Design Patterns

Chapters 1–3 of Head First Design Patterns

Design Patterns

Someone has already solved your problems

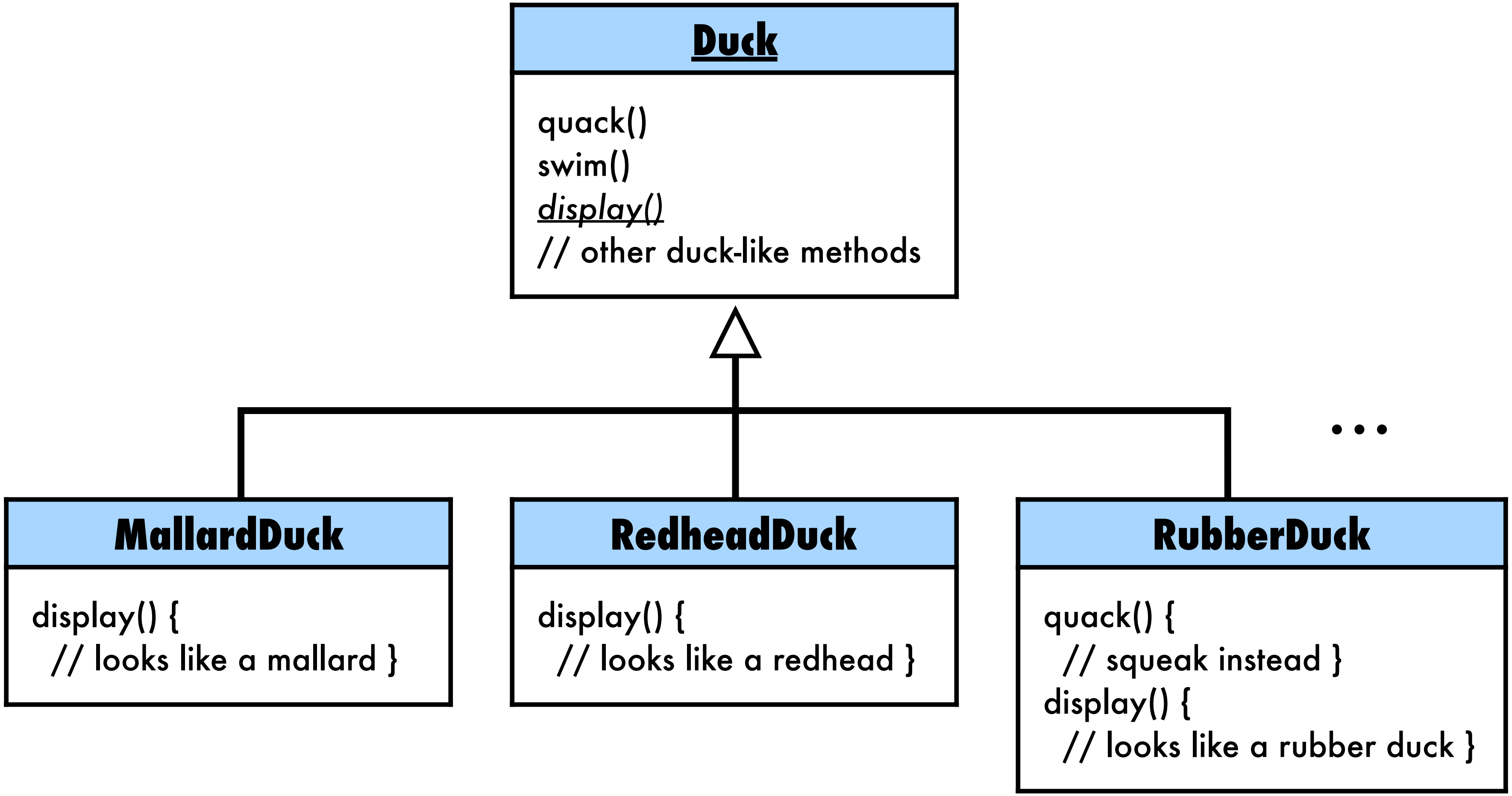
Chapter 1

Welcome to design patterns

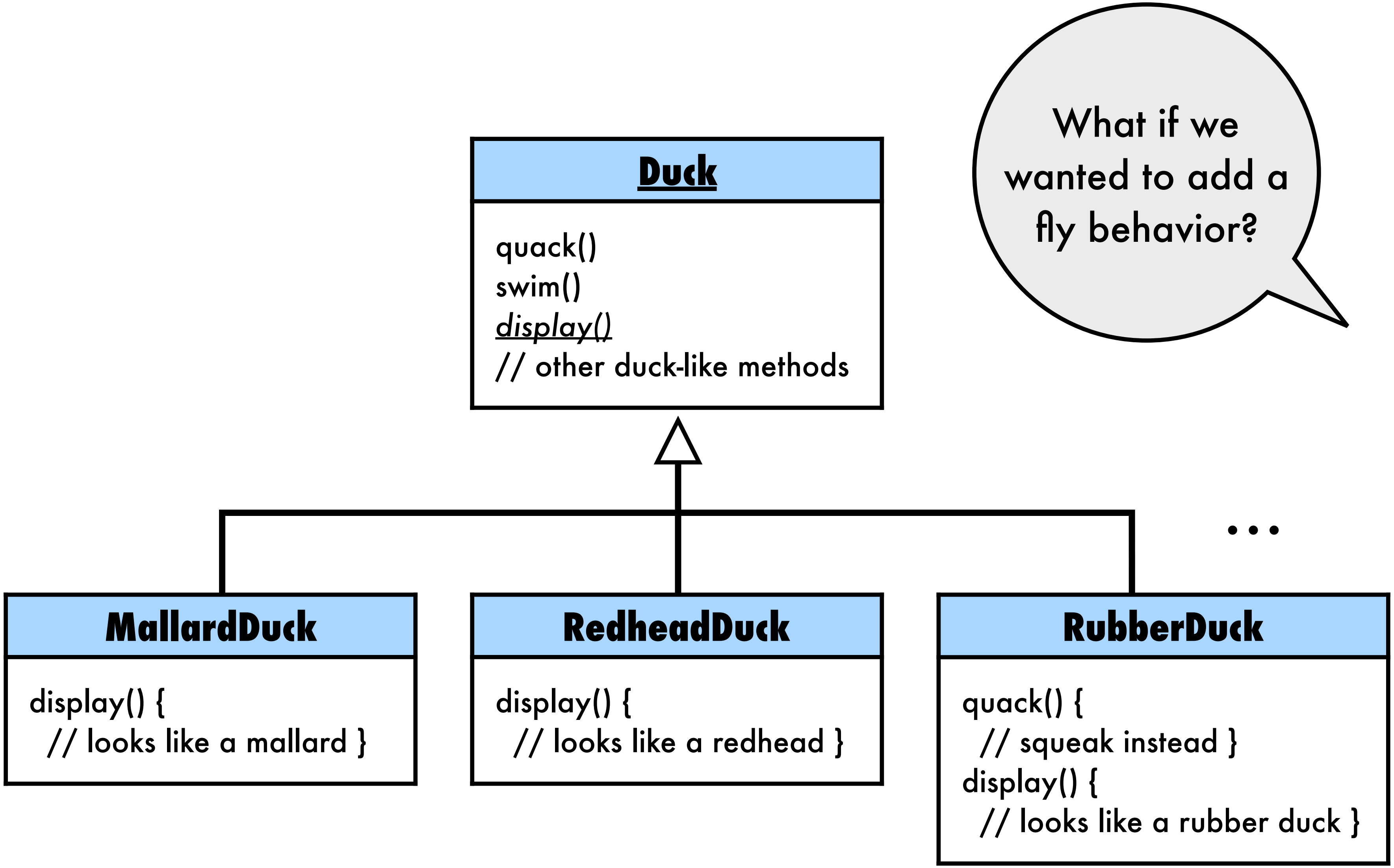
SimUDuck app

- Duck-pond simulation game
- Variety of duck species swimming and making quacking sounds

Initial design



Initial design



What if we wanted to add a fly behavior?

Duck

quack()
swim()
display()
// other duck-like methods

MallardDuck

display() {
  // looks like a mallard }

RedheadDuck

display() {
  // looks like a redhead }

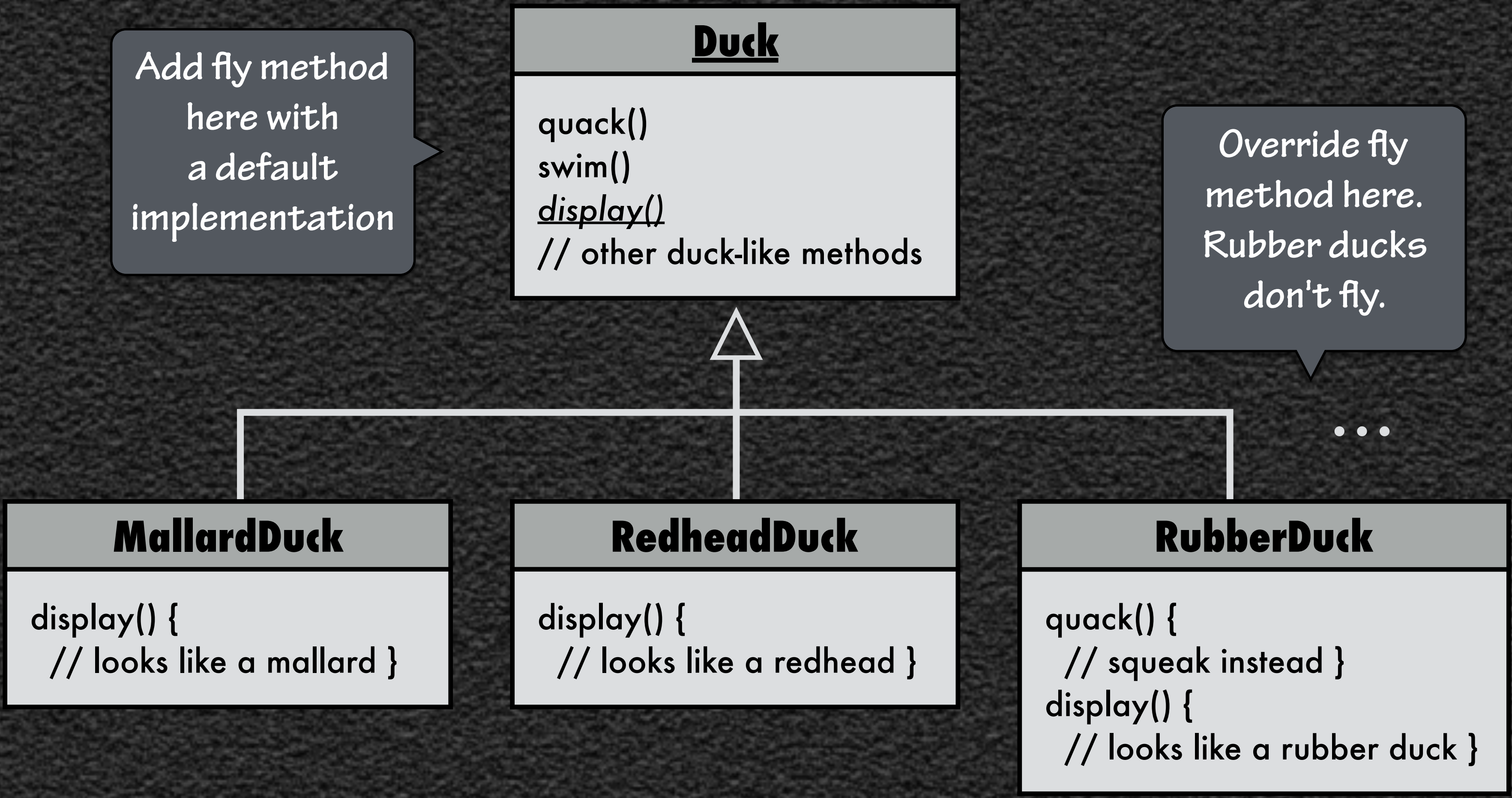
RubberDuck

quack() {
  // squeak instead }
display() {
  // looks like a rubber duck }

...

Pause and Think

How do we add fly behavior to this design?



Add fly method here with a default implementation

Duck

quack()
swim()
display()
// other duck-like methods

Override fly method here. Rubber ducks don't fly.

...

MallardDuck

```
display() {
  // looks like a mallard }
```

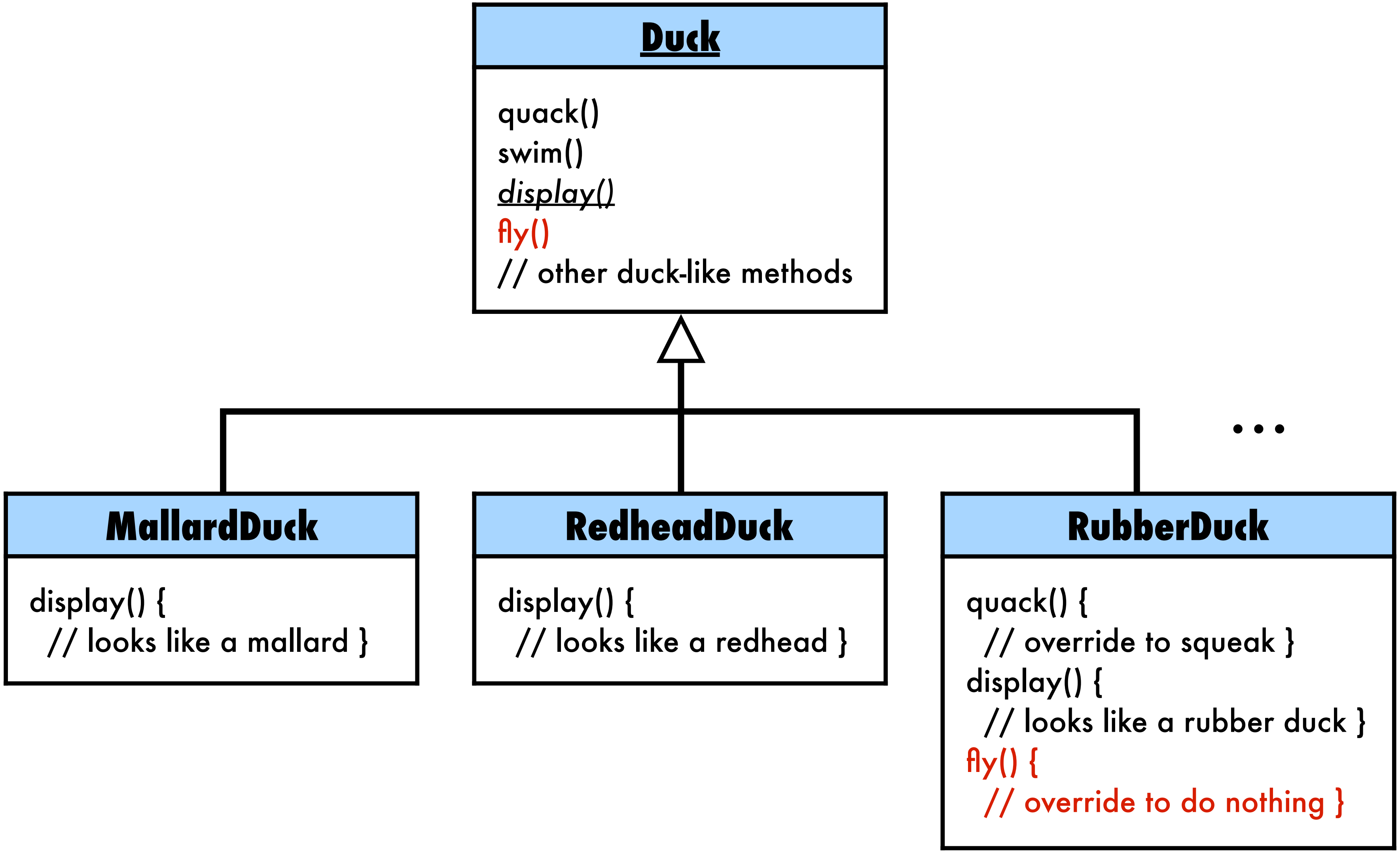
RedheadDuck

```
display() {
  // looks like a redhead }
```

RubberDuck

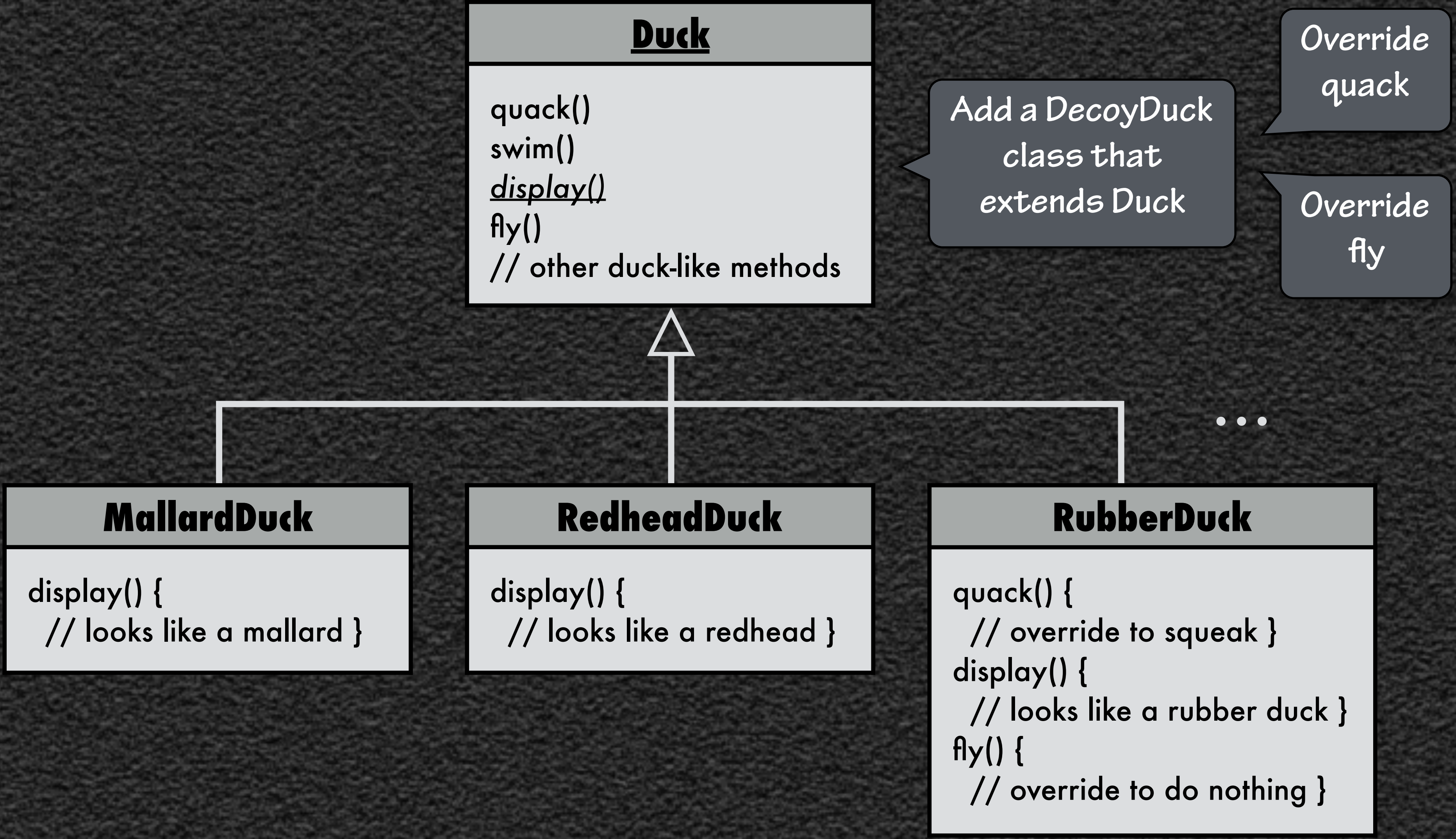
```
quack() {
  // squeak instead }
display() {
  // looks like a rubber duck }
```

Adding fly behavior



Pause and Think

How do we add decoy ducks to this design?



Duck

quack()
swim()
display()
fly()
// other duck-like methods

Add a DecoyDuck class that extends Duck

Override quack

Override fly

...

MallardDuck

display() {
  // looks like a mallard }

RedheadDuck

display() {
  // looks like a redhead }

RubberDuck

quack() {
  // override to squeak }
display() {
  // looks like a rubber duck }
fly() {
  // override to do nothing }

□ Code is duplicated across subclasses

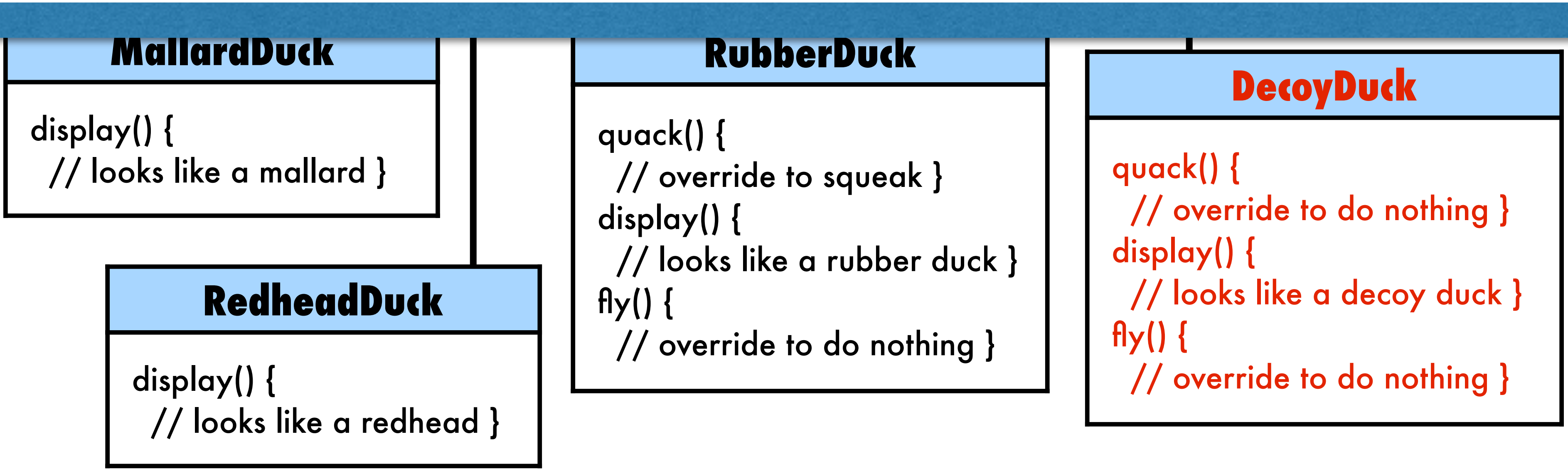
□ Runtime behavior changes are difficult

□ We can't make ducks dance

□ Hard to gain knowledge of all duck behaviors

□ Ducks can't fly and quack at the same time

□ Changes can unintentionally affect other ducks



MallardDuck

```
display() {
  // looks like a mallard }
```

RedheadDuck

```
display() {
  // looks like a redhead }
```

RubberDuck

```
quack() {
  // override to squeak }
display() {
  // looks like a rubber duck }
fly() {
  // override to do nothing }
```

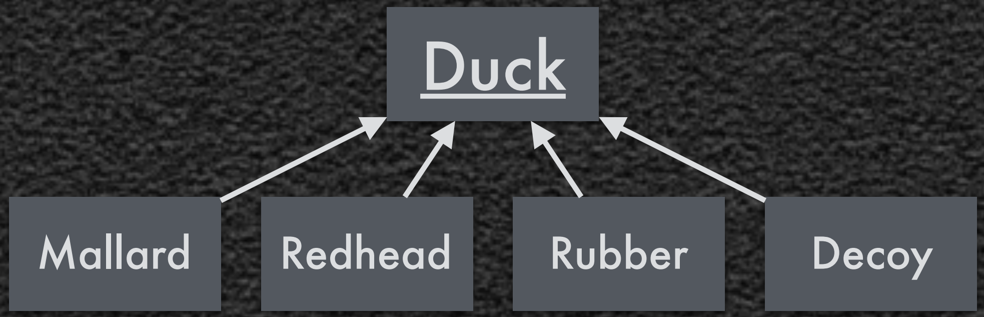
DecoyDuck

```
quack() {
  // override to do nothing }
display() {
  // looks like a decoy duck }
fly() {
  // override to do nothing }
```

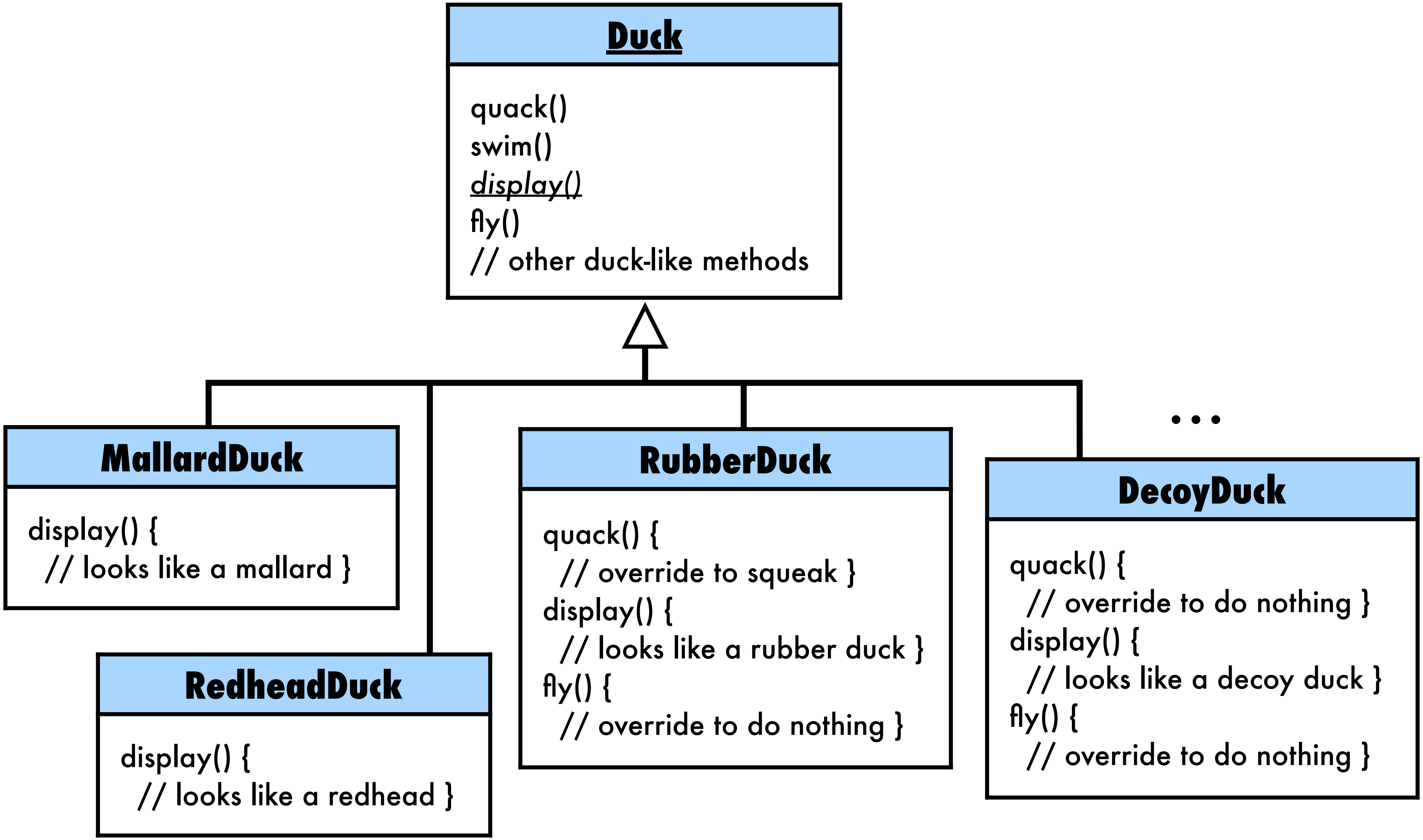
Pause and Think

Which of the following are true about this design?

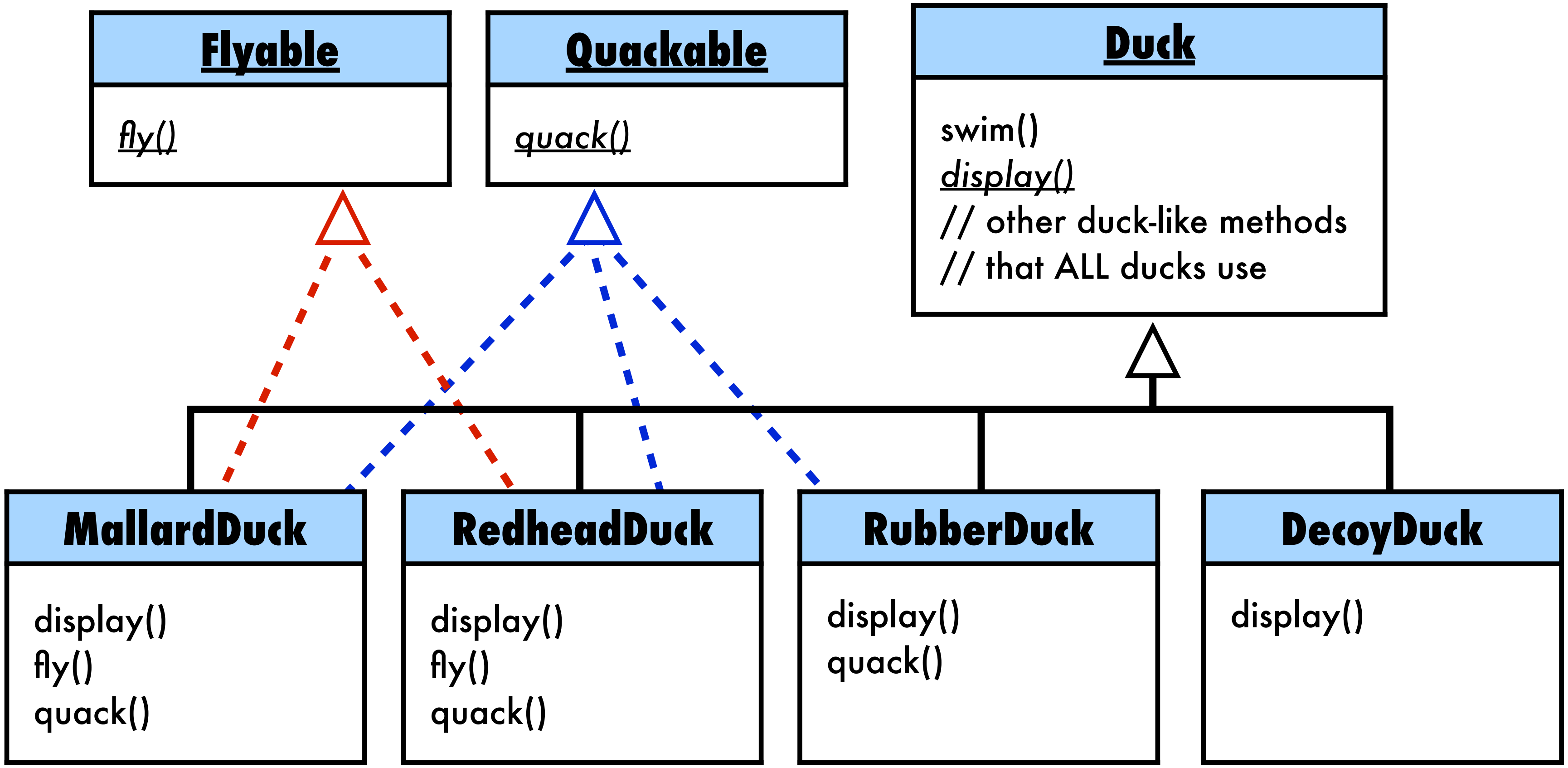
- ☑ Code is duplicated across subclasses
- ☑ Runtime behavior changes are difficult
- □ We can't make ducks dance
- ☑ Hard to gain knowledge of all duck behaviors
- □ Ducks can't fly and quack at the same time
- ☑ Changes can unintentionally affect other ducks



Initial design

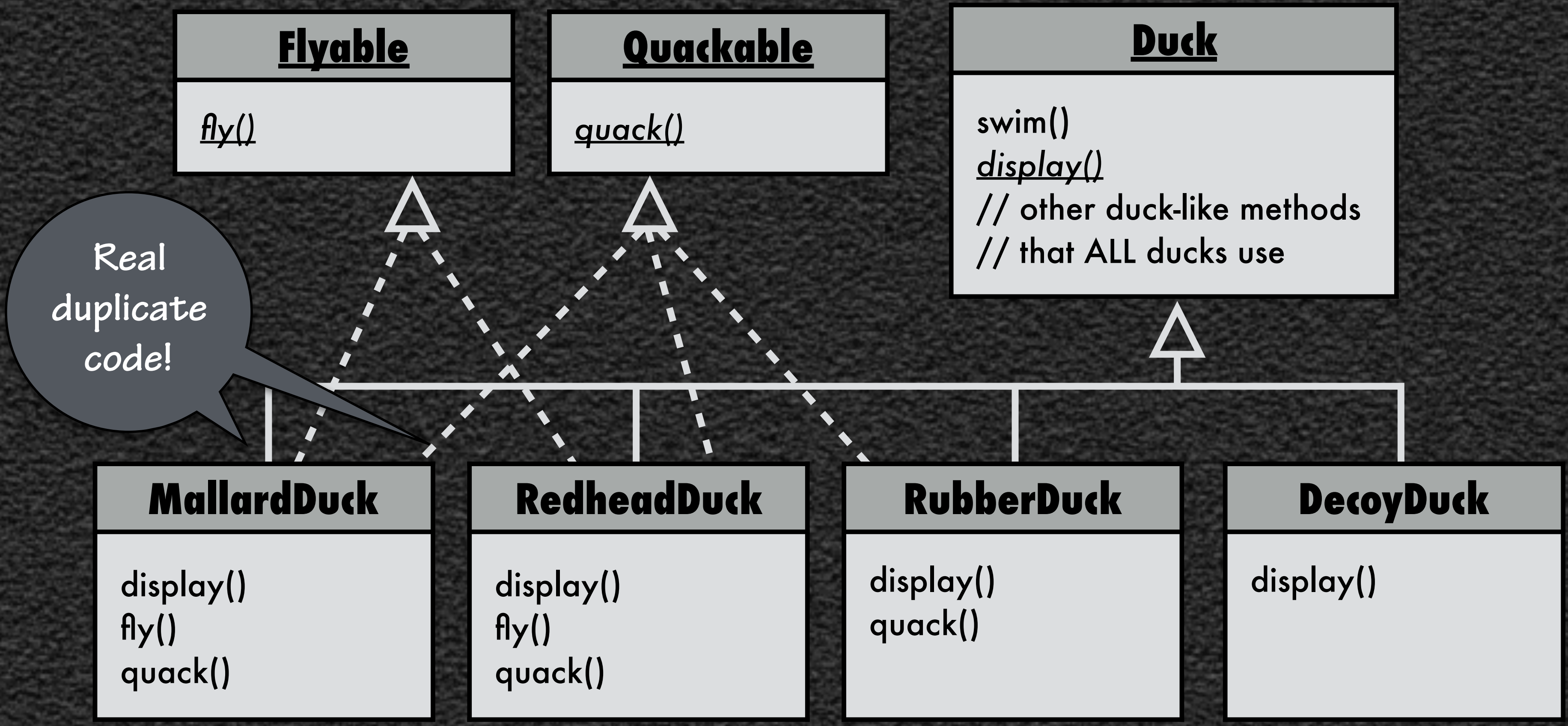

Duck
quack()
swim()
display()
fly()
// other duck-like methods
...
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
RubberDuck
quack() {
// override to squeak }
display() {
// looks like a rubber duck }
fly() {
// override to do nothing }
DecoyDuck
quack() {
// override to do nothing }
display() {
// looks like a decoy duck }
fly() {
// override to do nothing }

Design using interfaces



Pause and Think

What is a serious problem this design?



One constant in software development

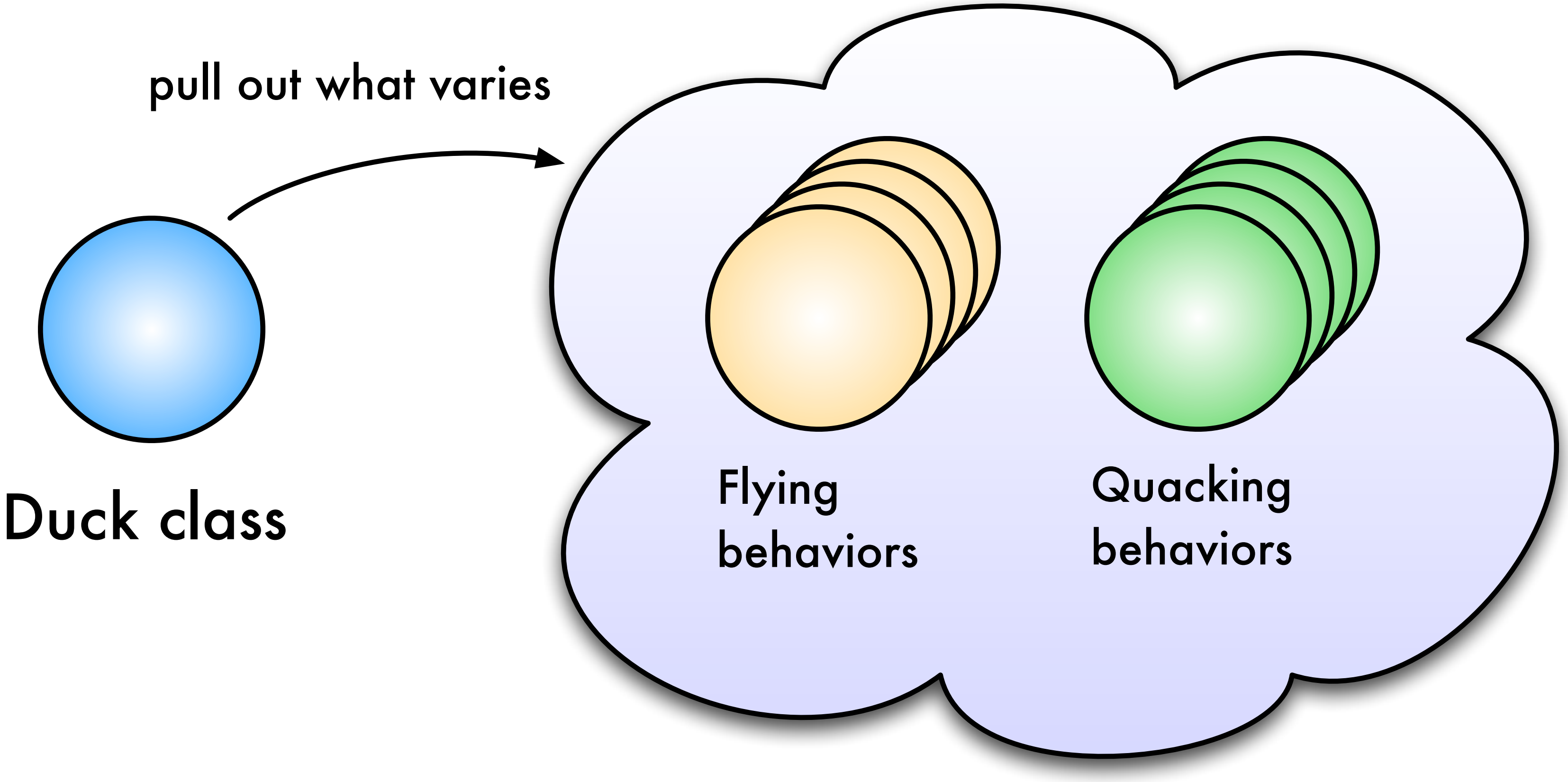
CHANGE

Design Principle

Identify the aspects of your application that vary and separate them from what stays the same.

Encapsulate parts that vary

- Using inheritance for flying behavior means that when you need to modify behavior, you must change subclasses.
- Instead, take the parts of the code that vary and encapsulate them, so that later you can alter or extend those parts without affecting those that don’t.



pull out what varies

Duck class

Flying behaviors

Quacking behaviors

Duck behaviors

Design Principle

*Program to an interface,
not to an implementation.*

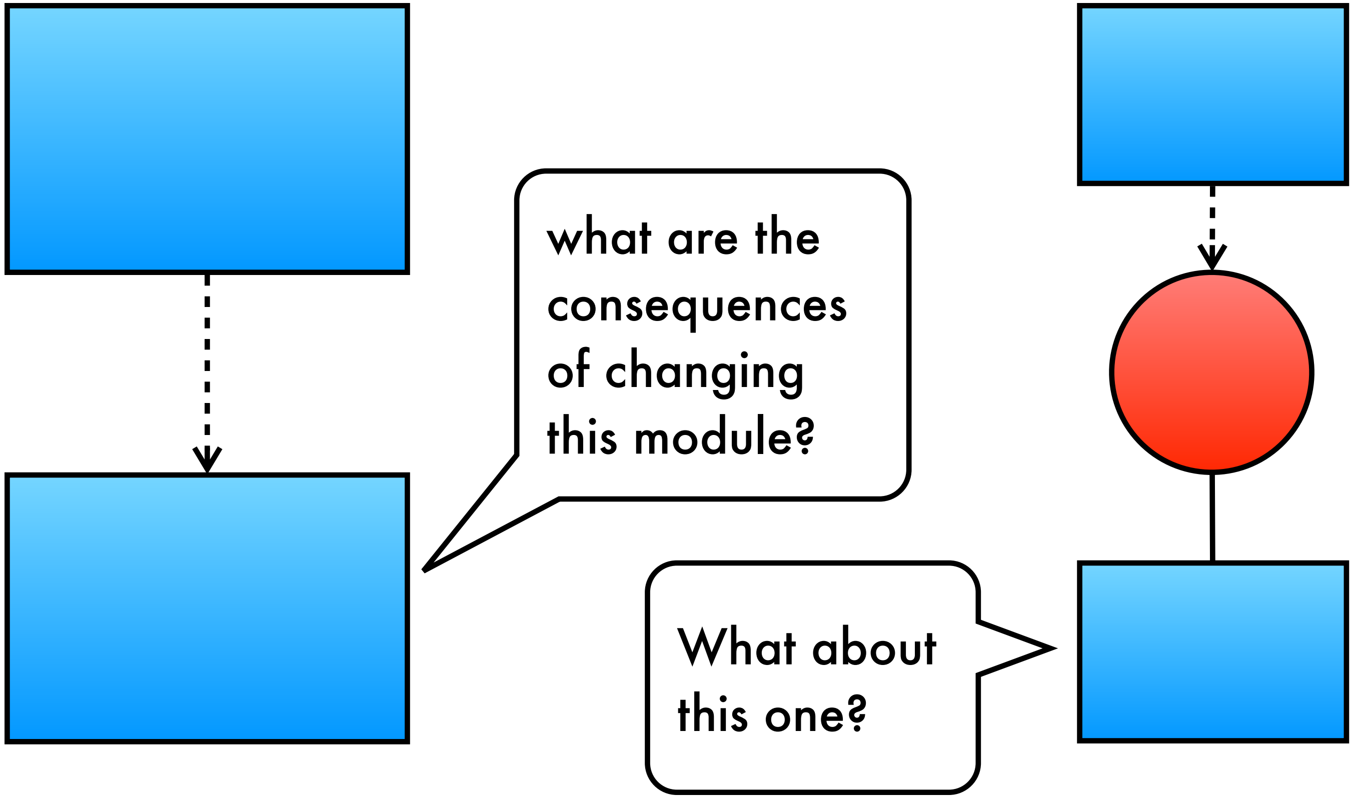
Use interfaces

- We might want the flexibility to assign different behaviors to new instances of duck
    - one mallard might fly, another might not
- If we represent behaviors with interfaces, the implementation won't be locked in to a specific duck.

Use interfaces

- Interfaces change less than implementations
  - Interfaces are subsets of implementations that provide essential signatures
- Implementations can be created, modified, and swapped out without affecting the rest of the code

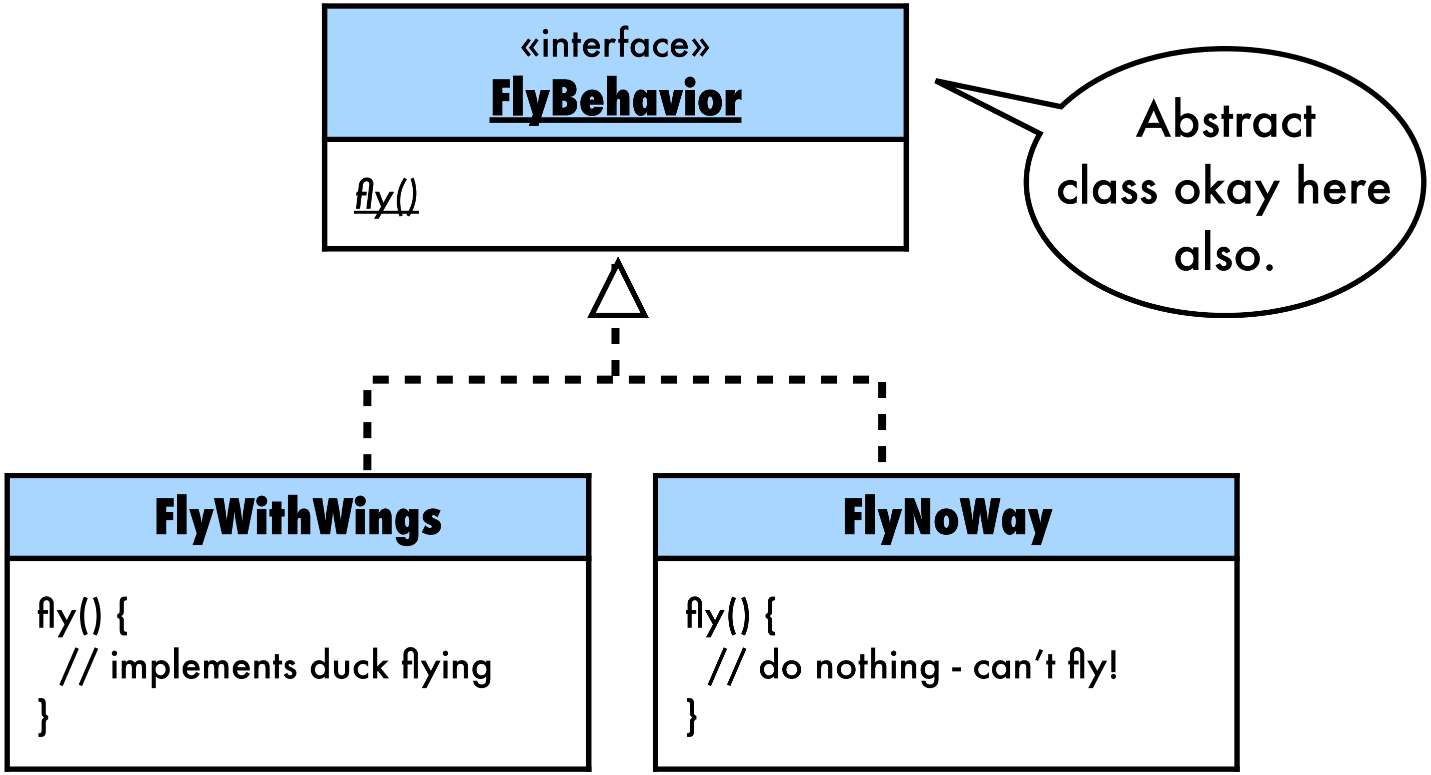
Use interfaces



what are the consequences of changing this module?

What about this one?

Duck behaviors live in classes that implement an interface



«interface»
FlyBehavior

fly()

Abstract class okay here also.

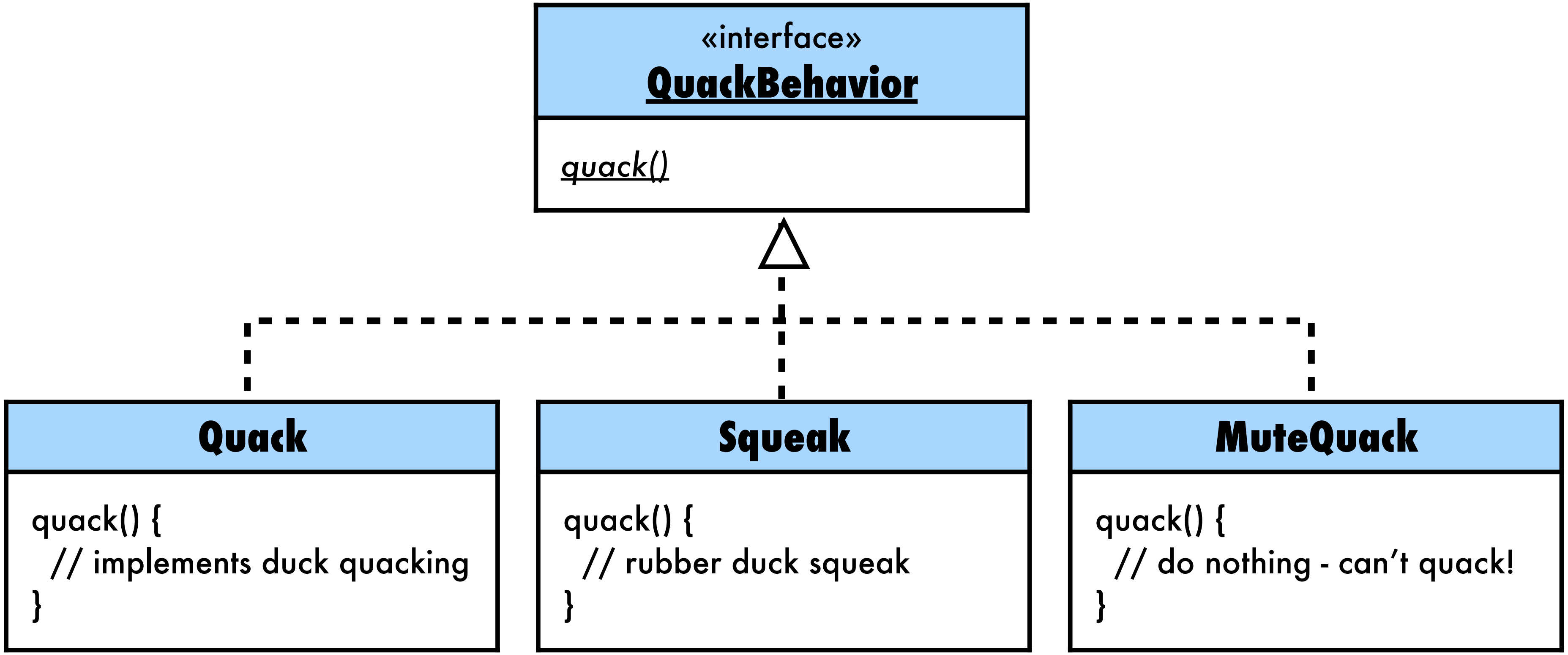
FlyWithWings

```
fly() {
  // implements duck flying
}
```

FlyNoWay

```
fly() {
  // do nothing - can’t fly!
}
```

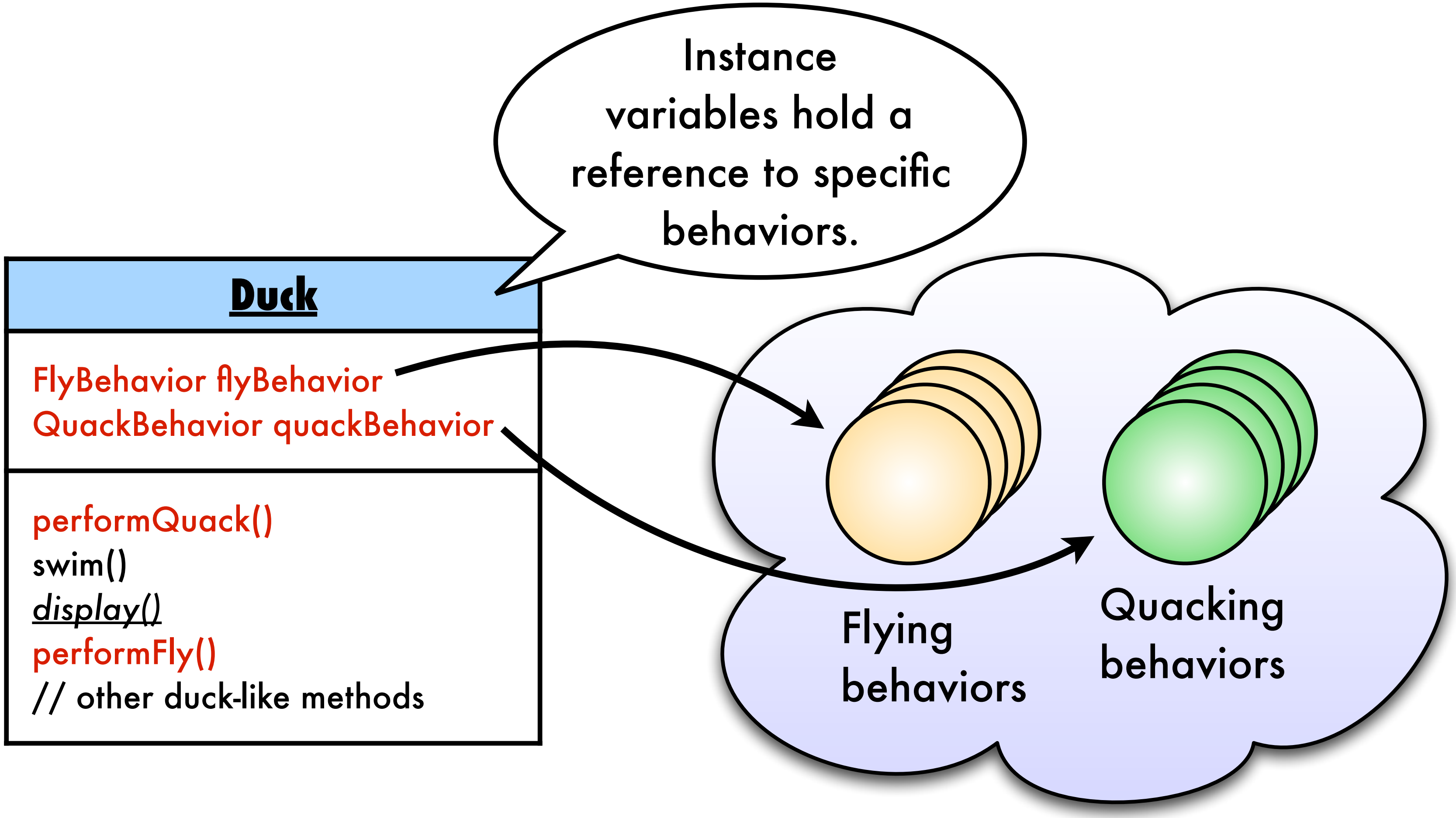
Applies only to behaviors that vary (fly and quack, but not swim)



Benefits of this design

- other types of objects can reuse our fly and quack behaviors
- can add new behaviors without modifying existing behavior classes
- has benefits of reuse without baggage of inheritance

Integrating the Duck behavior



Instance variables hold a reference to specific behaviors.

Duck

FlyBehavior flyBehavior
QuackBehavior quackBehavior

performQuack()
swim()
display()
performFly()
// other duck-like methods

Flying behaviors

Quacking behaviors

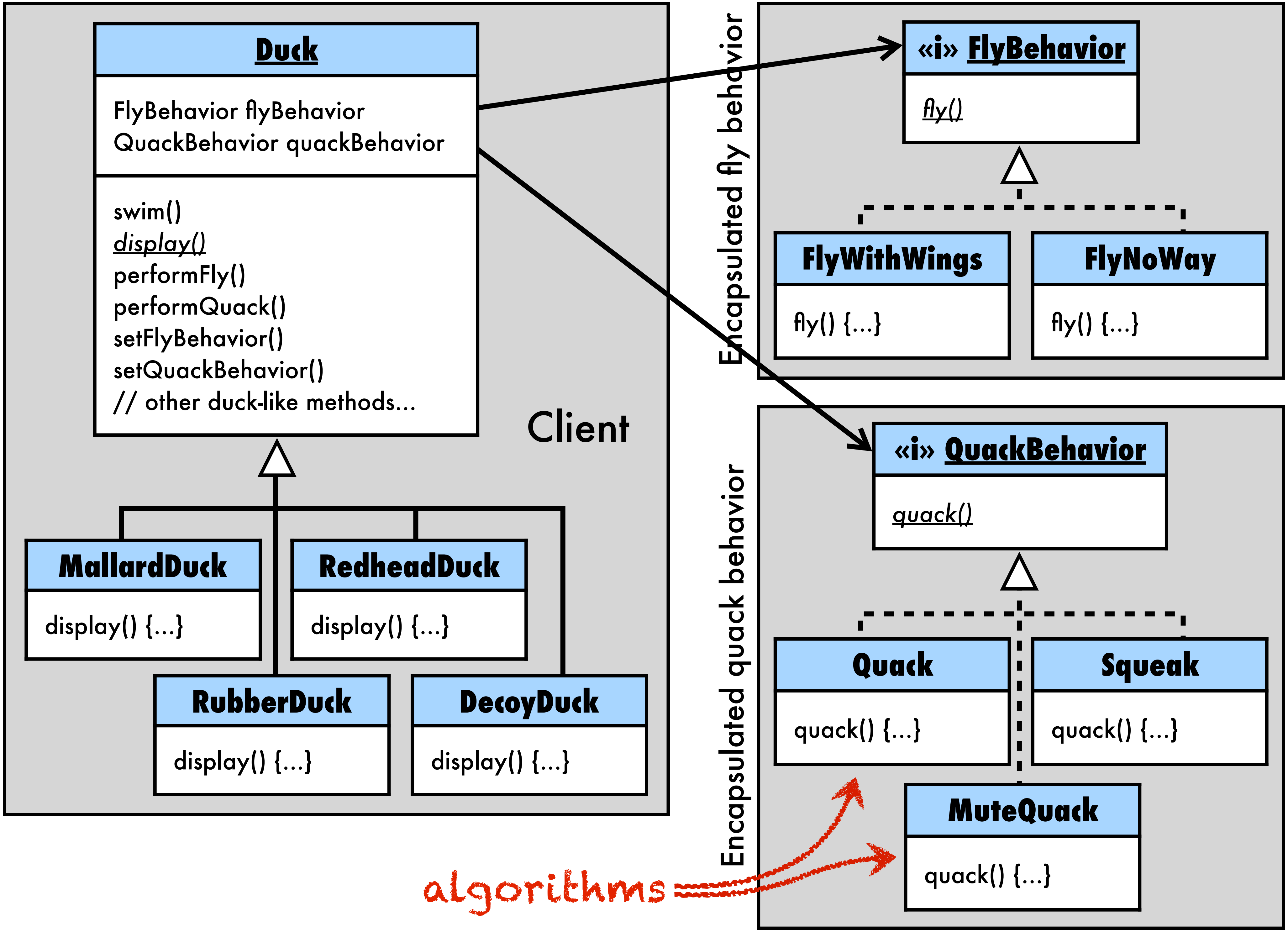
Duck behaviors

MallardDuck class

```
public class MallardDuck extends Duck {
    public MallardDuck() {
        quackBehavior = new Quack();
        flyBehavior = new FlyWithWings();
    }
    public void display() {
        System.out.println(“I’m a real Mallard duck”);
    }
}
```

Duck application test

```
public class DuckApplicationTester {
    public static void main(String[] args) {
        Duck d = new MallardDuck();
        d.performQuack();
        d.swim();
        d.display();
        d.performFly();
        d = new RubberDuck();
        ...
    }
}
```



Duck

FlyBehavior flyBehavior
QuackBehavior quackBehavior

swim()
display()
performFly()
performQuack()
setFlyBehavior()
setQuackBehavior()
// other duck-like methods...

Client

MallardDuck
display() {...}

RedheadDuck
display() {...}

RubberDuck
display() {...}

DecoyDuck
display() {...}

Encapsulated fly behavior

«i» FlyBehavior
fly()

FlyWithWings
fly() {...}

FlyNoWay
fly() {...}

Encapsulated quack behavior

«i» QuackBehavior
quack()

Quack
quack() {...}

Squeak
quack() {...}

MuteQuack
quack() {...}

algorithms

Design Principle

Allows you to encapsulate a family of classes

Favor composition over inheritance.

Allows you to change behavior at runtime

Composition over inheritance

- Has-A rather than Is-A
- More flexibility – can change behavior easily at runtime
- Reasoning is simplified and localized

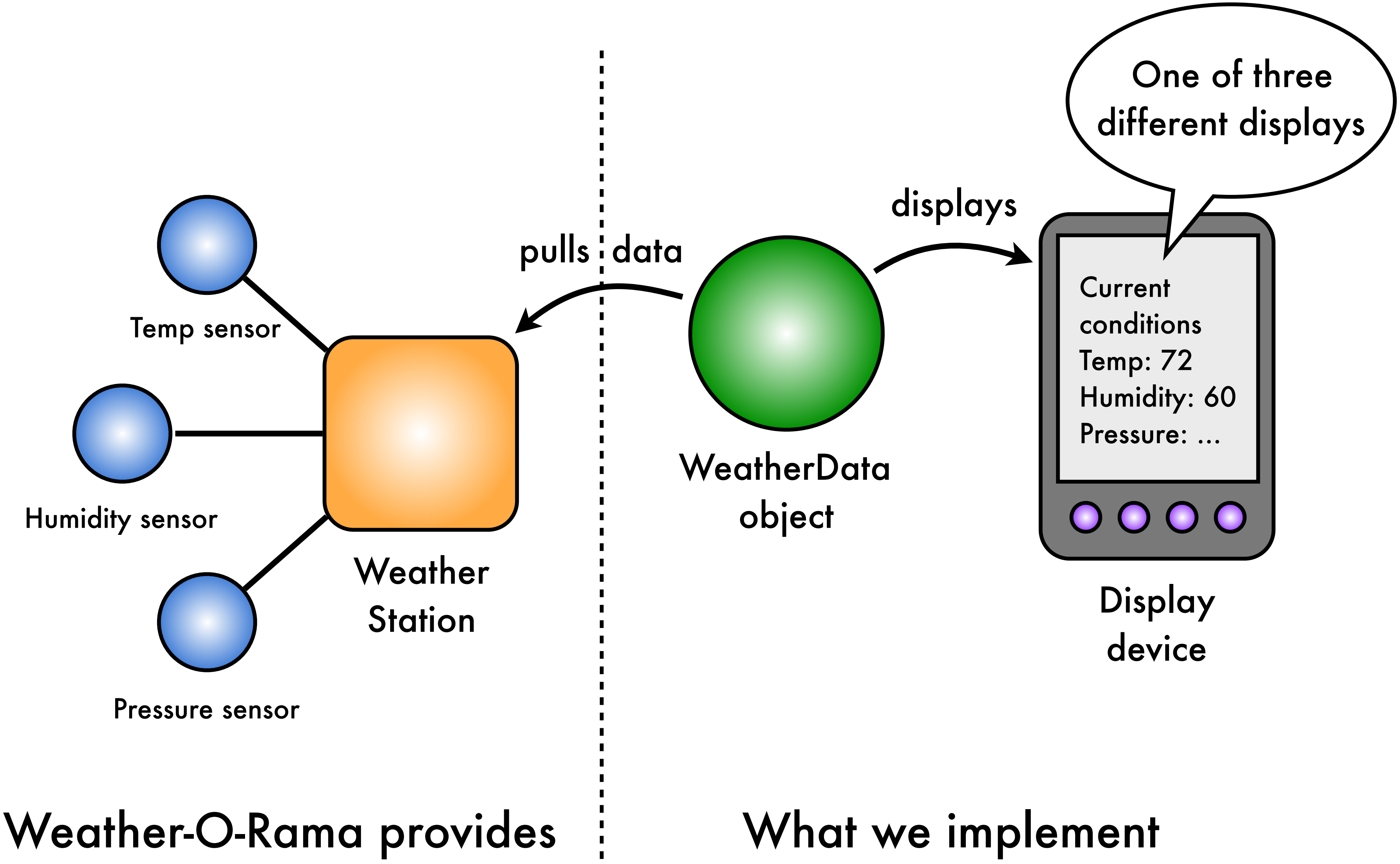
Strategy Pattern

The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Chapter 2

Keeping your objects in the know

Weather Monitoring app



Weather-O-Rama provides

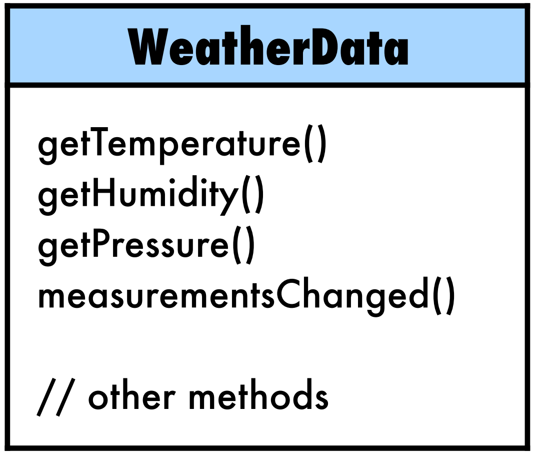
What we implement

Our task

- Create an app that uses the WeatherData object to update three displays: (1) current conditions, (2) weather stats, and (3) forecast.

Our more specific task

We need to implement <u>measurementsChanged</u> so that it updates our display elements

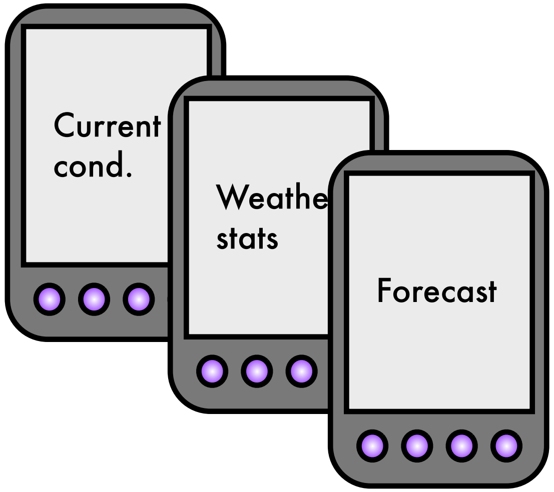


WeatherData

getTemperature()
getHumidity()
getPressure()
measurementsChanged()

// other methods

We need to implement three display elements that are updated each time weather data has new measurements



□ We are coding to concrete implementations, not interfaces

□ For every new display element, we need to alter code

□ We have no way to add/remove display elements at run time

□ The display elements don't implement a common interface

□ We haven't encapsulated the part that changes

□ We are violating encapsulation of the WeatherData class

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

    // other WeatherData methods here
}
```

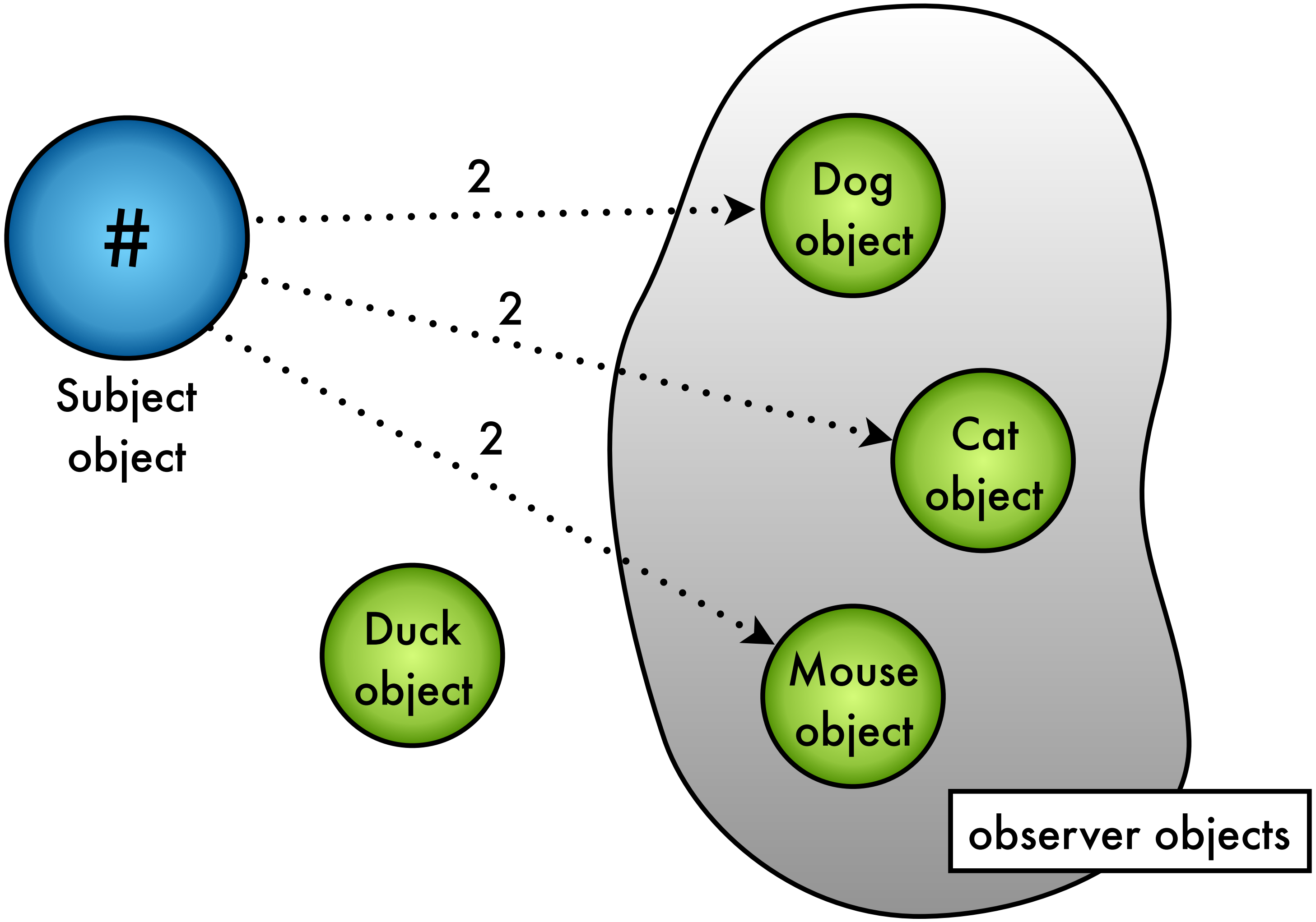
Pause and Think

Which of the following are true about this design?

- ☑ We are coding to concrete implementations, not interfaces
- ☑ For every new display element, we need to alter code
- ☑ We can't add/remove display elements at runtime
- ☐ The display elements don't implement a common interface
- ☑ We haven't encapsulated the part that changes
- ☐ We are violating encapsulation of the WeatherData class

```
public void measurementsChanged() { ...
    currentConditionsDisplay.update(temp, humidity, pressure);
    statisticsDisplay.update(temp, humidity, pressure);
    forecastDisplay.update(temp, humidity, pressure);
}
```

Publishers + Subscribers = Observer Pattern



Observer Pattern

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

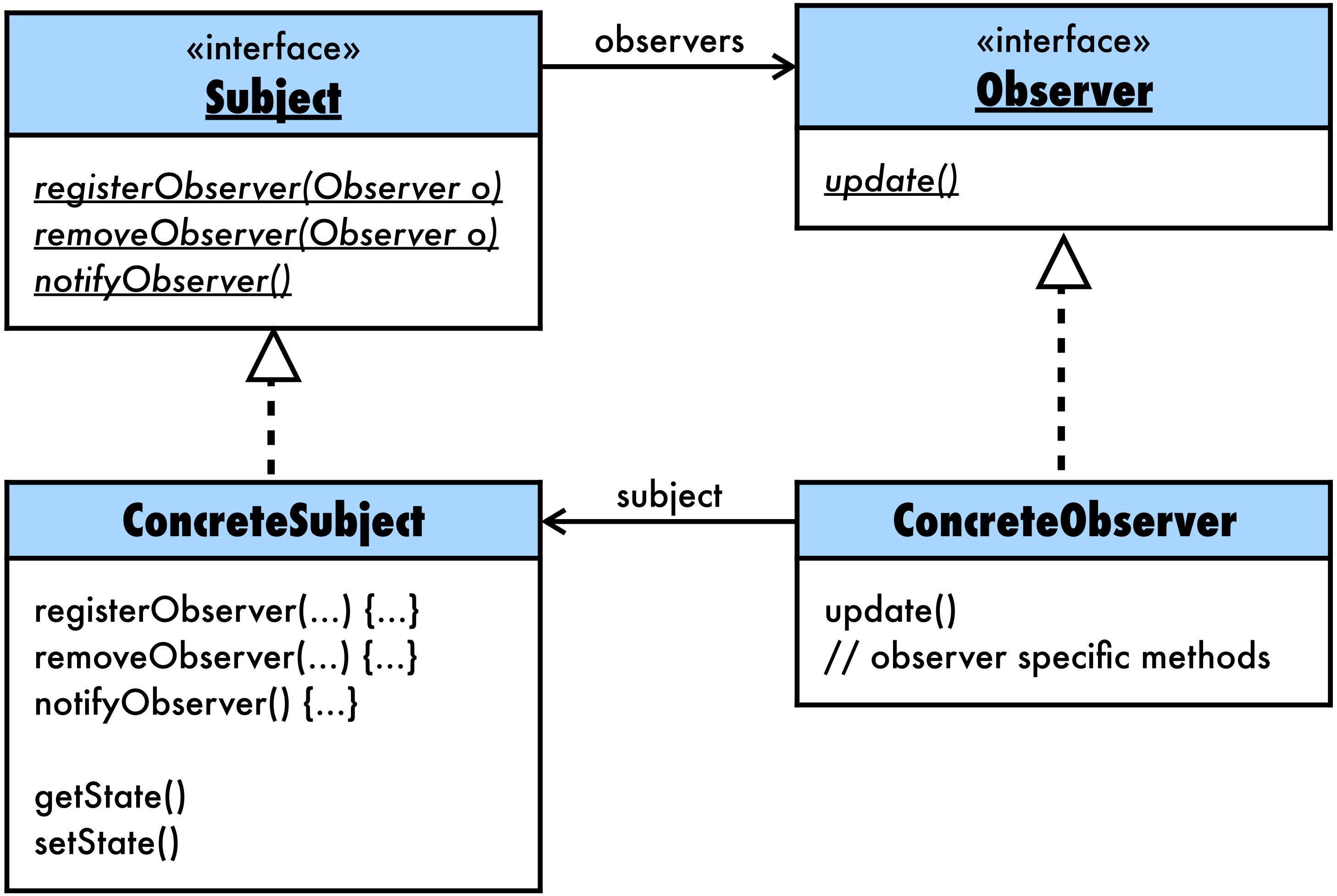
Design Principle

Strive for loosely coupled designs between objects that interact.

Loose coupling

- When two objects are loosely coupled, they can interact, but have very little knowledge of each other
- Loosely coupled designs can handle change easier because they minimize the interdependency between objects

Class diagram for observer pattern



«interface»
Subject

registerObserver(Observer o)
removeObserver(Observer o)
notifyObserver()

«interface»
Observer

update()

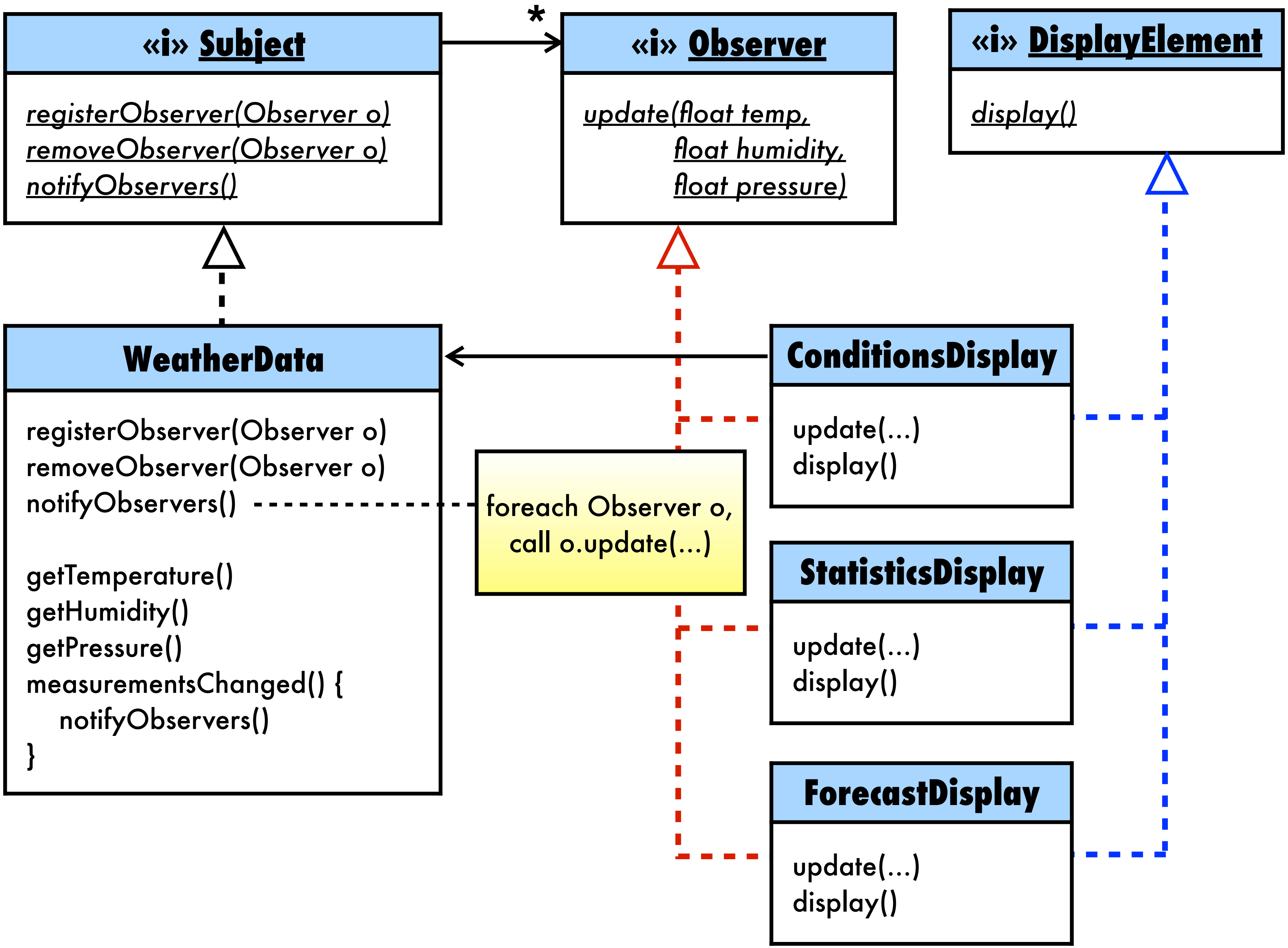
ConcreteSubject

registerObserver(...) {...}
removeObserver(...) {...}
notifyObserver() {...}

getState()
setState()

ConcreteObserver

update()
// observer specific methods



«i» Subject

registerObserver(Observer o)
removeObserver(Observer o)
notifyObservers()

*

«i» Observer

update(float temp,
float humidity,
float pressure)

«i» DisplayElement

display()

WeatherData

registerObserver(Observer o)
removeObserver(Observer o)
notifyObservers()

getTemperature()
getHumidity()
getPressure()
measurementsChanged() {
    notifyObservers()
}

foreach Observer o,
call o.update(...)

ConditionsDisplay

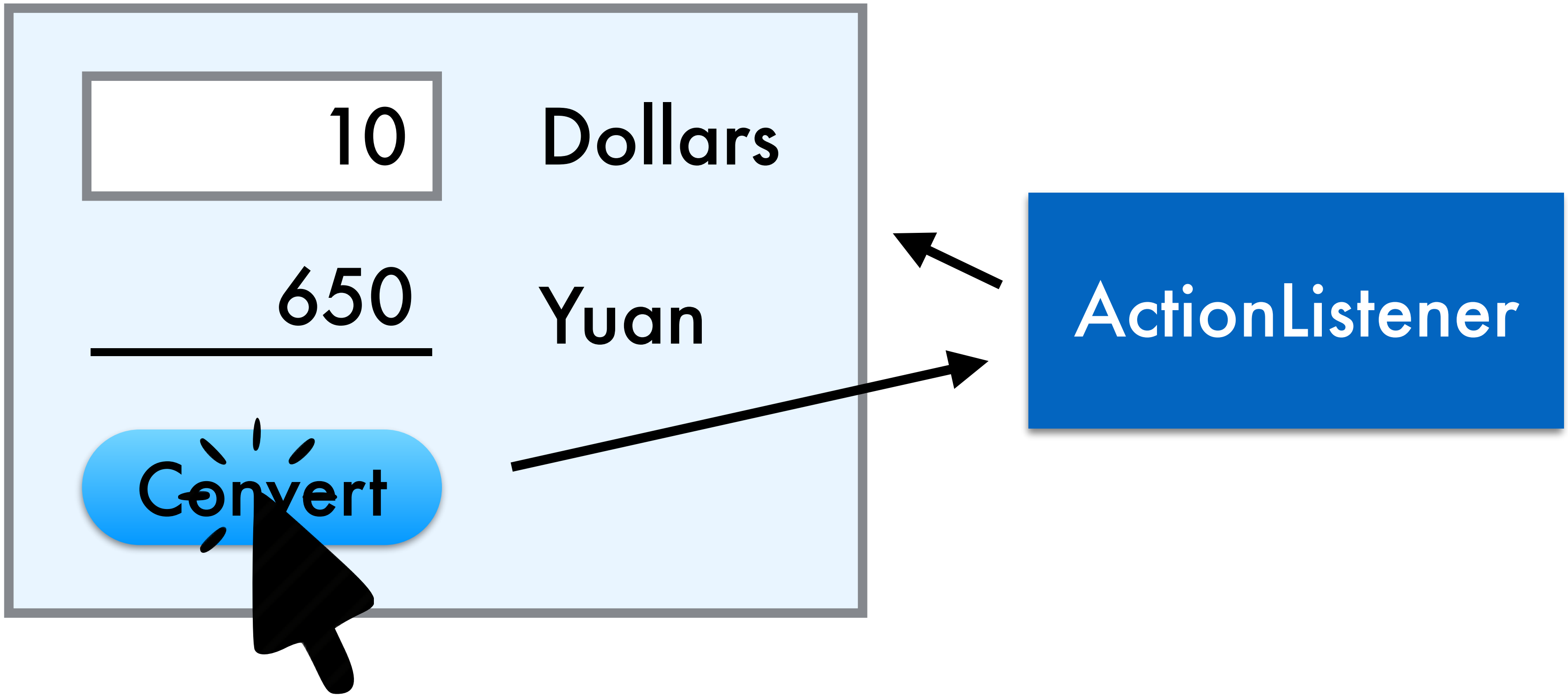
update(...)
display()

StatisticsDisplay

update(...)
display()

ForecastDisplay

update(...)
display()



10

Dollars

650

Yuan

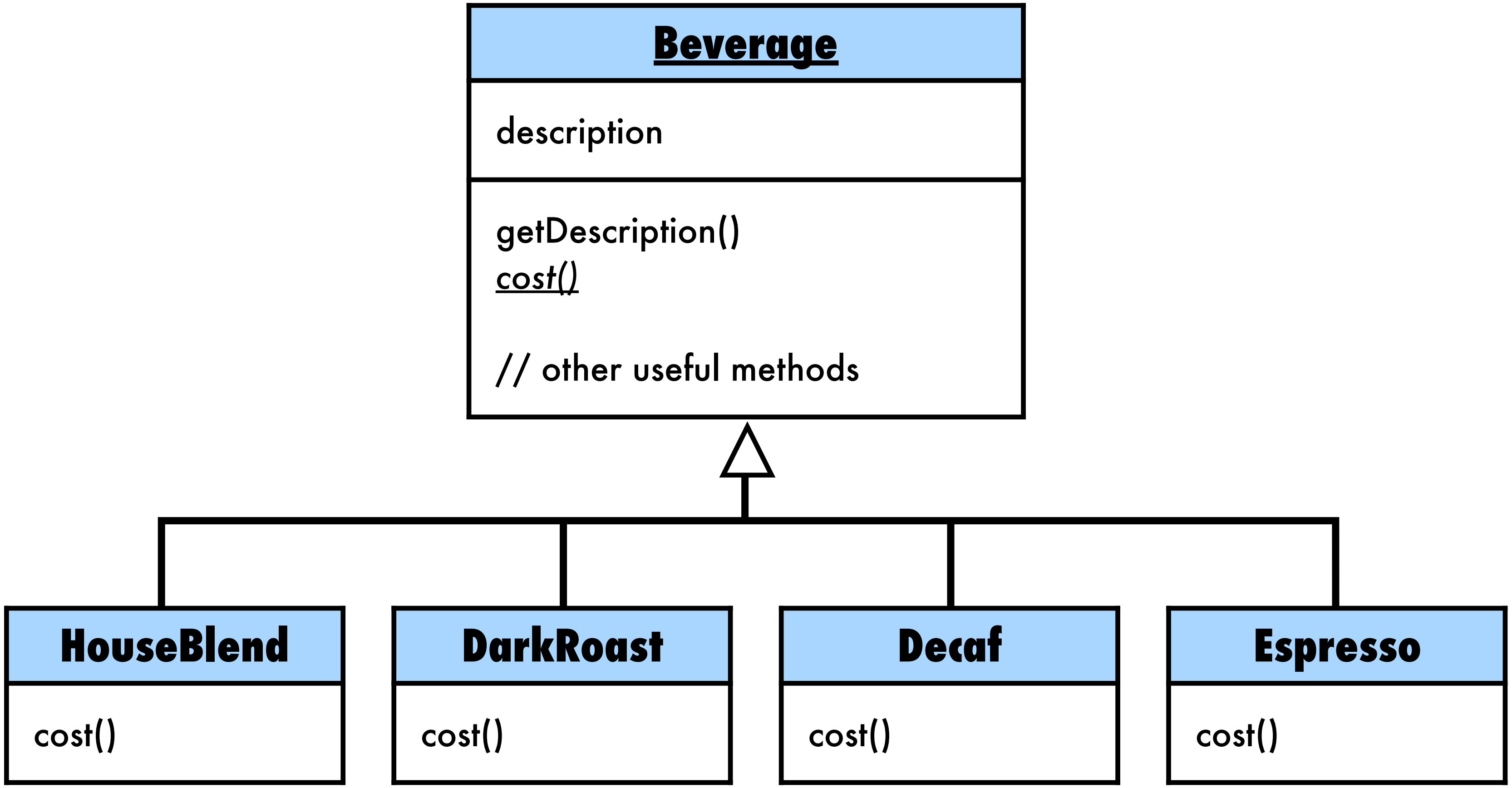
Convert

ActionListener

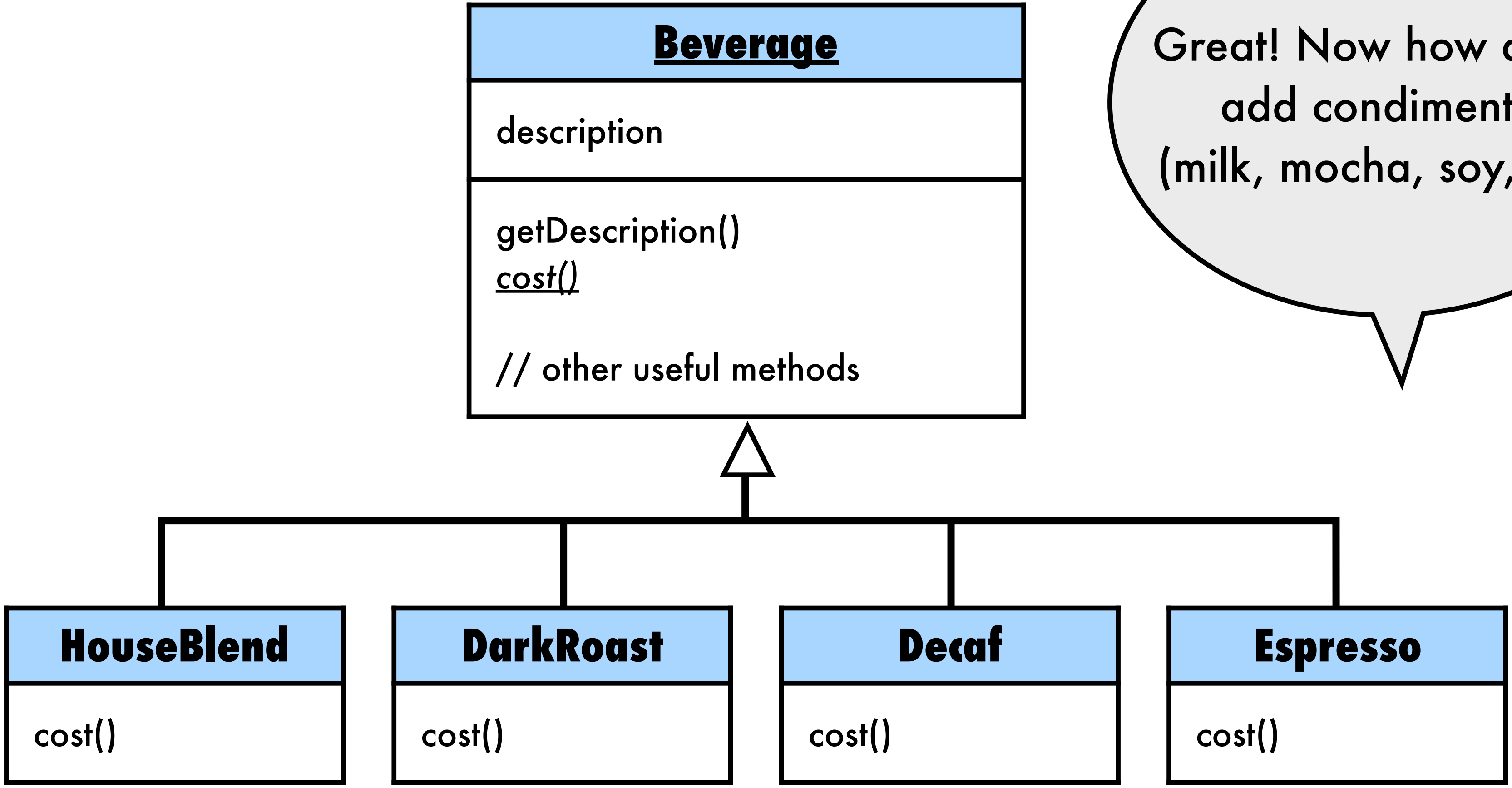
Chapter 3

Decorating objects

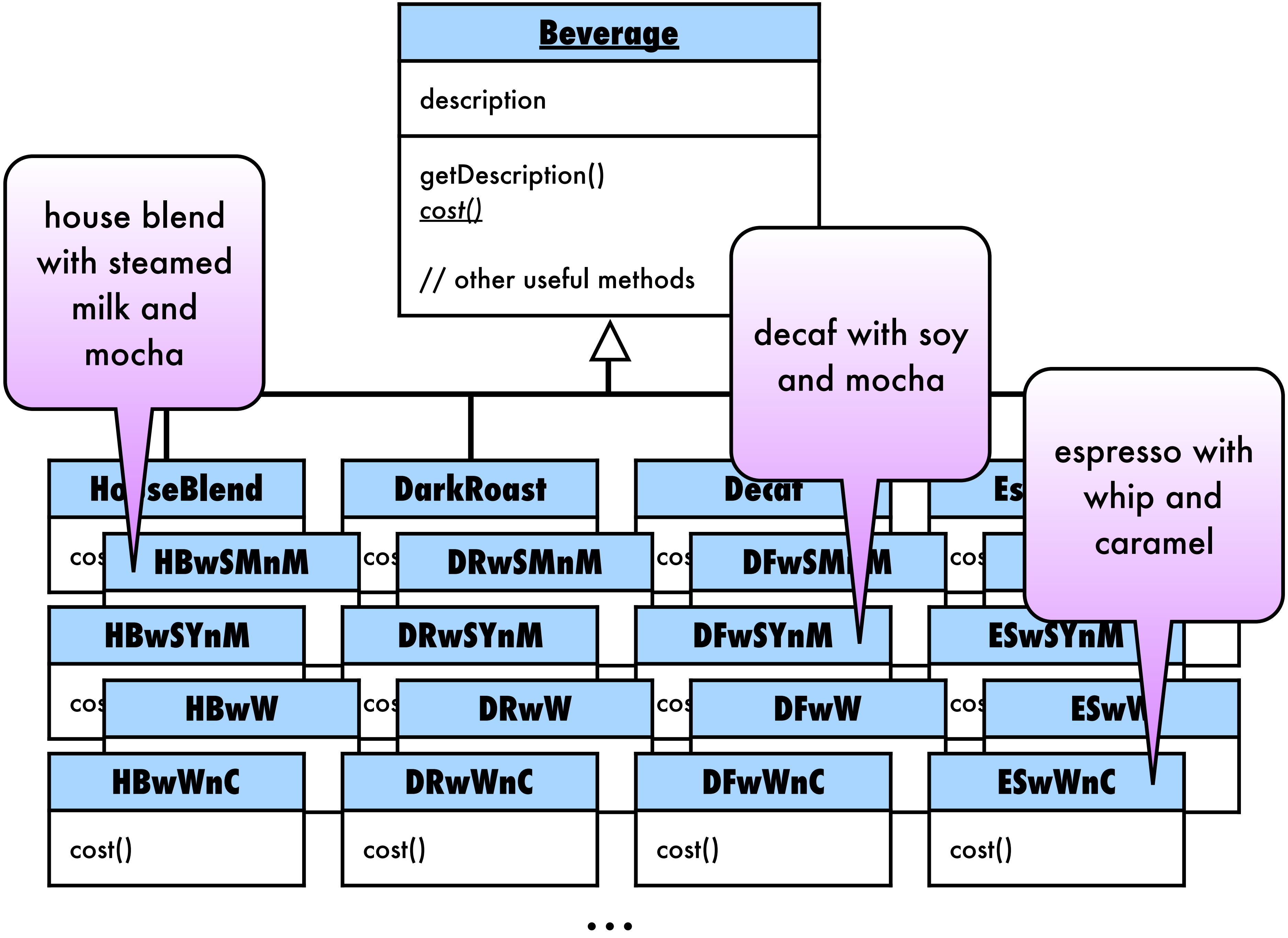
Starbuzz Coffee app



Starbuzz Coffee app



Great! Now how do we add condiments? (milk, mocha, soy, etc.)



Beverage

description

getDescription()
cost()

// other useful methods

Ho seBlend

DarkRoast

Decaf

Es

HBwSMnM

DRwSMnM

DFwSM M

HBwSYnM

DRwSYnM

DFwSYnM

ESwSYnM

HBwW

DRwW

DFwW

ESwW

HBwWnC

DRwWnC

DFwWnC

ESwWnC

cost()

cost()

cost()

cost()

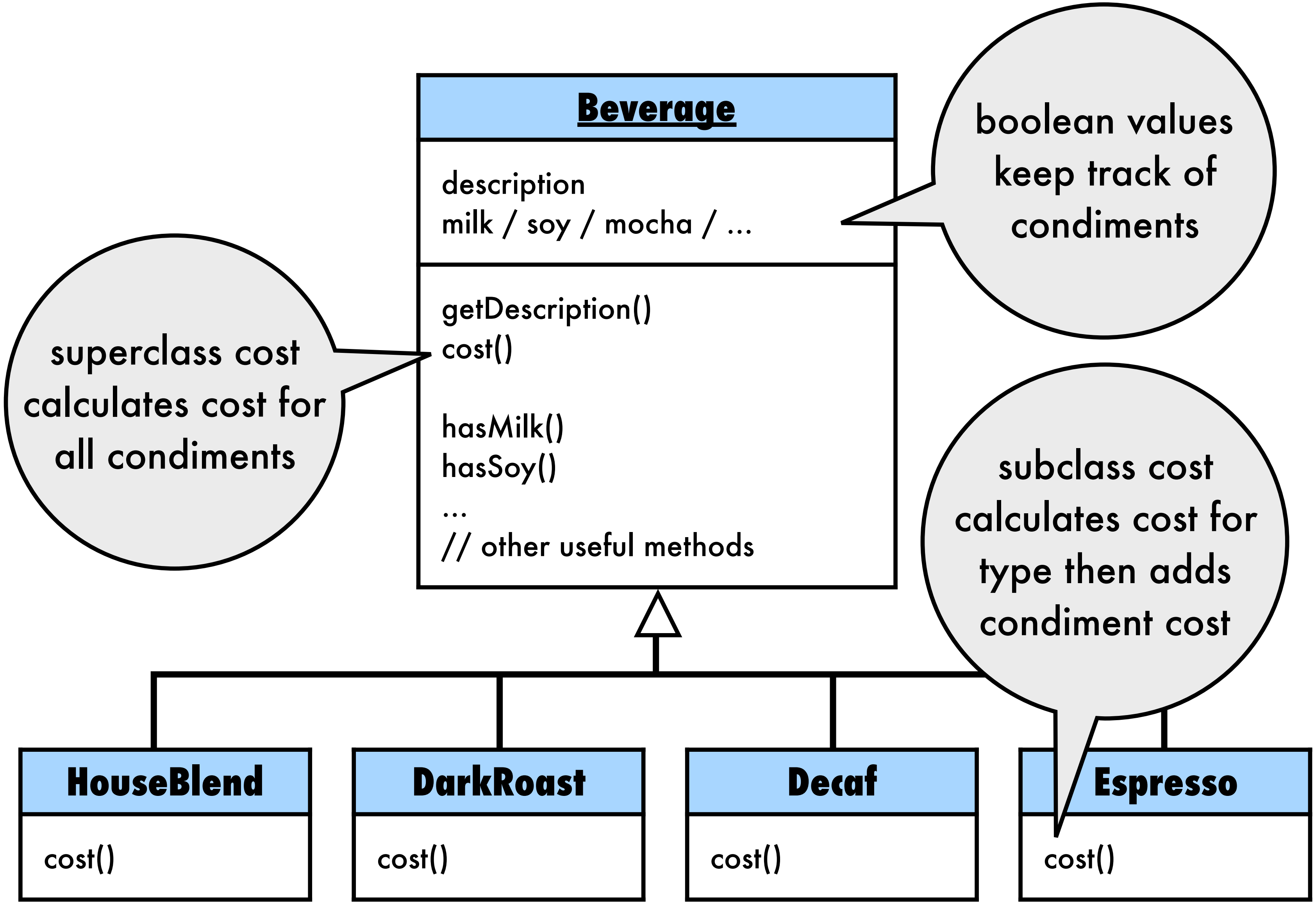
...

house blend with steamed milk and mocha

decaf with soy and mocha

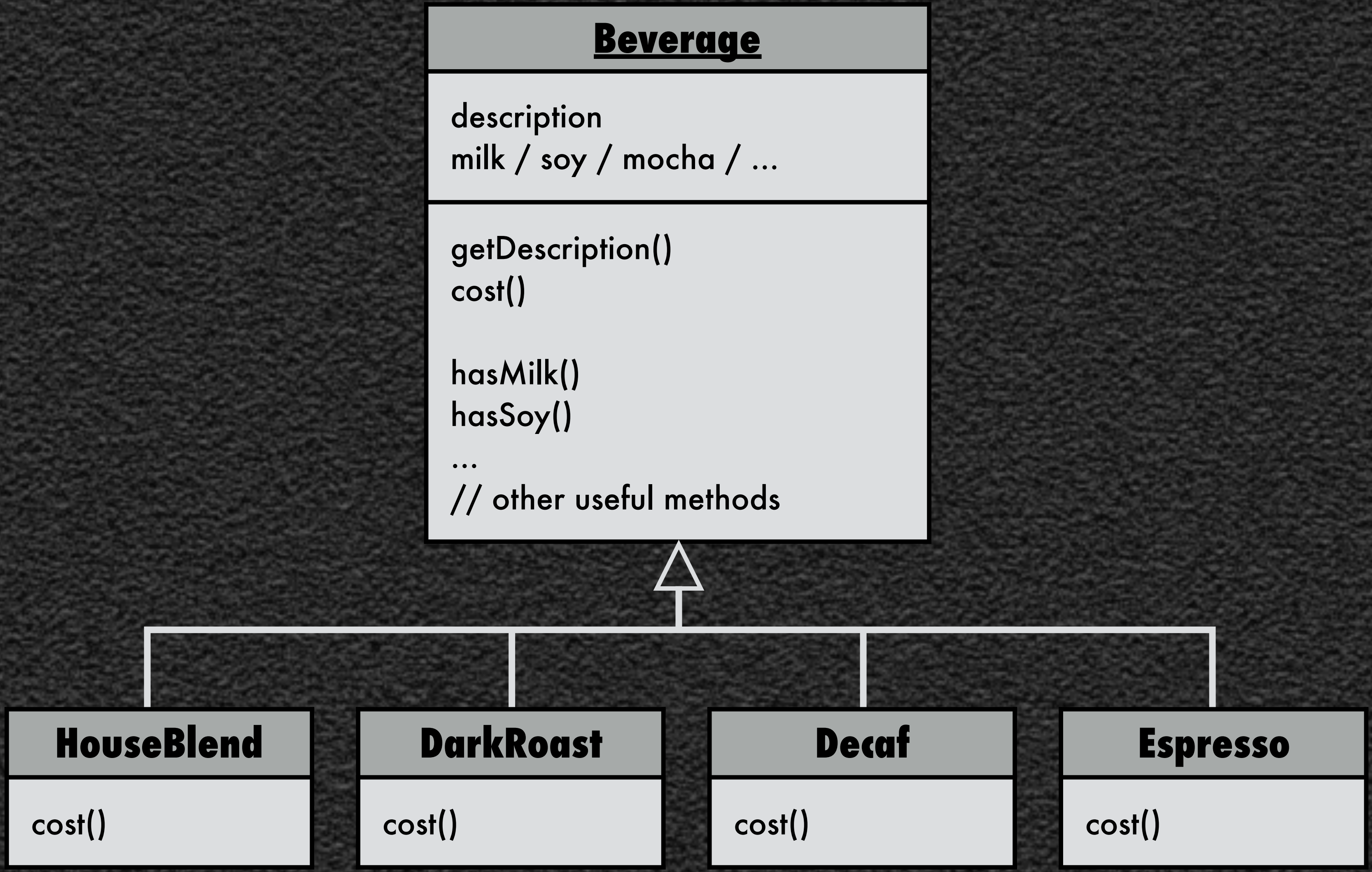
espresso with whip and caramel

Alternative design



Pause and Think

What requirements might change that will impact this design?



Beverage

description
milk / soy / mocha / ...

getDescription()
cost()

hasMilk()
hasSoy()
...
// other useful methods

HouseBlend

cost()

DarkRoast

cost()

Decaf

cost()

Espresso

cost()

Potential Problems?

- Price changes for condiments will force us to alter existing code
- New condiments will force us to add new methods and alter the cost method in the superclass
- We may have new beverages. For some, the condiments may not be appropriate, yet a Tea subclass will still inherit methods like hasWhip().
- What if a customer wants a double mocha?

Design Principle

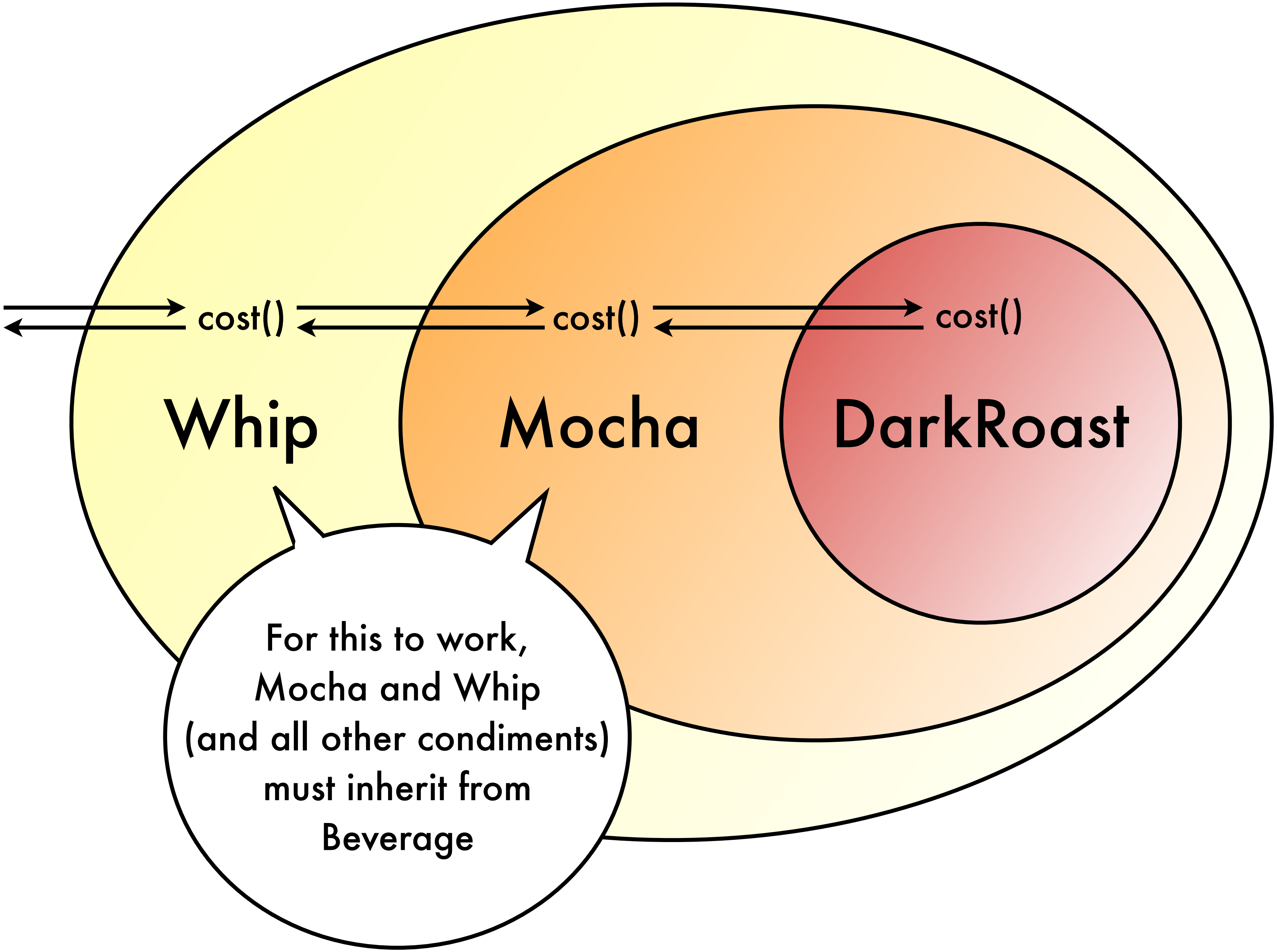
Classes should be open for extension, but closed for modification.

Open-Closed Principle

- Open – Feel free to extend our classes with new behavior if your needs and requirements change (as they will).
- Closed – We spent a lot of time getting this code correct and bug free, so we can’t let you alter existing code.

Decorator Pattern

1. Take a DarkRoast object
2. Decorate it with a Mocha object
3. Decorate it with a Whip object
4. Call the cost() method and rely on delegation to add on the condiment costs



cost()

cost()

cost()

Whip

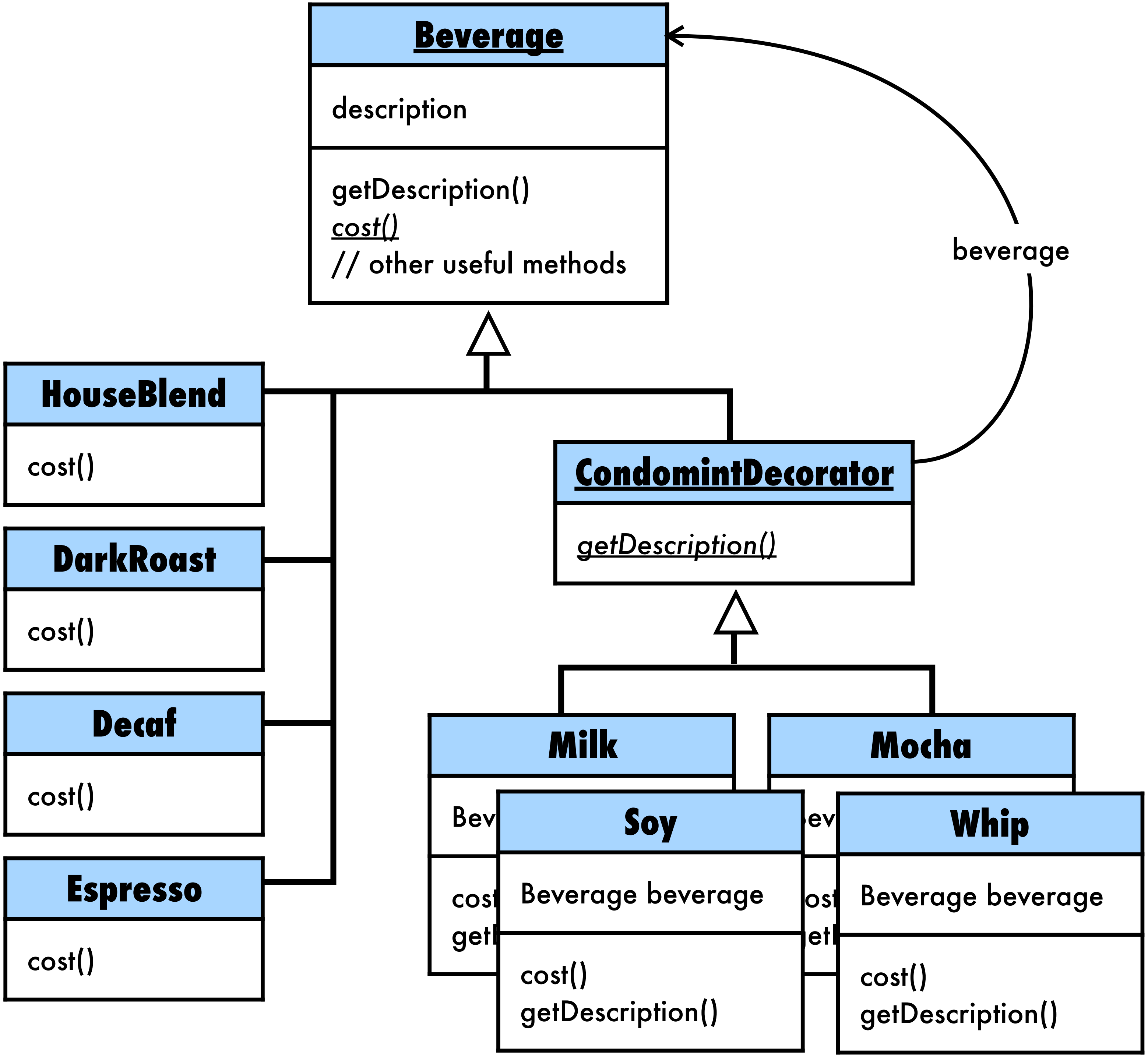
Mocha

DarkRoast

For this to work, Mocha and Whip (and all other condiments) must inherit from Beverage

Decorator Pattern

The Decorator Pattern attaches additional functionality to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



Beverage

description

getDescription()
cost()
// other useful methods

beverage

HouseBlend

cost()

DarkRoast

cost()

Decaf

cost()

Espresso

cost()

CondomintDecorator

getDescription()

Milk

Bev

cost
get

Mocha

ev

ost
et

Soy

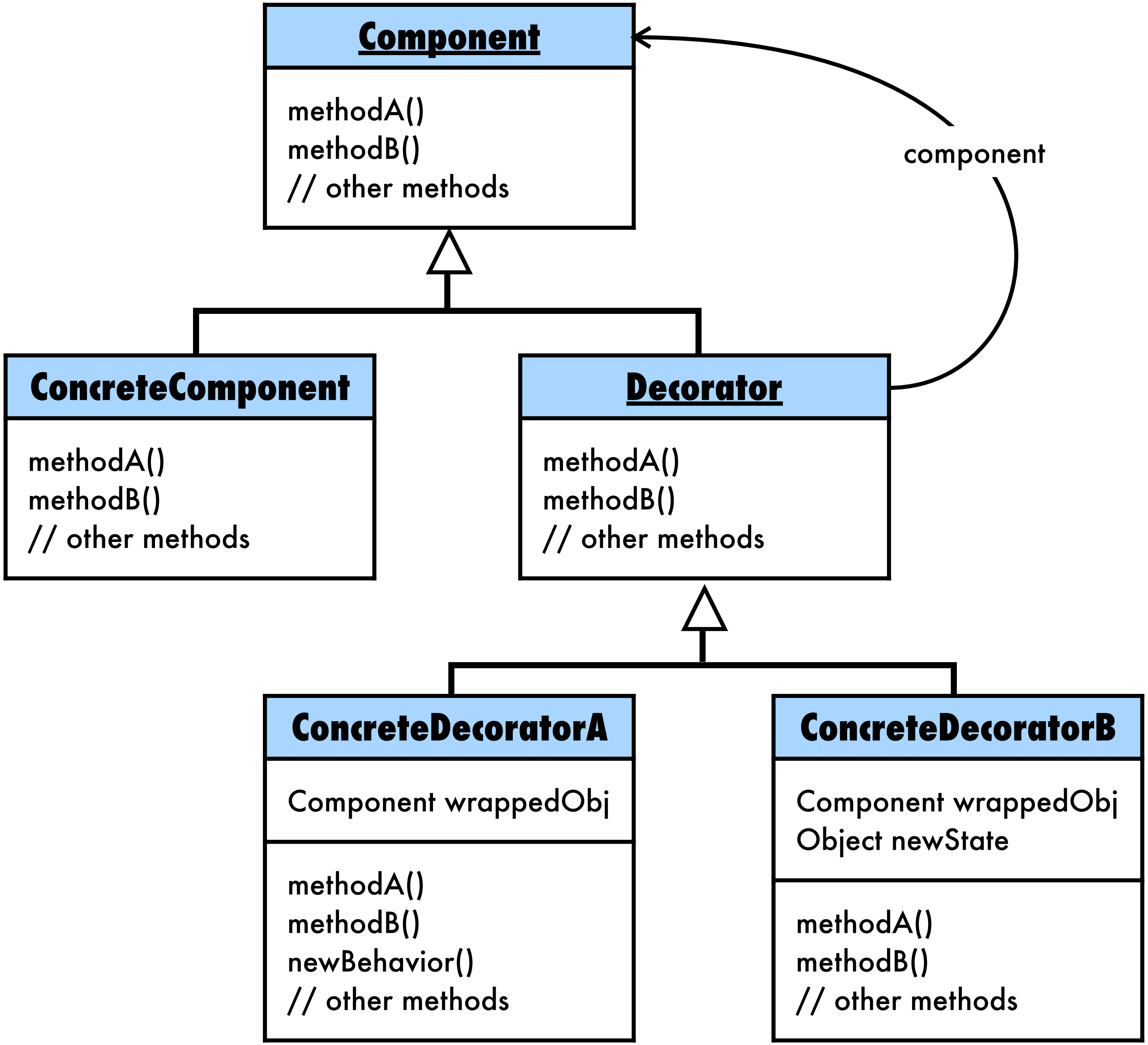
Beverage beverage

cost()
getDescription()

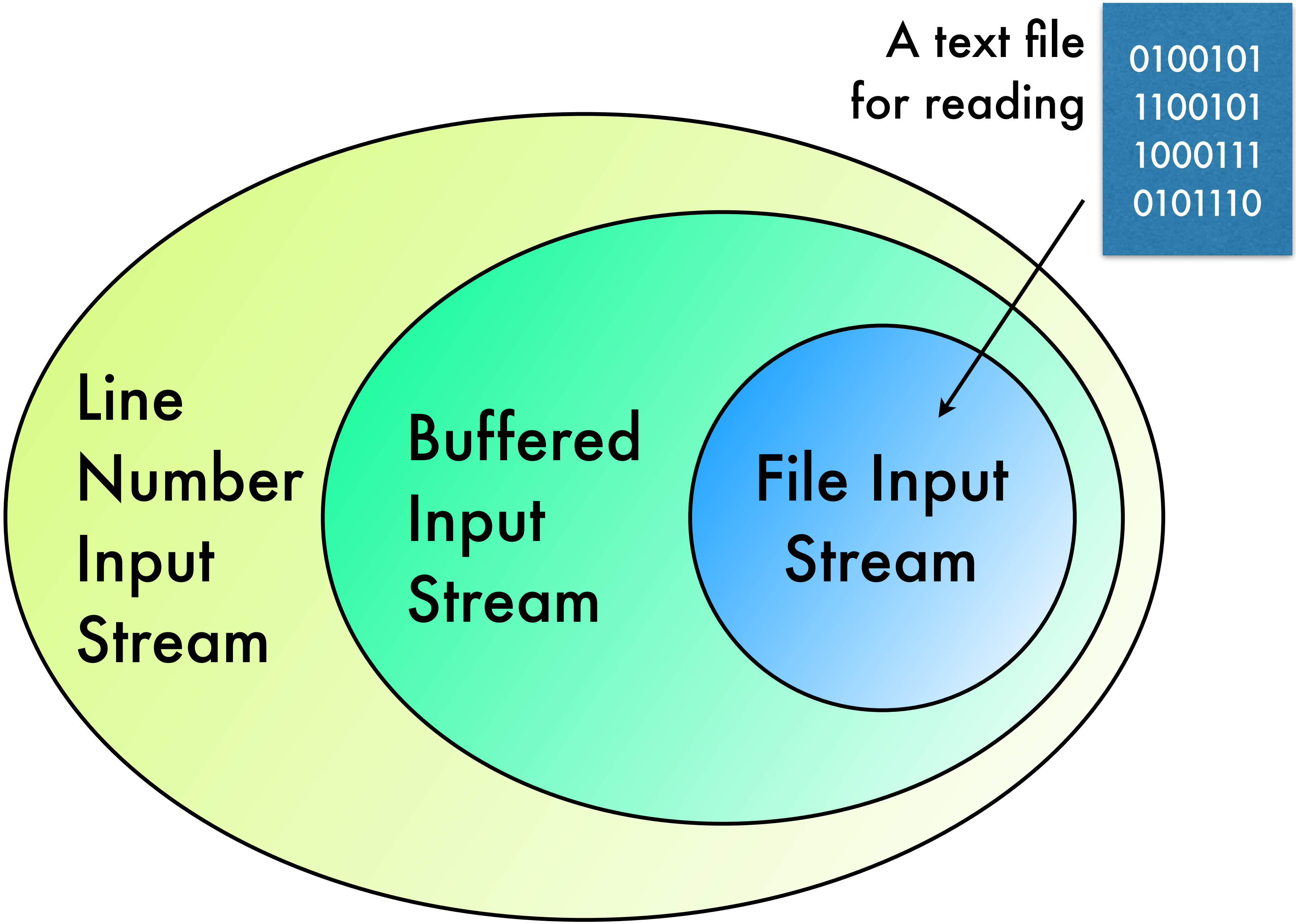
Whip

Beverage beverage

cost()
getDescription()



Component
methodA()
methodB()
// other methods
component
ConcreteComponent
methodA()
methodB()
// other methods
Decorator
methodA()
methodB()
// other methods
ConcreteDecoratorA
Component wrappedObj
methodA()
methodB()
newBehavior()
// other methods
ConcreteDecoratorB
Component wrappedObj
Object newState
methodA()
methodB()
// other methods



A text file for reading

0100101
1100101
1000111
0101110

Line Number Input Stream

Buffered Input Stream

File Input Stream