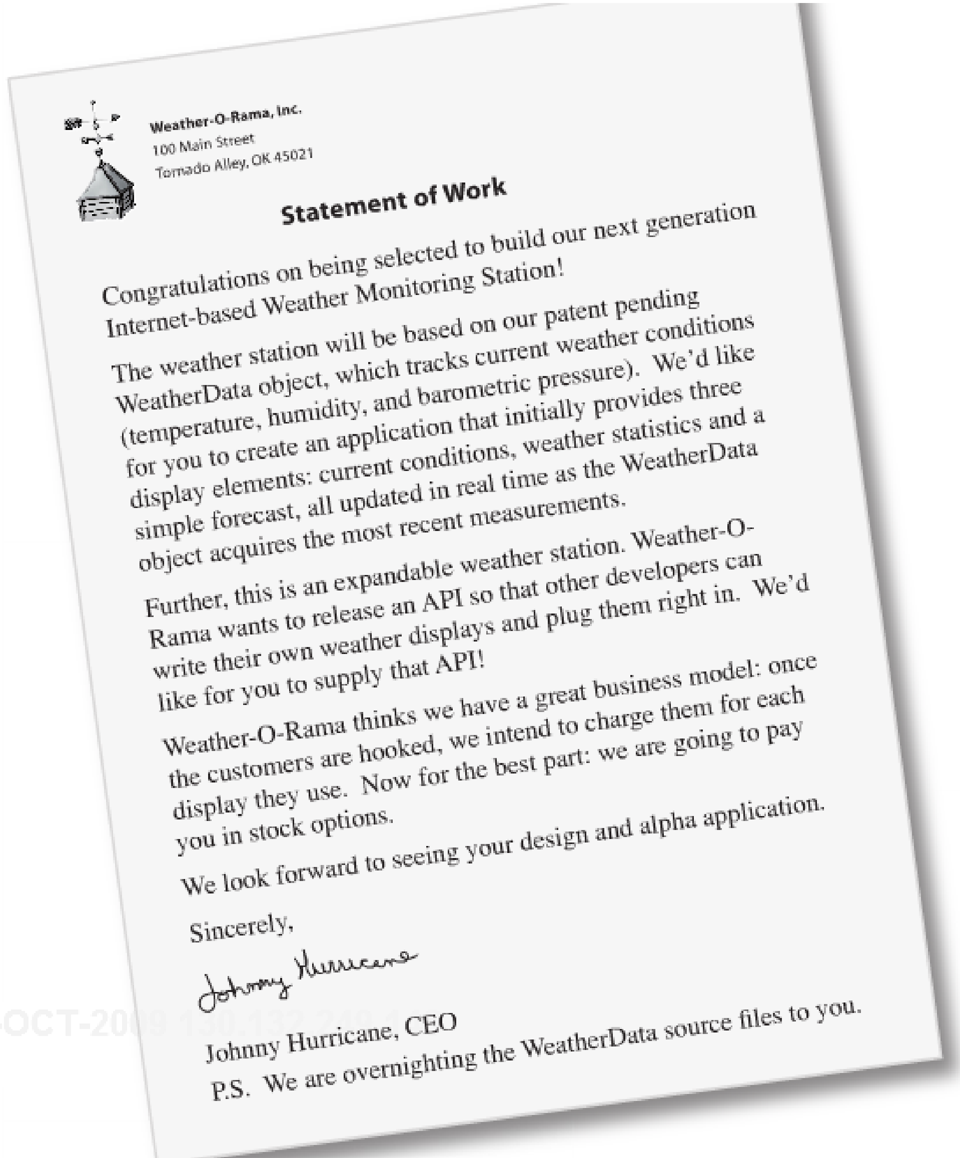
Exercise

- Suppose we allow different sizes for the beverages
  - Tall (small)
  - Grande (medium)
  - Venti (large)

Some Common Design Patterns

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Example: Weather-O-Rama



Weather-O-Rama, Inc.
100 Main Street
Tornado Alley, OK 45021

Statement of Work

Congratulations on being selected to build our next generation Internet-based Weather Monitoring Station!

The weather station will be based on our patent pending WeatherData object, which tracks current weather conditions (temperature, humidity, and barometric pressure). We'd like for you to create an application that initially provides three display elements: current conditions, weather statistics and a simple forecast, all updated in real time as the WeatherData object acquires the most recent measurements.

Further, this is an expandable weather station. Weather-O-Rama wants to release an API so that other developers can write their own weather displays and plug them right in. We'd like for you to supply that API!

Weather-O-Rama thinks we have a great business model: once the customers are hooked, we intend to charge them for each display they use. Now for the best part: we are going to pay you in stock options.

We look forward to seeing your design and alpha application.

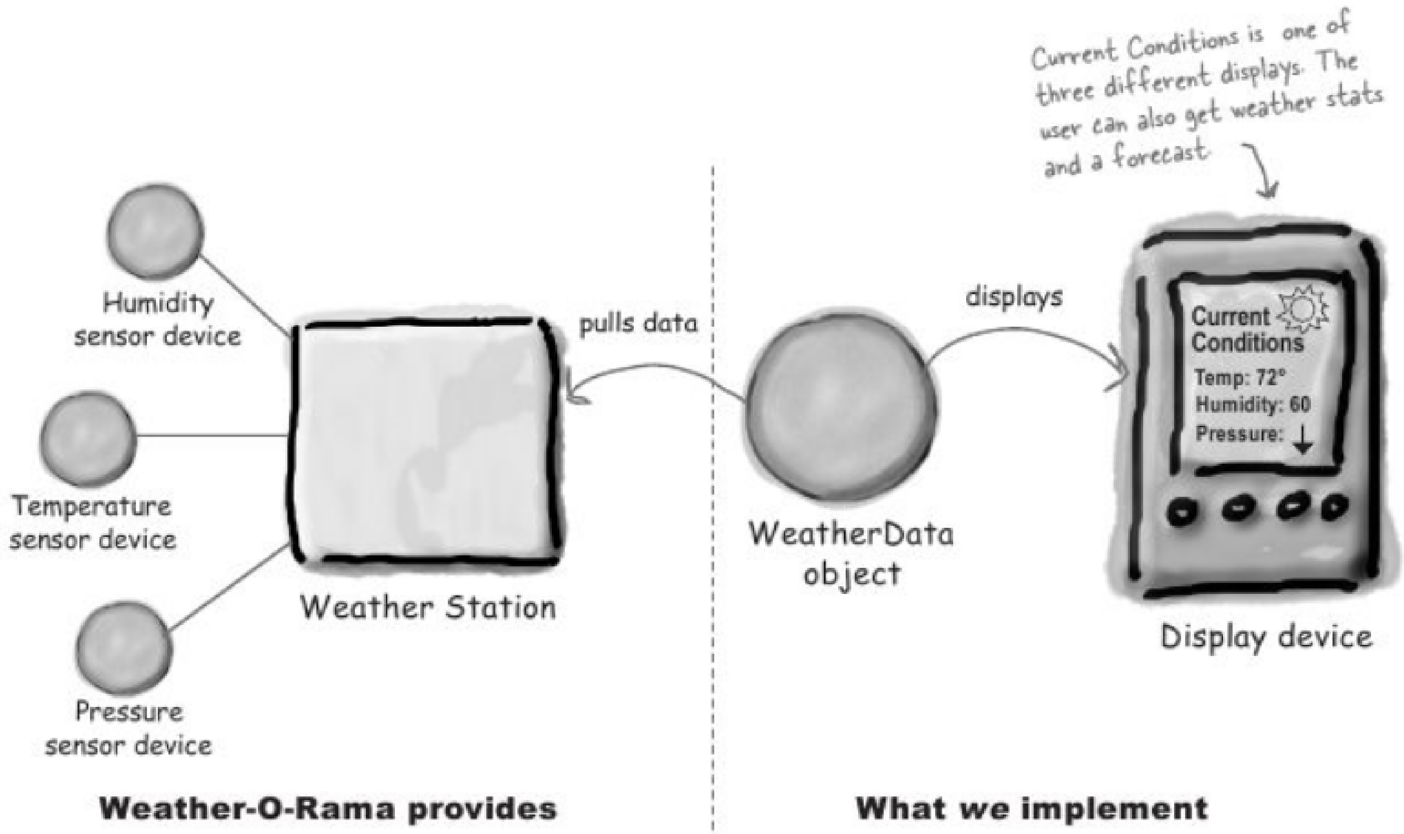
Sincerely,

Johnny Hurricane

Johnny Hurricane, CEO

P.S. We are overnighting the WeatherData source files to you.

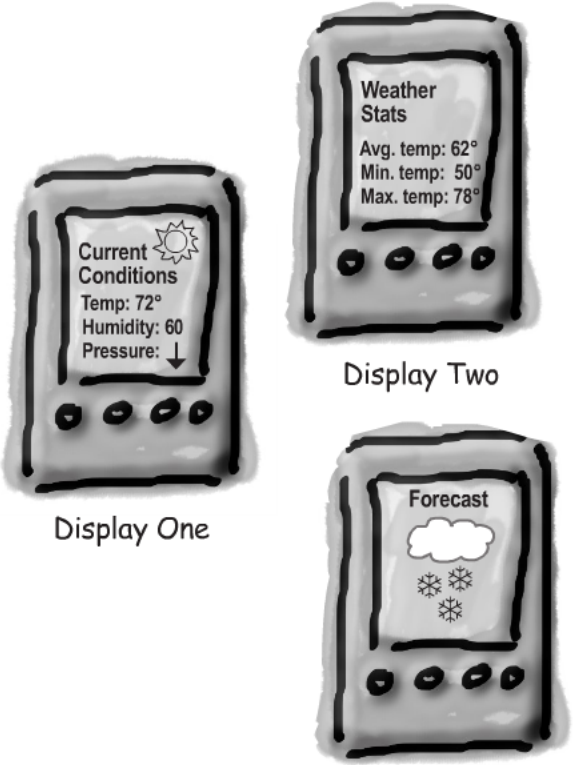
Weather-O-Rama



Weather-O-Rama Interface

WeatherData
getTemperature(); *getHumidity();* *getPressure();* *measurementsChanged();* *setMeasurements();* *// other methods*

This method gets called whenever the weather measurements have been updated.



First Implementation

```
void measurementsChanged() {

    float temp = getTemperature() ;
    float humidity = getHumidity() ;
    float pressure = getPressure() ;

    currentConditionsDisplay->update(temp, humidity,
    pressure) ;
    statisticsDisplay->update(temp, humidity, pressure) ;
    forecastDisplay->update(temp, humidity, pressure) ;
}
```

By coding to concrete implementation, we have no way of allowing other displays and plug in.

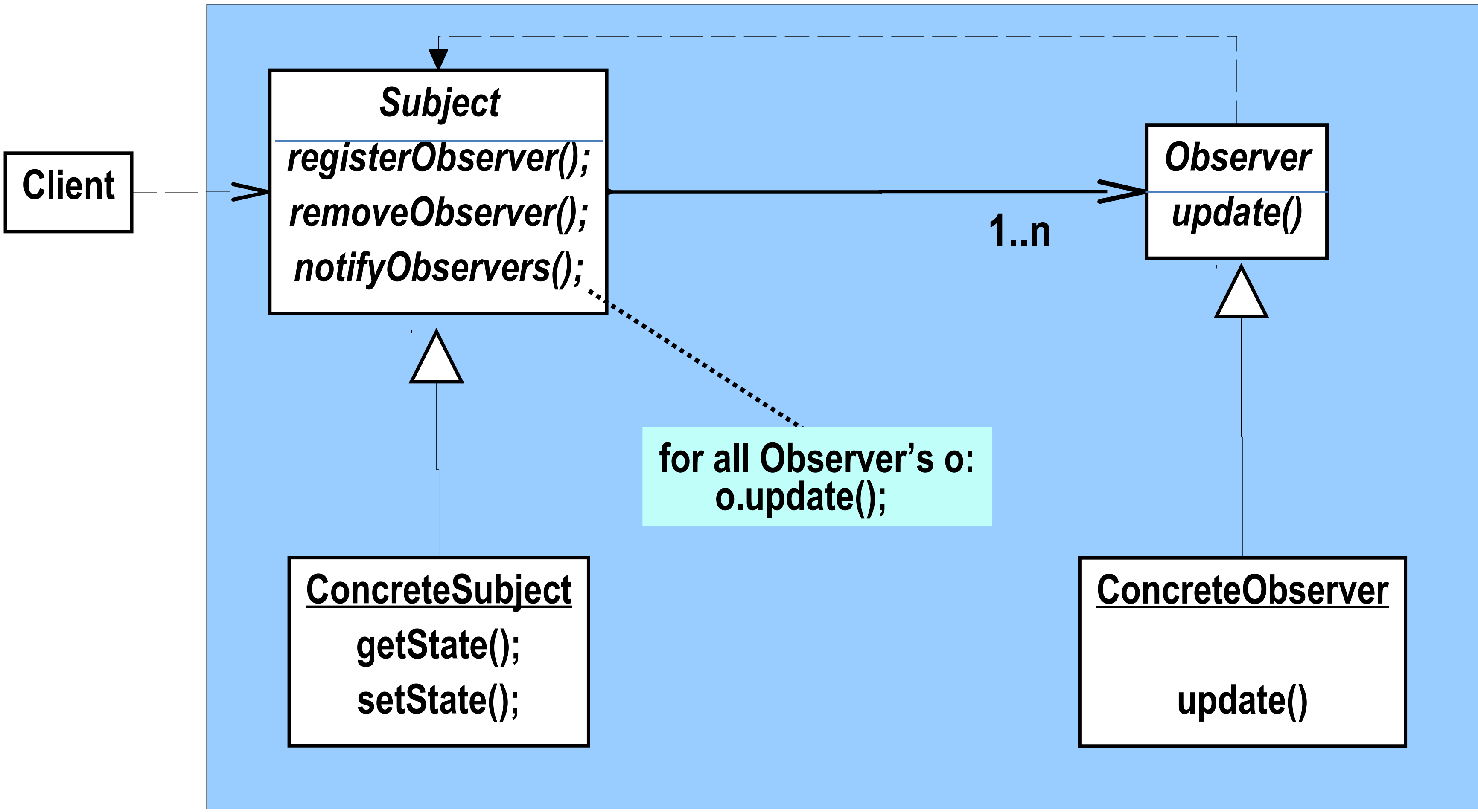
Observer Pattern

- Design Purpose: defines a run-time, one-to-many dependency between objects so that when one object (the subject) changes state, all of the dependents (observers) are notified.

Observer Design Pattern

Server part

Client part



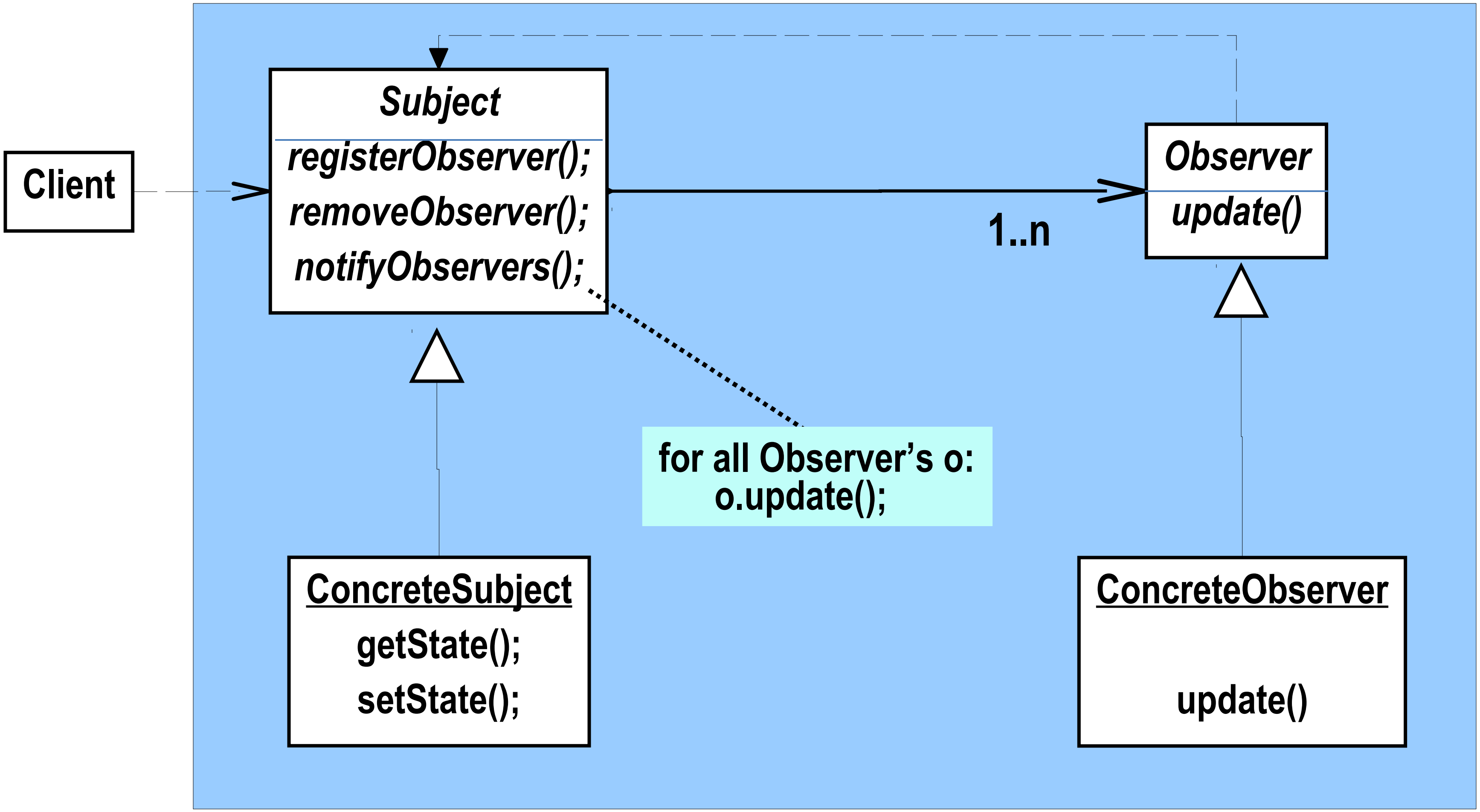
How does Observer apply these design principles?

- Identify the aspects of your application that vary and separate them from what stay the same

- Program to an interface not implementation

- Favor composition over inheritance

Discussion



- Java Observation design: update(Observable o, Object obj);