Design Patterns BOOTCAMP

with Eric Freeman & Elisabeth Robson

Your Instructors

Eric Freeman

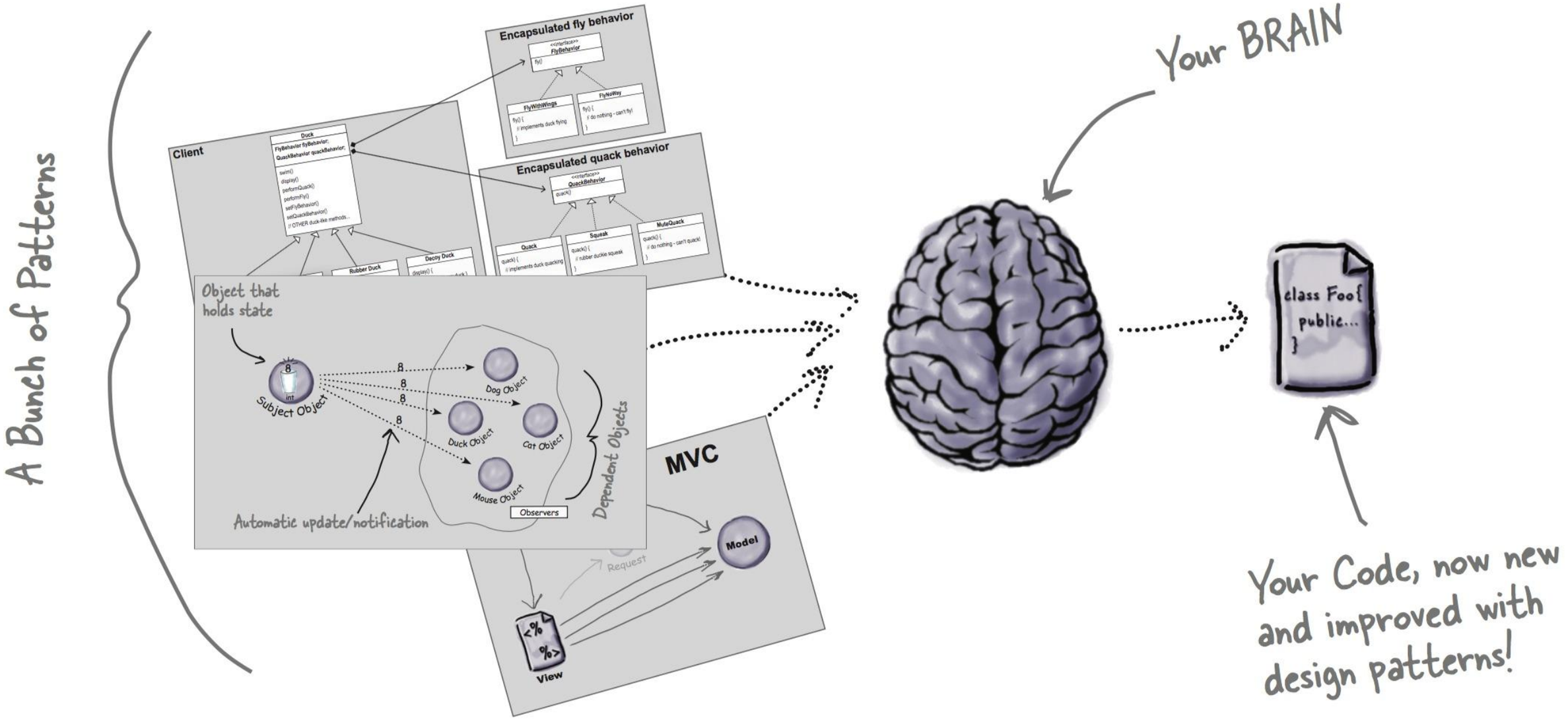
Elisabeth Robson

Logistics & Schedule

3 hours, 2 days

10 minute breaks approximately on the hour.

How to think about Design Patterns



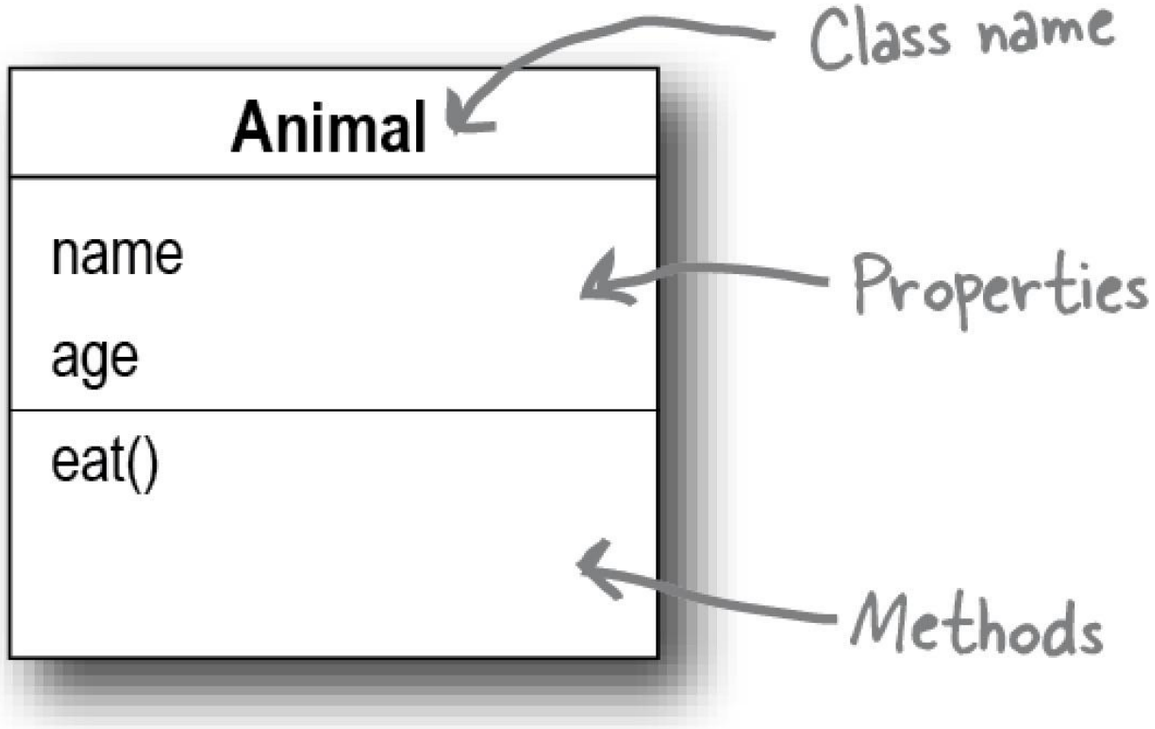
What's in it for You?

- Reuse hard-fought design experience of others, don't reinvent the wheel
- Create more maintainable software that is resilient to change
- Better understand underlying OO design principles
- More quickly use the patterns that are adopted across common frameworks and libraries.
- Turbo charge your team with better communication

A Quick Object-Oriented Review

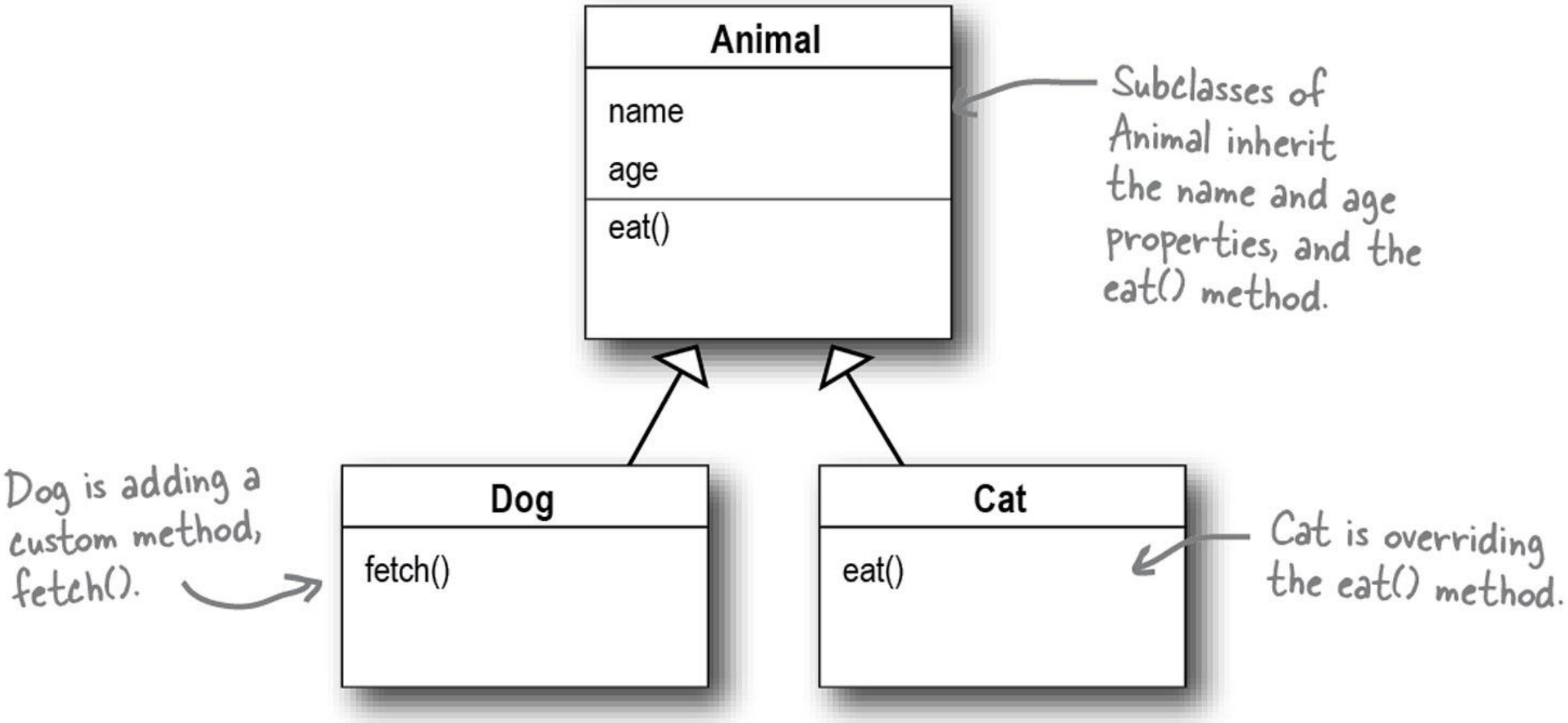
Class

A class is a template for creating objects that provides properties for state and methods for behavior.



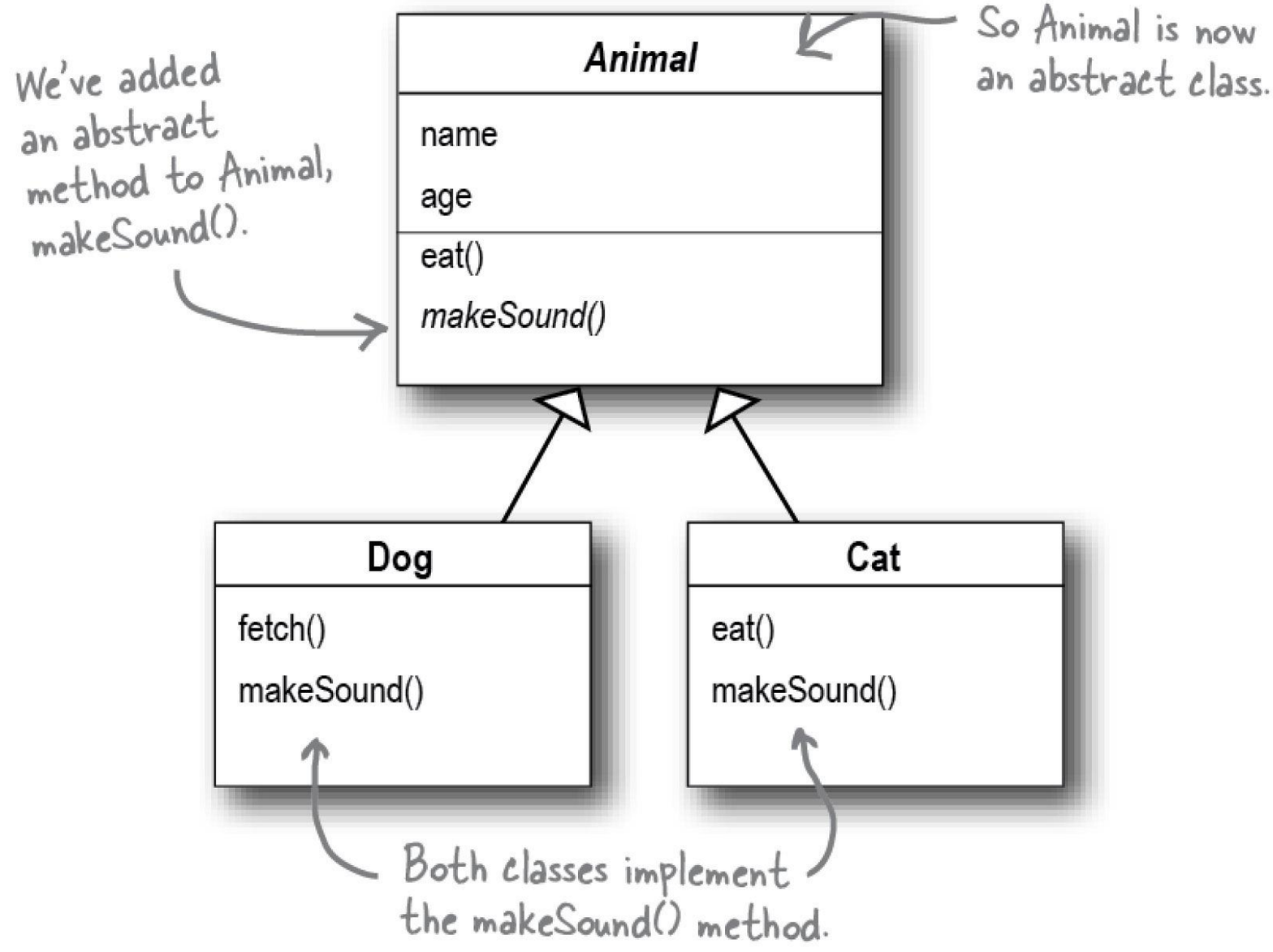
Inheritance

We can extend a class through subclassing.



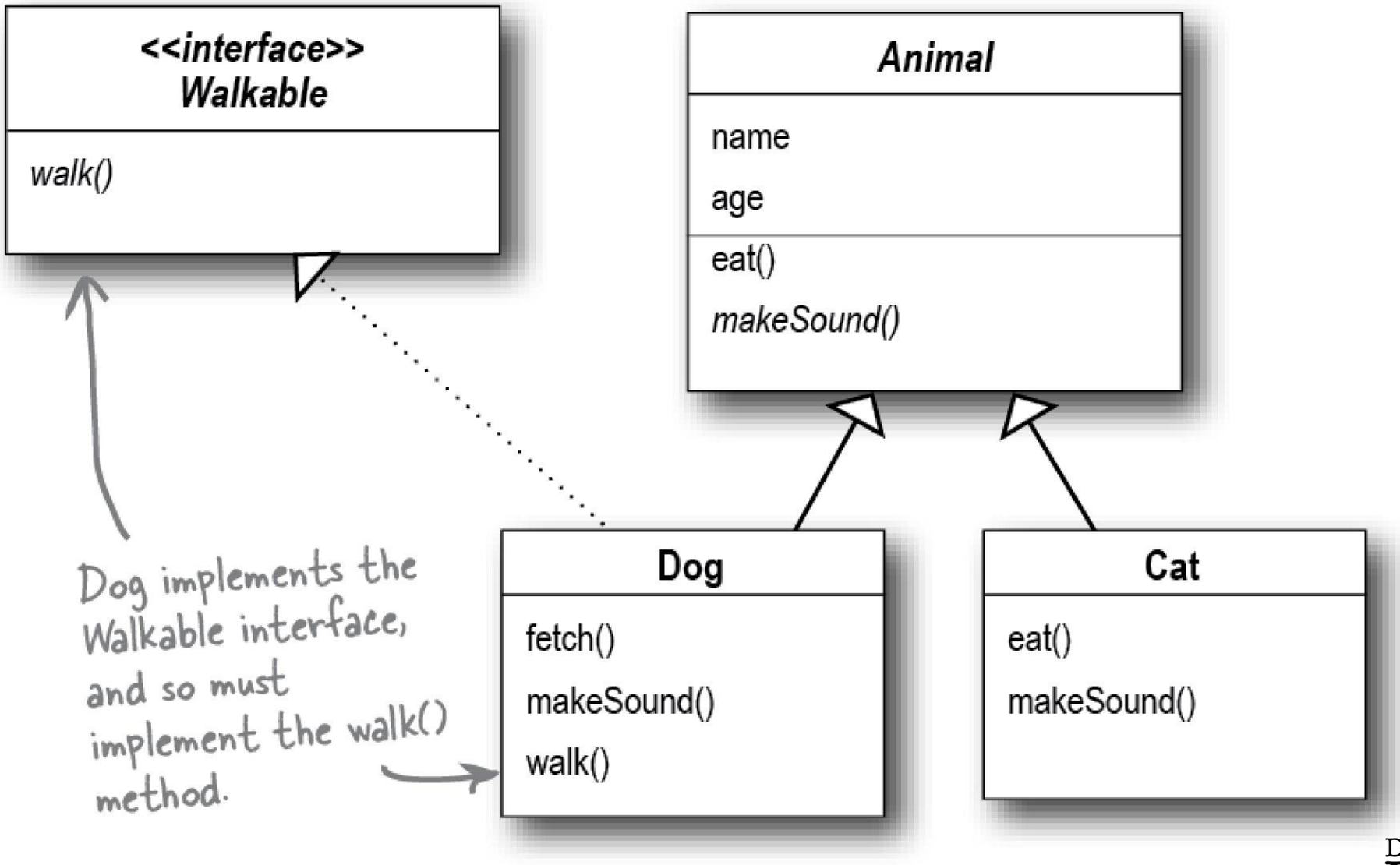
Abstract Class

An abstract class includes one or more methods that must be implemented by concrete subclasses.



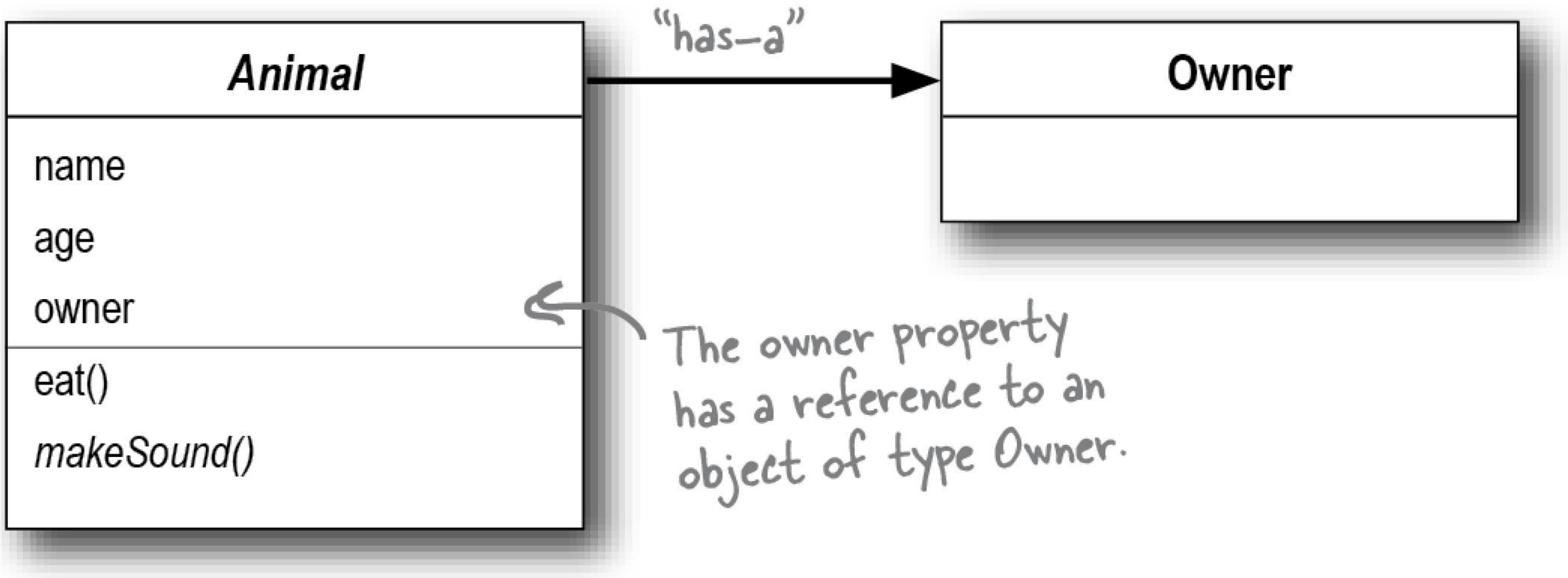
Interface

Interfaces contain no implementation. They allow classes to formally declare the behavior they are going to provide.



Composition

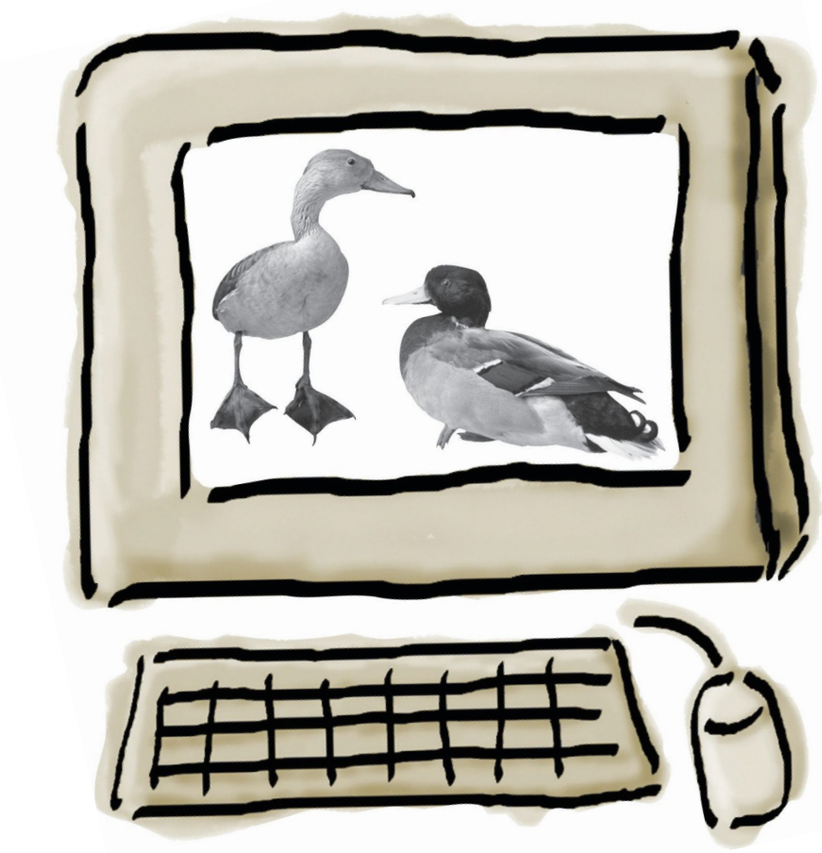
We can also associate classes through composition.



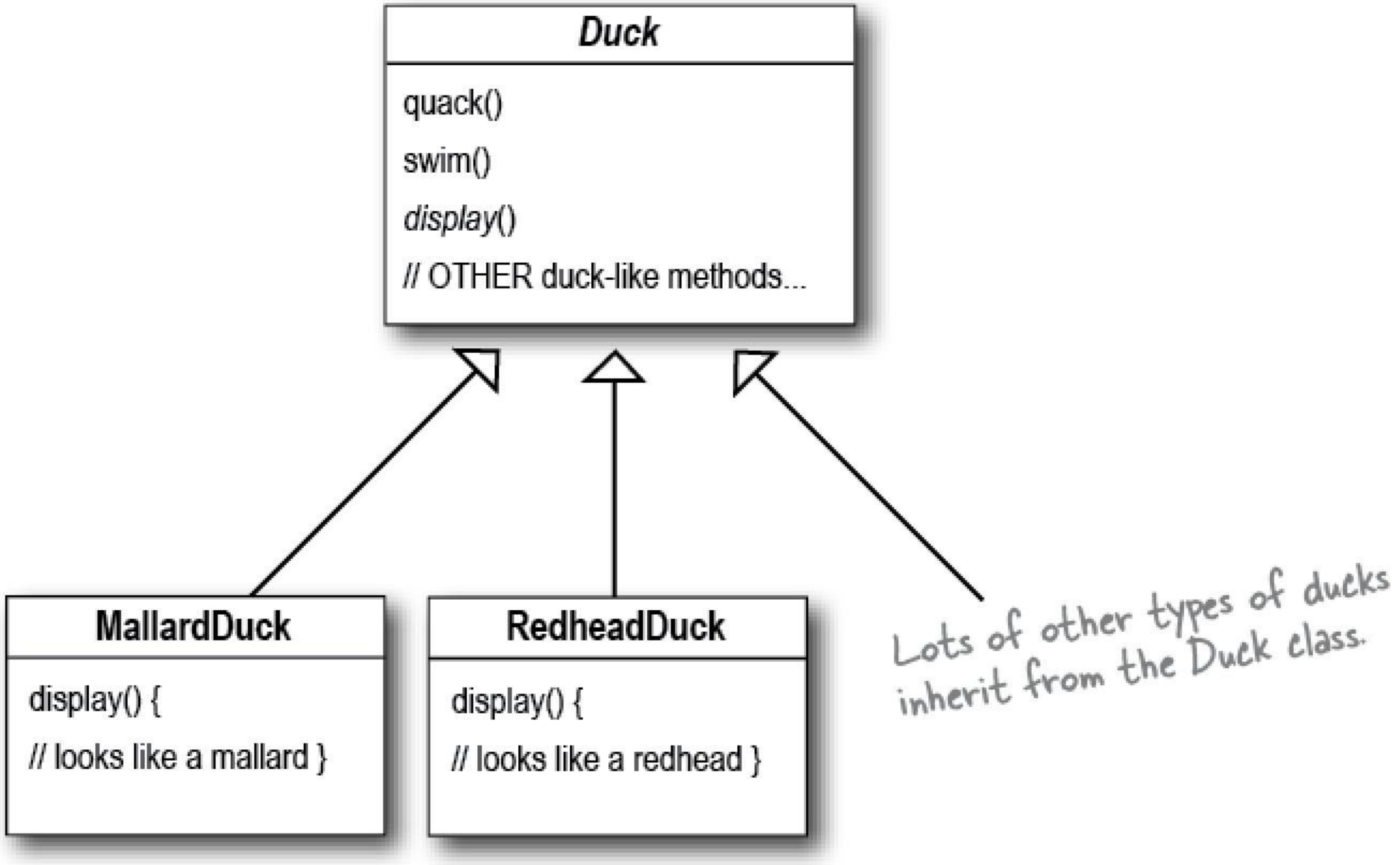
Diving in...

Introducing SimUDuck

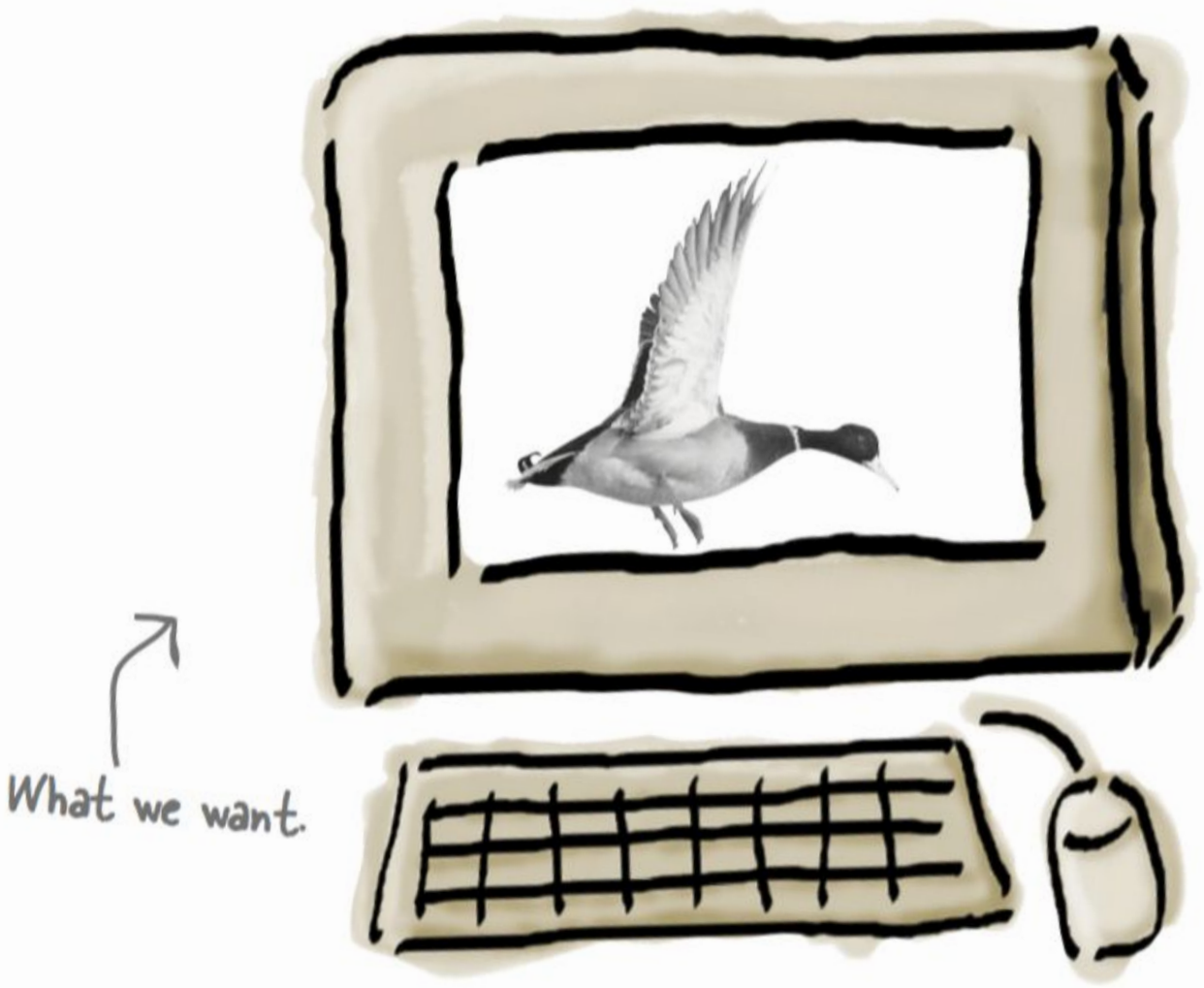
You now work for a company that makes a highly successful simulation game, SimUDuck. The initial designers of the system used standard OO techniques in their design.



The Design

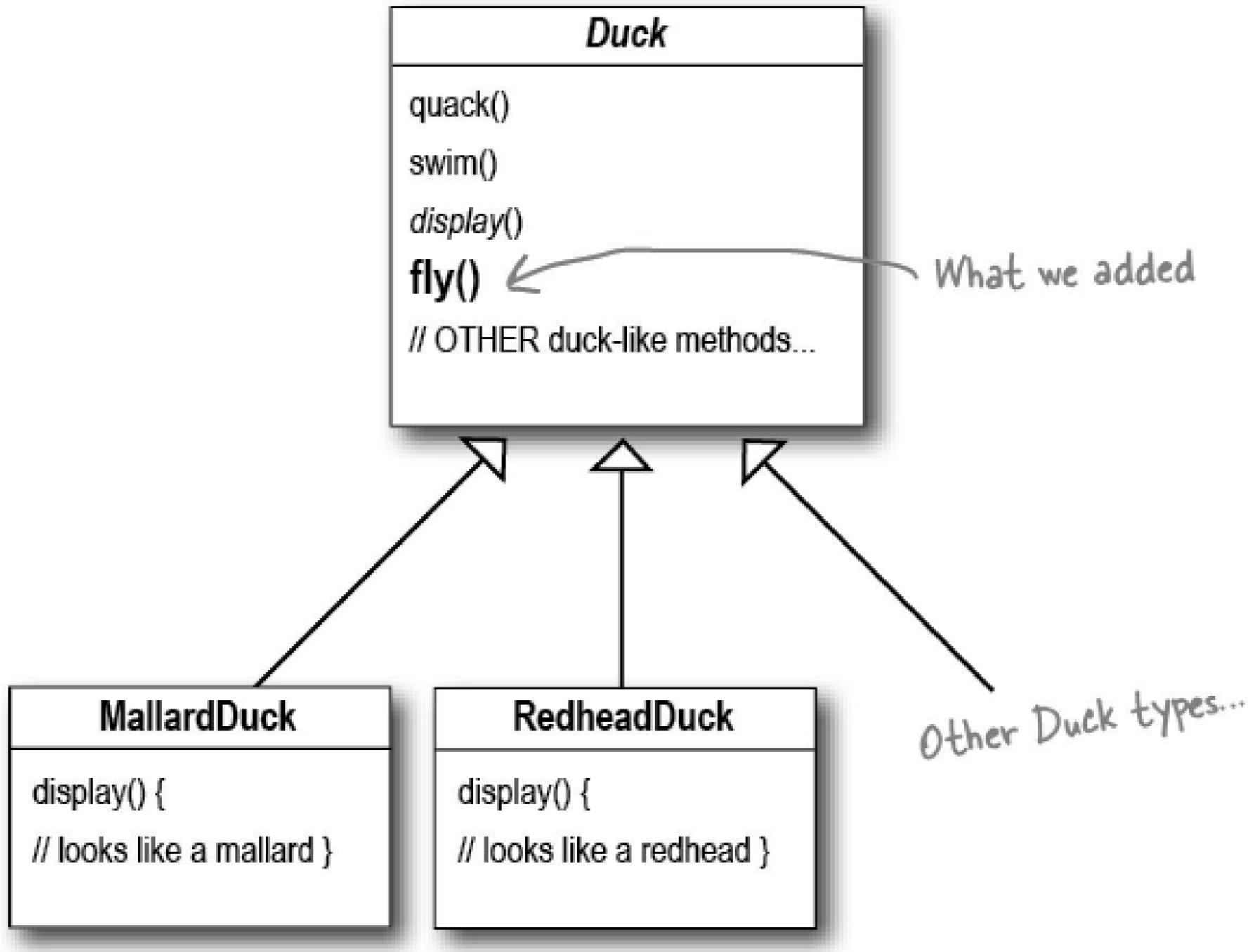

Duck
quack()
swim()
display()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
Lots of other types of ducks inherit from the Duck class.

A New Requirement



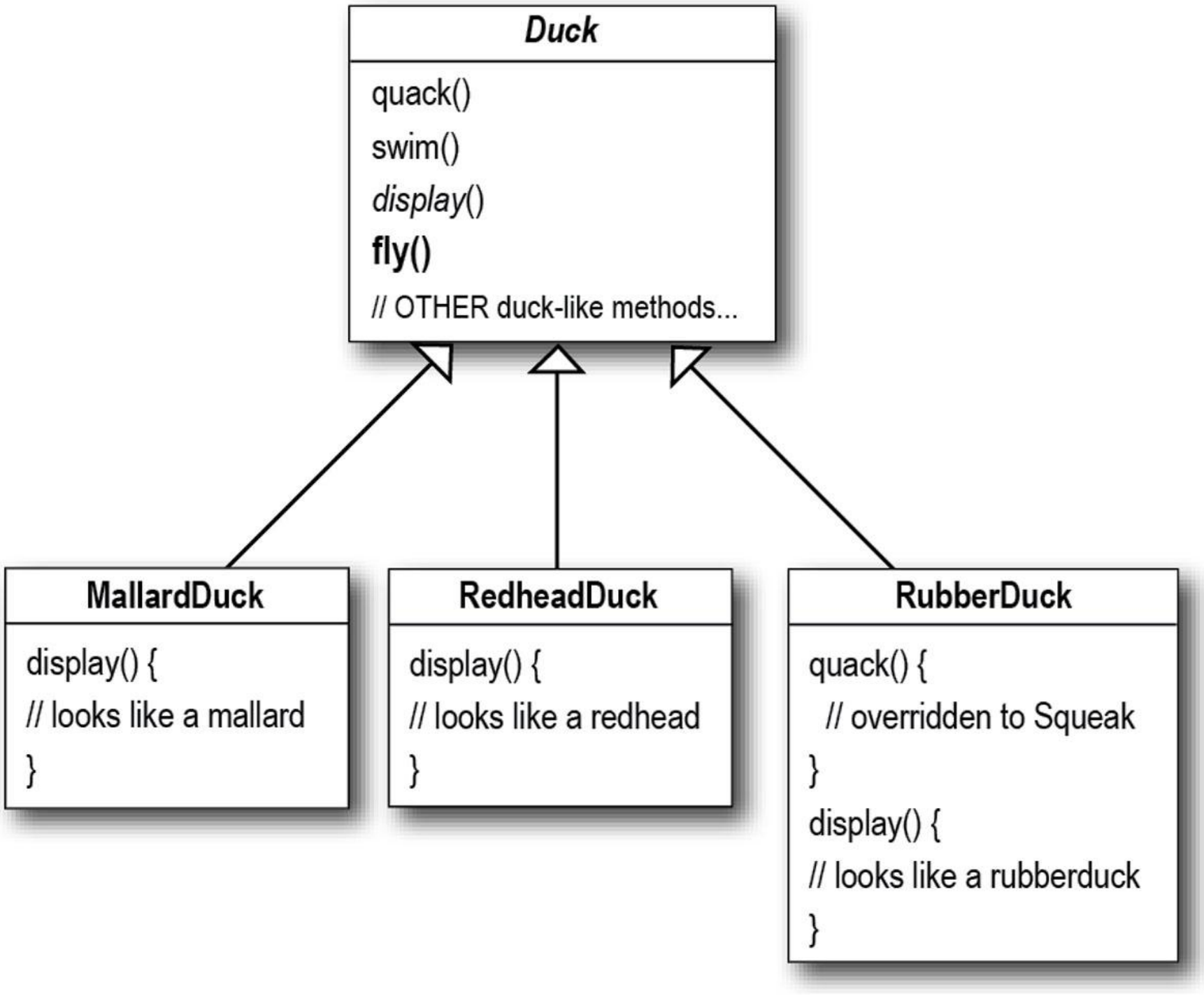
The executives have decided the new great feature is to make ducks fly, and of course your manager told the execs it would be no problem!

What we Want


Duck
quack()
swim()
display()
fly()
// OTHER duck-like methods...
What we added
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
Other Duck types...

Then something went horribly wrong…

What Happened?



Where did the design go wrong? Take some time and rework the design so that flying ducks can fly, non-flying ducks can't fly, and adding both kinds of ducks (flying and non-flying) is as painless as possible in the future.

What did you come up with?
How about overriding fly()?

RubberDuck

quack() { // squeak}

display() { // rubber duck }

fly() {

// override to do nothing

}

DecoyDuck

quack() {
// override to do nothing
}

display() { // decoy duck}

fly() {
// override to do nothing
}

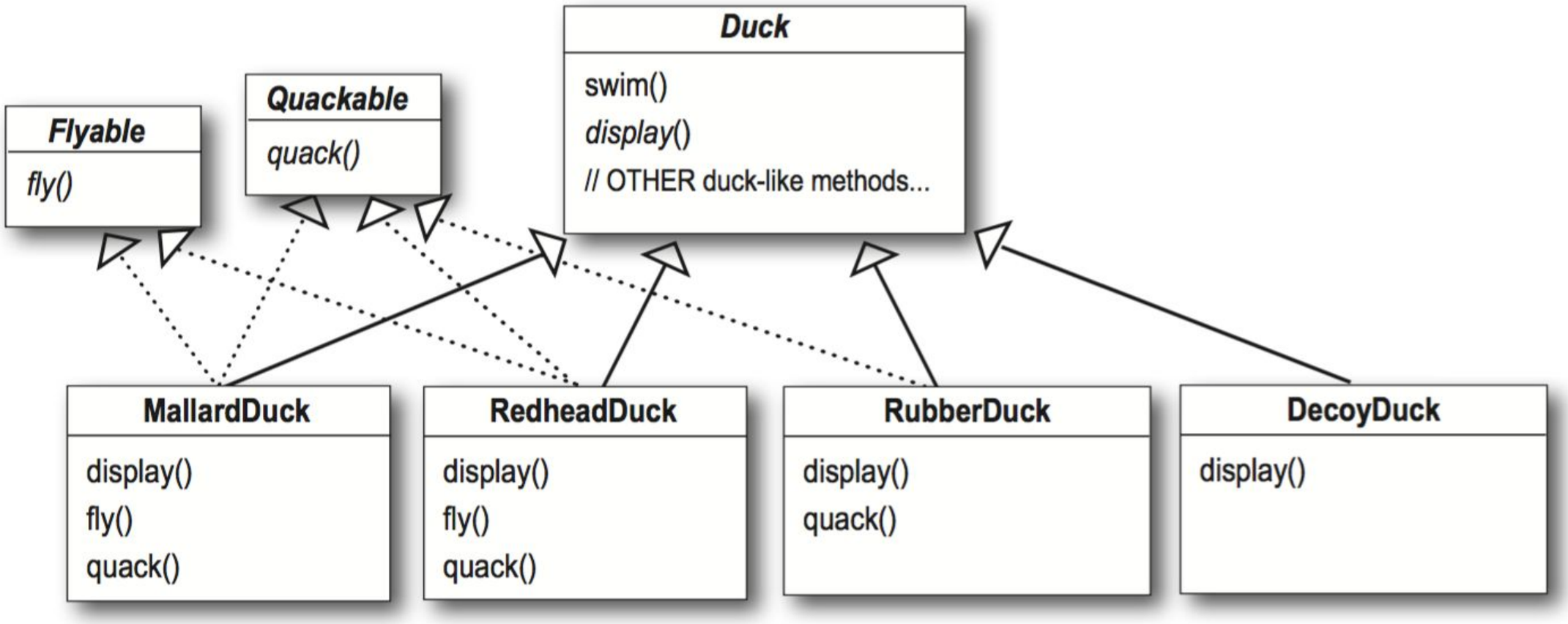
Here's another class in the hierarchy; notice that like RubberDuck, it doesn't fly, but it also doesn't quack.

What are the disadvantages of continuing to override methods from the superclass that don't work for the subclasses? Type your answers into chat.

What are the disadvantages of continuing to override methods from the superclass that don't work for the subclasses? Type your answers into chat.

- Code is duplicated across subclasses.
- Runtime behavior changes are difficult.
- Hard to gain knowledge of all duck behaviors.
- Changes can unintentionally affect other ducks.

Hello, use an Interface!



What do YOU think about this design?

What to Do?

- Not all of the subclasses should have flying or quacking behavior that's the same as the superclass, so inheritance isn't the right answer.

- And having the subclasses implement Flyable and/or Quackable completely destroys code reuse.

- Either way we have a maintenance nightmare!

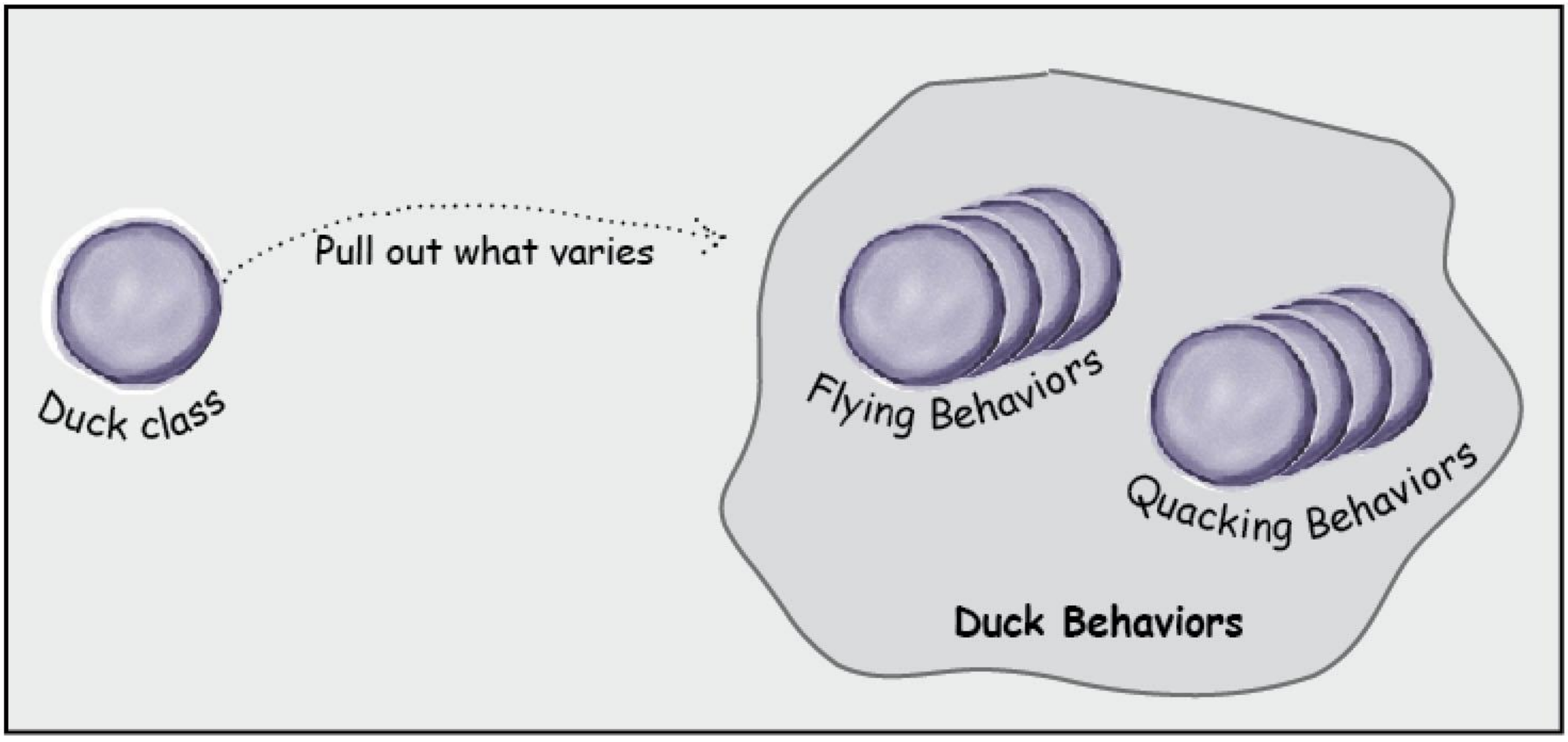
How about a little design inspiration?

Design Principle

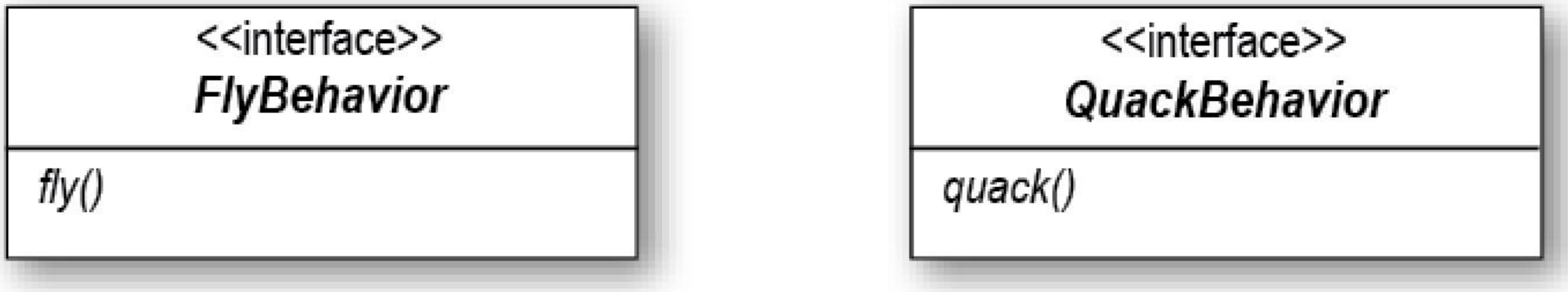
Identify the aspects of your application that vary and separate them from what stays the same.

Take a minute to identify the aspects of the ducks that “varies”. Think about all the ways potential new requirements are going to change the design. Type your answers into chat.

Separating what Changes



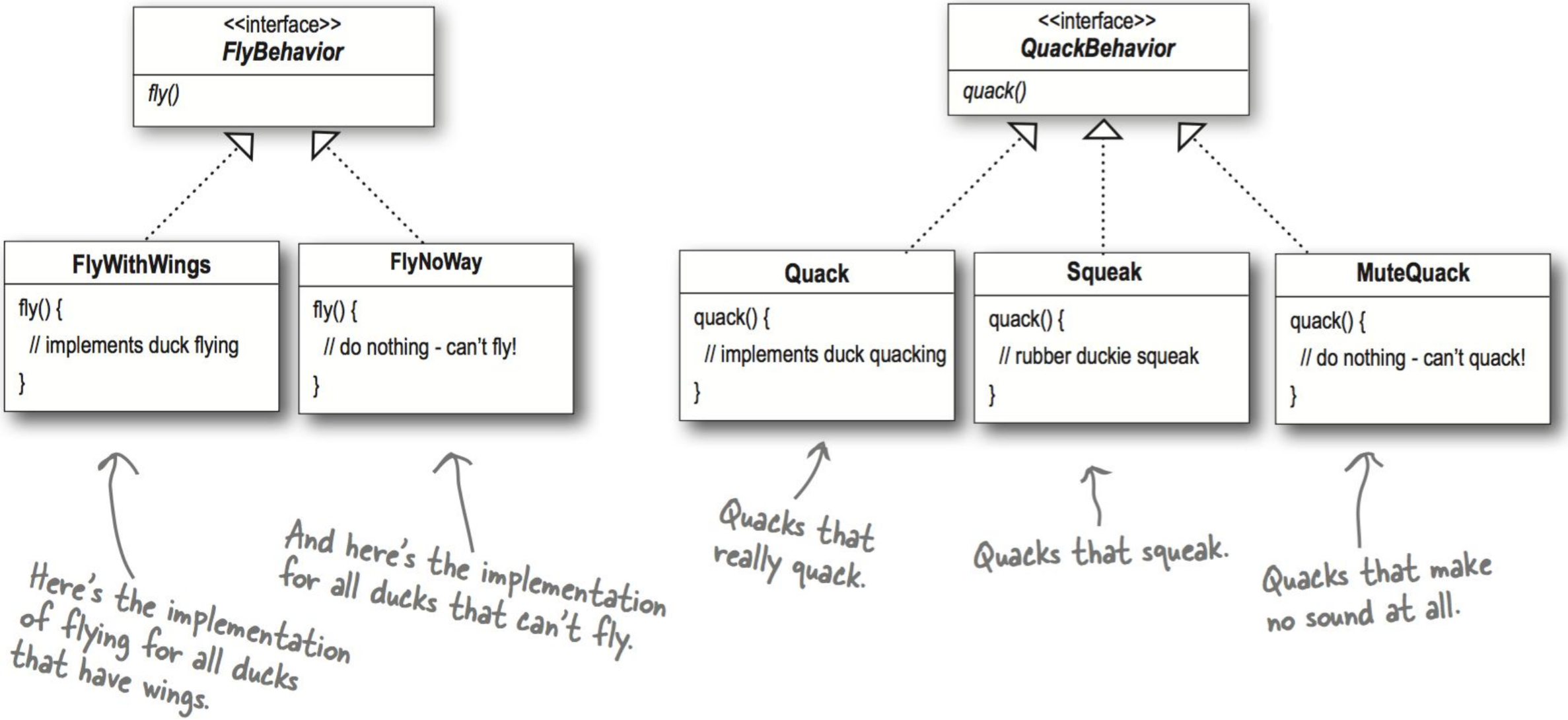
Separating what Changes with an Interface



Create two interfaces, one for flying behavior and one for quacking behavior.

Adding Concrete Classes

Implement some concrete behaviors for the fly and quack behavior interfaces.

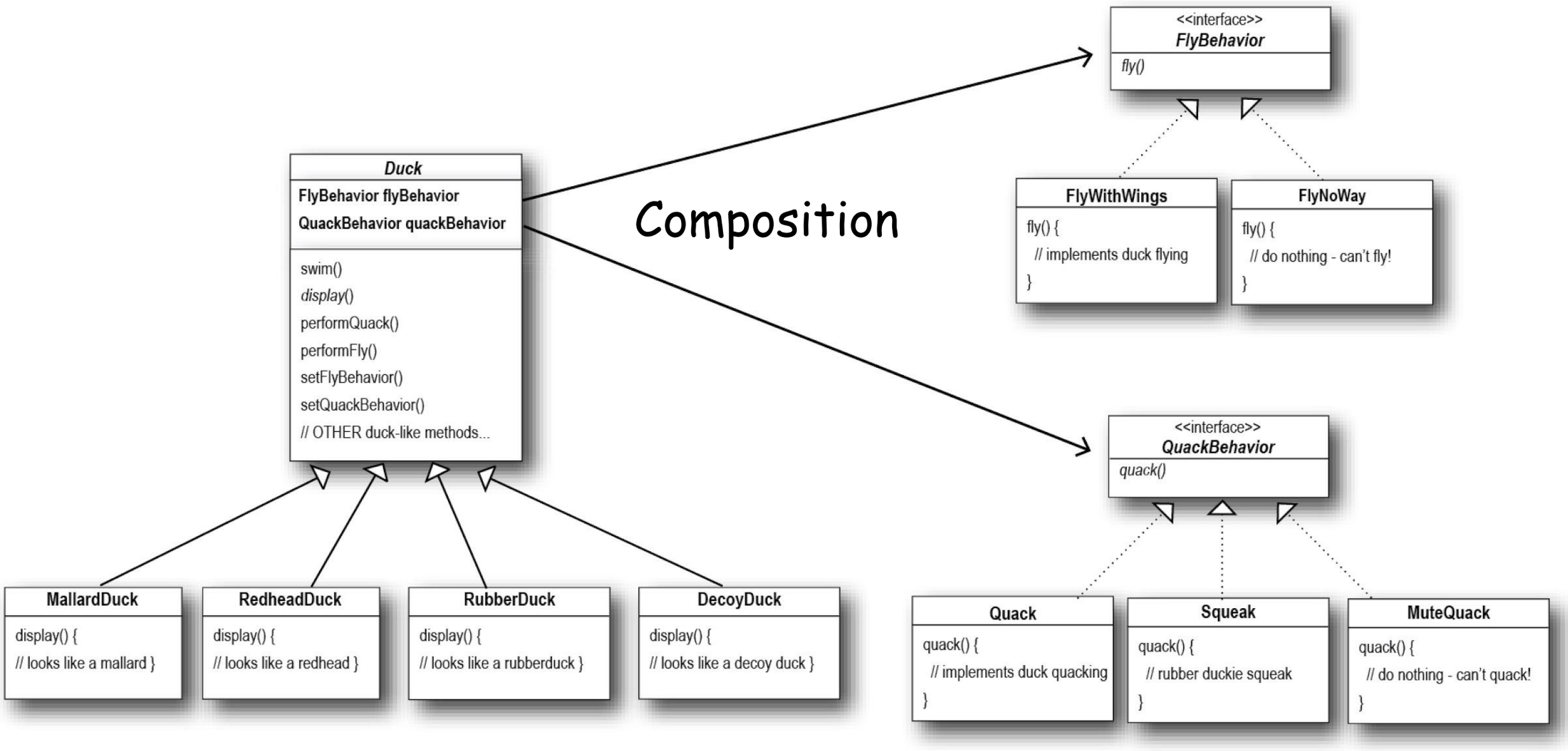


HAS-A is better than IS-A
Or...

Design Principle

Favor composition over inheritance.

Using composition to add behavior



What have we achieved?

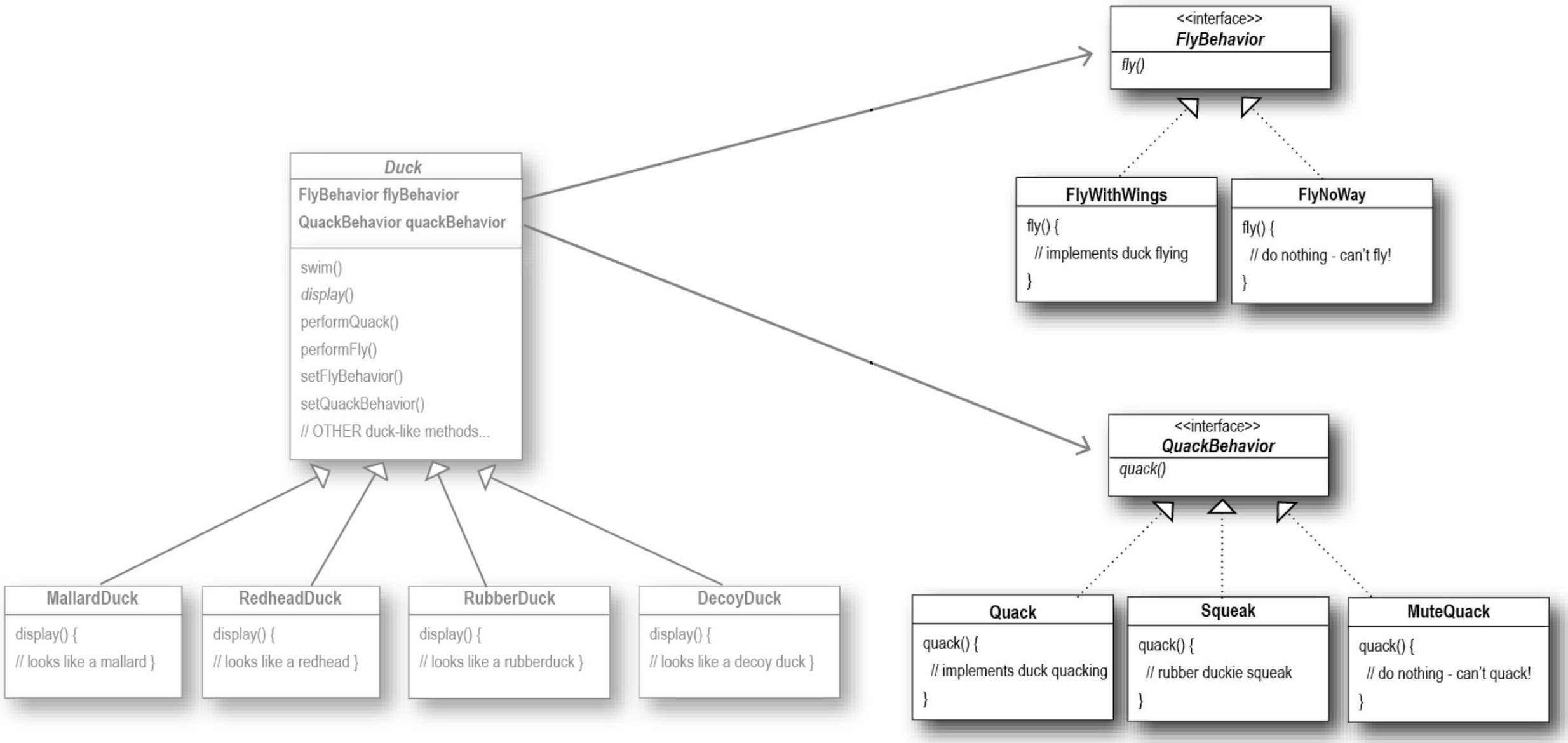
- All our duck behaviors are in one place.
- Any duck can be assigned any of the methods of flying.
- Easy to add new behaviors.
- Easy to add new ducks.
- Can change behavior at any time.

Meet The Strategy Pattern

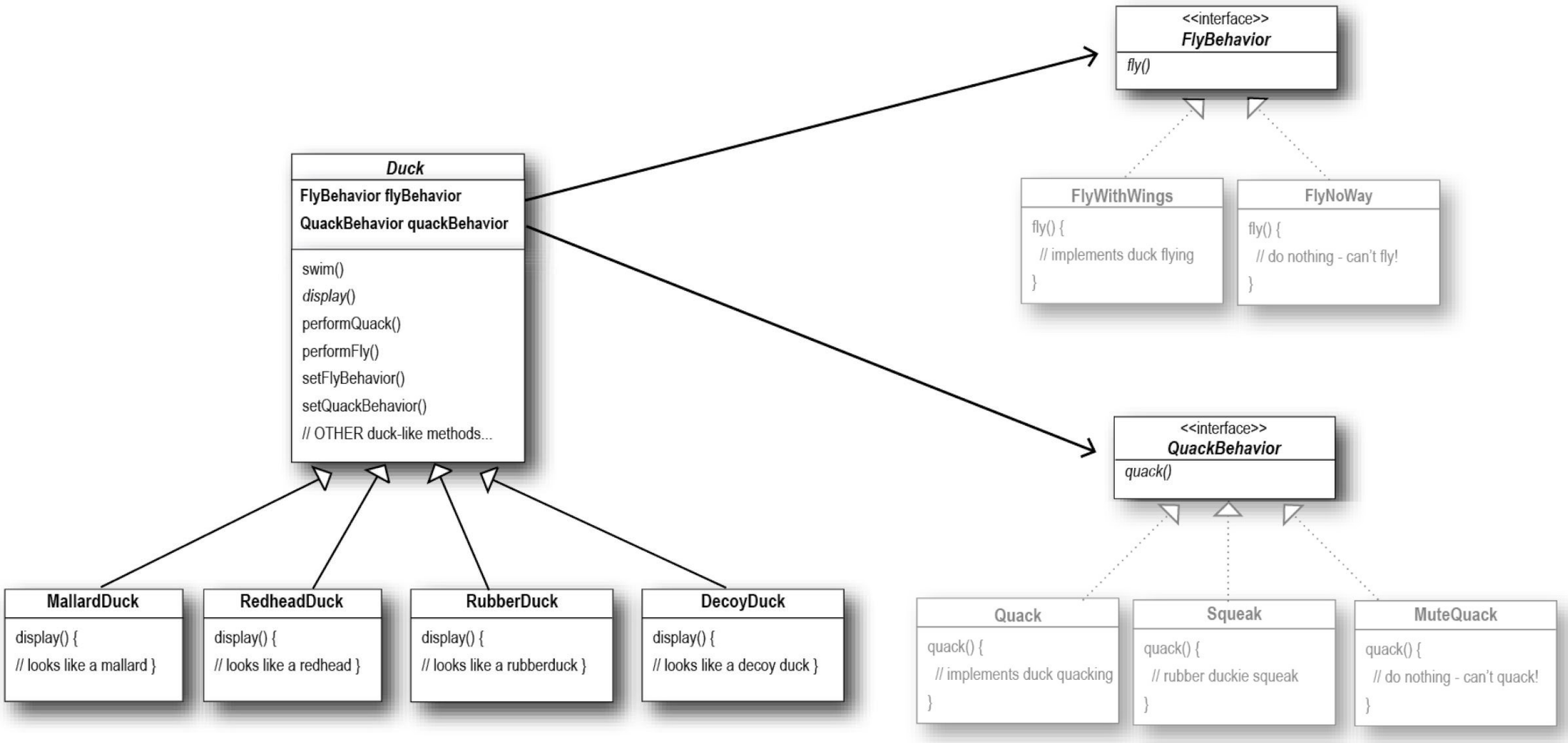
The Strategy Pattern

Defines a family of algorithms encapsulating each one, and makes them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

Defines a family of algorithms encapsulating each one, and makes them interchangeable



Strategy lets the algorithm vary independently from the clients that use it.



Some Code...

Duck

```
public abstract class Duck {
   FlyBehavior flyBehavior;
   QuackBehavior quackBehavior;
   abstract void display();
   public Duck() {
   }
   public void setFlyBehavior(FlyBehavior fb) {
      flyBehavior = fb;
   }
   public void setQuackBehavior(QuackBehavior qb) {
      quackBehavior = qb;
   }
   public void performFly() {
      flyBehavior.fly();
   }
   public void performQuack() {
      quackBehavior.quack();
   }
   public void swim() {
      System.out.println("All ducks float, even decoys!");
   }
}
```

Fly Behaviors

```
public interface FlyBehavior {
   public void fly();
}

public class FlyWithWings implements FlyBehavior {
   public void fly() {
      System.out.println("I'm flying!!");
   }
}

public class FlyNoWay implements FlyBehavior {
   public void fly() {
      System.out.println("I can't fly");
   }
}
```

Quack Behaviors

```
public interface QuackBehavior {
   public void quack();
}

public class Quack implements QuackBehavior {
   public void quack() {
      System.out.println("Quack");
   }
}
```

```
public class MuteQuack implements QuackBehavior {
   public void quack() {
      System.out.println("<< Silence >>");
   }
}

public class Squeak implements QuackBehavior {
   public void quack() {
      System.out.println("Squeak");
   }
}
```

The Mallard Duck

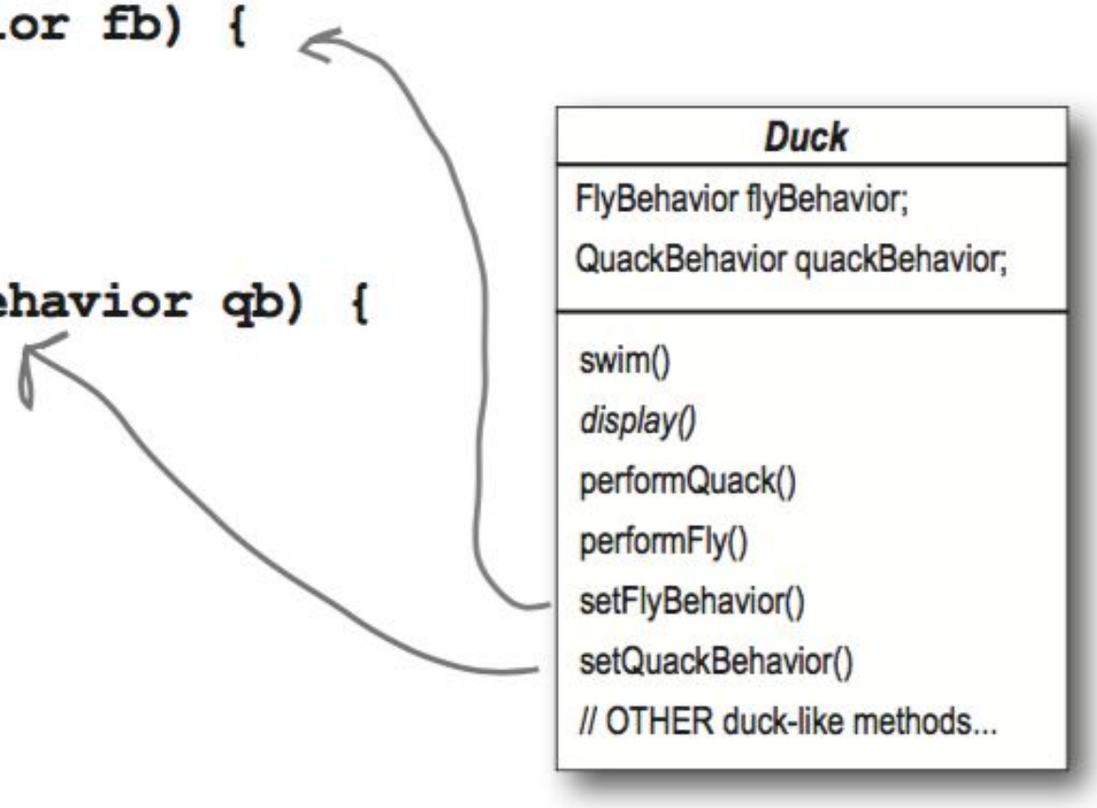
```
public class MallardDuck extends Duck {

   public MallardDuck() {

      quackBehavior = new Quack();
      flyBehavior = new FlyWithWings();

   }

   public void display() {
      System.out.println("I'm a real Mallard duck");
   }
}
```

Duck Simulator

```
public class MiniDuckSimulator {

   public static void main(String[] args) {

      Duck mallard = new MallardDuck();
      mallard.performQuack();
      mallard.performFly();

   }
}
```

Dynamic Behavior

```
public void setFlyBehavior(FlyBehavior fb) {
    flyBehavior = fb;
}

public void setQuackBehavior(QuackBehavior qb) {
    quackBehavior = qb;
}
```



Add a new Duck

Make a new Duck type

```
public class ModelDuck extends Duck {
    public ModelDuck() {
        flyBehavior = new FlyNoWay();
        quackBehavior = new Quack();
    }

    public void display() {
        System.out.println("I'm a model duck");
    }
}
```

Our model duck begins life grounded... without a way to fly.

Make a new FlyBehavior type

That's okay, we're creating a rocket-powered flying behavior.

```
public class FlyRocketPowered implements FlyBehavior {
    public void fly() {
        System.out.println("I'm flying with a rocket!");
    }
}
```

Duck Simulator

Change the test class (MiniDuckSimulator.java), add the ModelDuck, and make the ModelDuck rocket-enabled

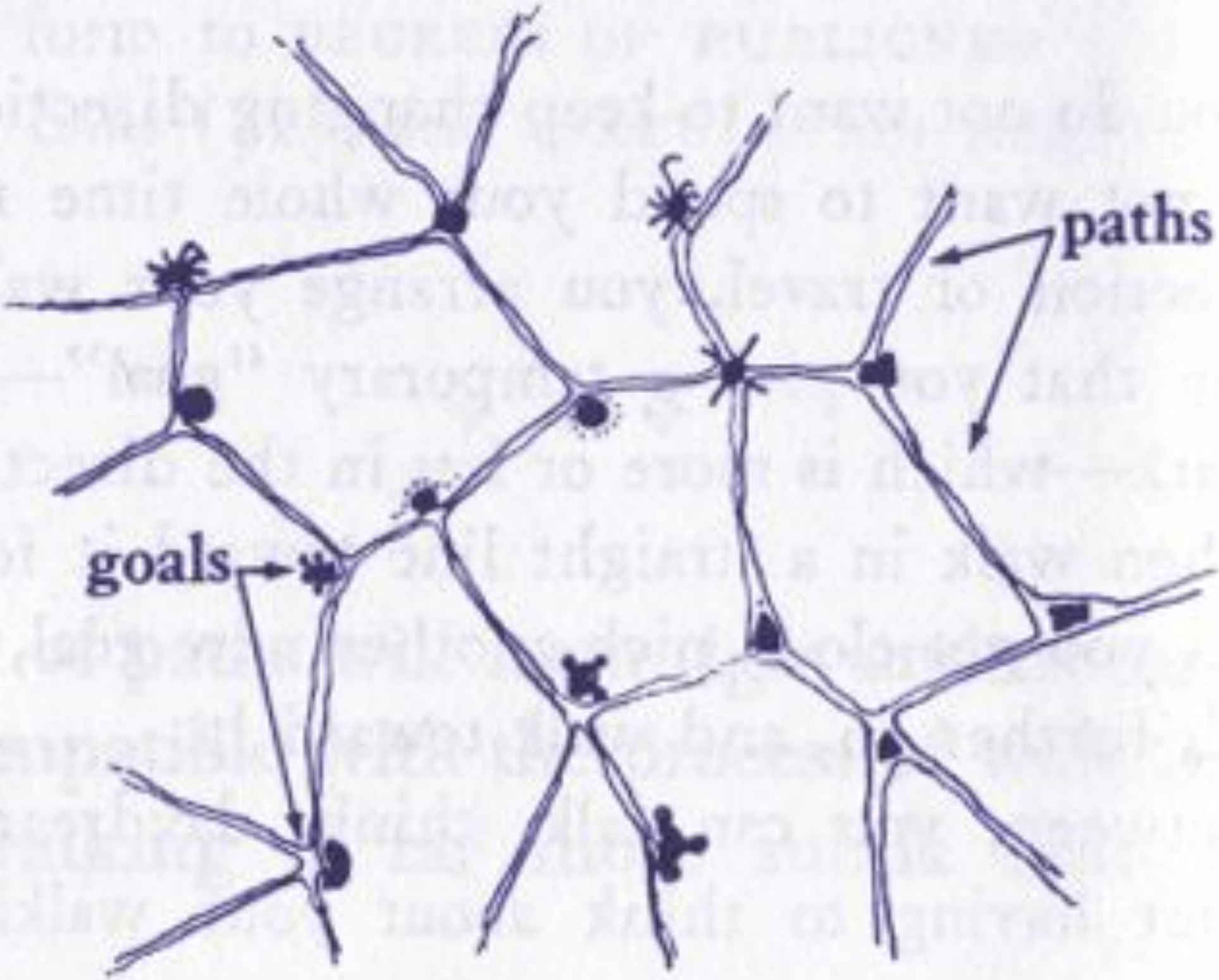
```
public class MiniDuckSimulator {
    public static void main(String[] args) {
        Duck mallard = new MallardDuck();
        mallard.performQuack();
        mallard.performFly();

        Duck model = new ModelDuck();
        model.performFly();
        model.setFlyBehavior(new FlyRocketPowered());
        model.performFly();
    }
}
```

Design Patterns, the Big Picture

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

Christopher Alexander

Pattern #120: Paths and Goals



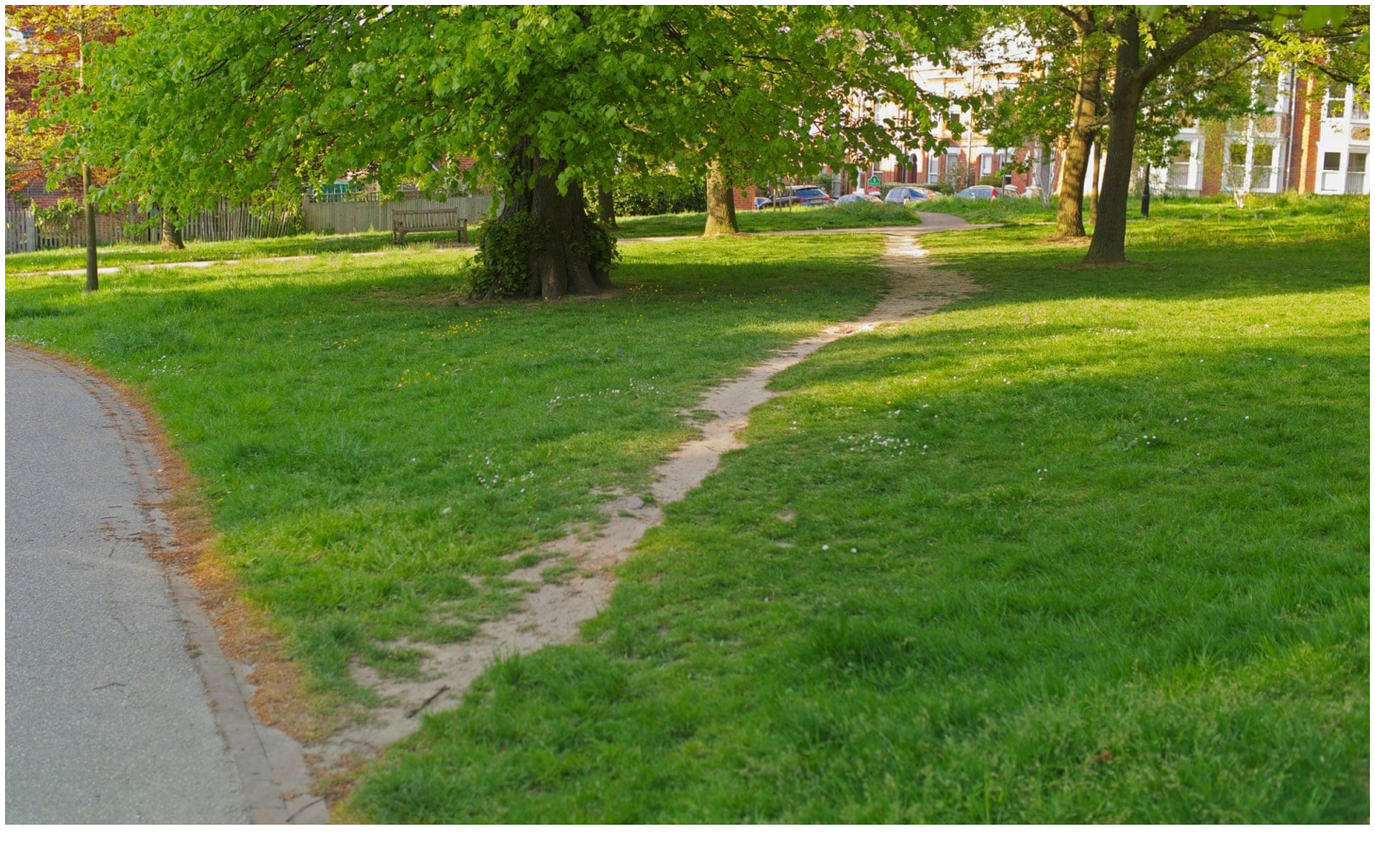
Pattern #120: Paths and Goals

Paths are a way to connect goals that are already defined and designed. All the ordinary things in the outdoors - trees, fountains, entrances, gateways, seats, statues, a swing, an outdoor room - can be the goals.

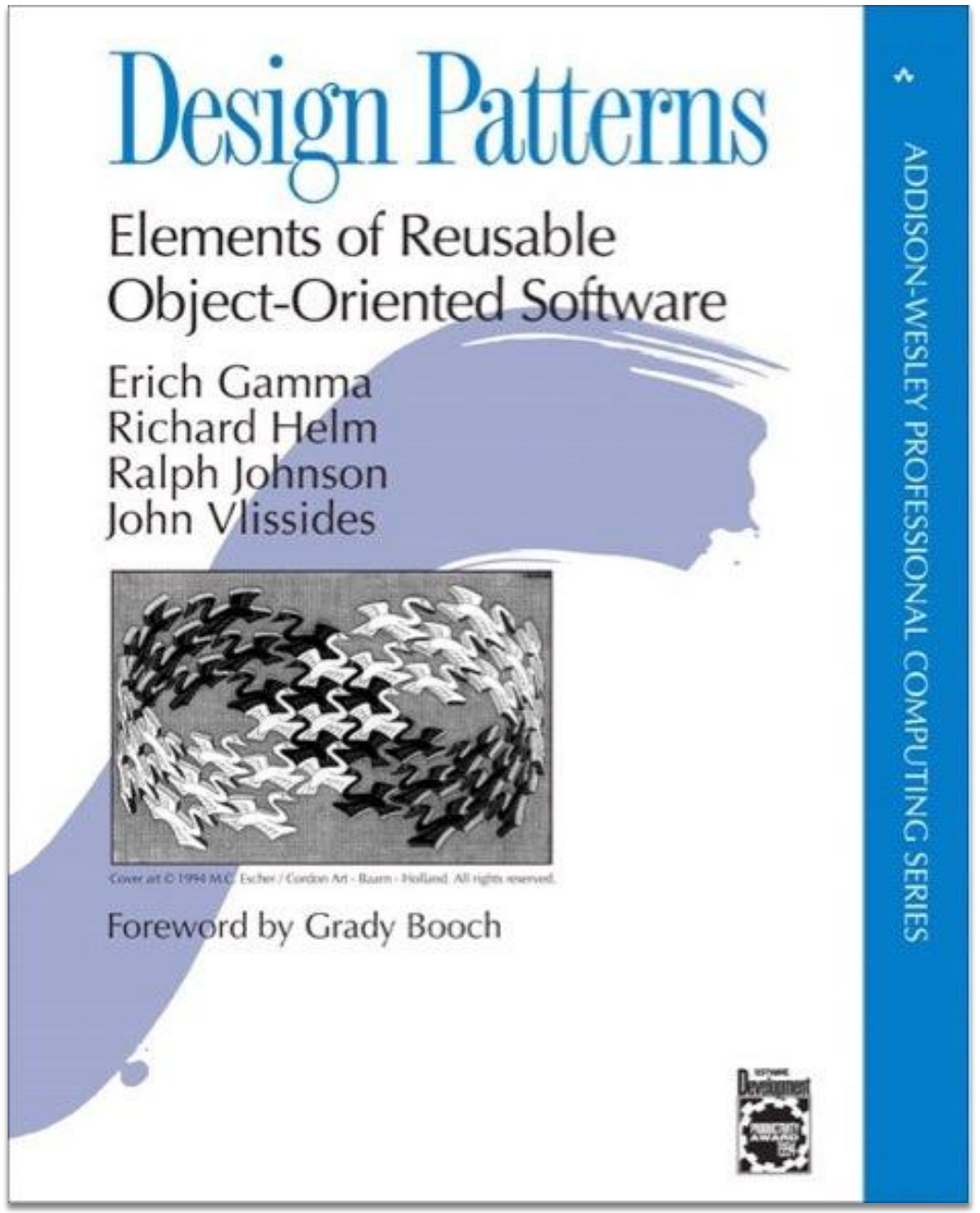
Therefore: to lay out paths, first place goals at natural points of interest. Then connect the goals to one another to form the paths. The paths may be straight, or gently curving between goals.

One “Implementation”

(1) Plant grass everywhere.
(2) Allow students to walk and form natural paths (by killing the grass).
(3) Pave or make proper trails from the natural pathways.



So what does all this have to do with SOFTWARE?



All patterns in a catalog start with a name. The name is a vital part of a pattern – without a good name, a pattern can't become part of the vocabulary that you share with other developers.

The motivation gives you a concrete scenario that describes the problem and how the solution solves the problem.

The applicability describes situations in which the pattern can be applied.

The participants are the classes and objects in the design. This section describes their responsibilities and roles in the pattern.

The consequences describe the effects that using this pattern may have: good and bad.

Implementation provides techniques you need to use when implementing this pattern, and

STRATEGY

Object Behavioral

Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Motivation

Many algorithms exist for breaking a stream of text into lines. Hard-wiring all such algorithms into the classes that require them isn't desirable for several reasons:

* Clients that need linebreaking get more complex if they include the line-breaking code. That makes clients bigger and harder to maintain, especially if they support multiple linebreaking algorithms.

* Different algorithms will be appropriate at different times. We don't want to support multiple linebreaking algorithms if we don't use them all.

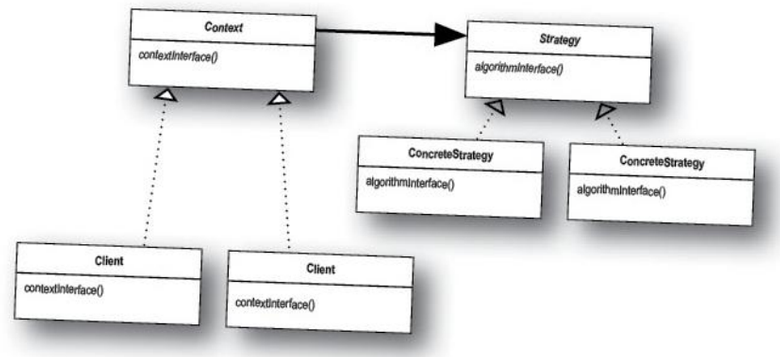
Applicability

Use the Strategy pattern when:

* many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.

* you need different variants of an algorithm.

Structure



Participants

Strategy - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.

Concrete Strategy - implements the algorithm using the Strategy interface.

Context - is configured with a ConcreteStrategy object; maintains a reference to a Strategy object; may define an interface that lets Strategy access its data.

Collaborations

Strategy and Context interact to implement the chosen algorithm.

A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively.

Consequences

The Strategy pattern has the following benefits and drawbacks:

* *Families of related algorithms.* Hierarchies of Strategy classes define a family of behaviors for contexts to reuse. Inheritance can help factor out common functionality of the algorithms.

* *An alterantive to subclassing.* You can subclass a Context class directly to give it different behaviors, but this hard-wires the behavior into the Context. It mixes the algorithm implementation with the Context's, making Context harder to understand, maintain, and extend. And you can't vary the algorithm dynamically.

* *Increased number of objects.* Strategies increase the number of objects in an application.

Implementation/Sample Code

Consider the following implementation issues:

* *Defining the Strategy and Context interfaces.* The Strategy and Context interfaces must give a ConcreteStrategy efficient access to any data it needs from a context. One approach is to have the Context pass data in parameters to Strategy operations. This keeps Strategy and Context decoupled, but it also means Context may pass data the Strategy doesn't need.

* *Making Strategy objects optional.* The Context class may be simplified if it's meaningful not to have a Strategy object. Context checks to see if it has a Strategy object before accessing it. If there is one, then Context uses it normally, otherwise, Context carries out default behavior.

This is the pattern's classification or category. We'll talk about these in a few pages.

The intent describes what the pattern does in a short statement. You can also think of this as the pattern's definition (just like we've been using in this book).

The structure provides a diagram illustrating the relationships among the classes that participate in the pattern.

Collaborations tells us how the participants work together in the pattern.

STRATEGY

Object Behavioral

Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Motivation

Many algorithms exist for breaking a stream of text into lines. Hard-wiring all such algorithms into the classes that require them isn't desirable for several reasons:

* Clients that need linebreaking get more complex if they include the line-breaking code. That makes clients bigger and harder to maintain, especially if they support multiple linebreaking algorithms.

* Different algorithms will be appropriate at different times. We don't want to support multiple linebreaking algorithms if we don't use them all.

Applicability

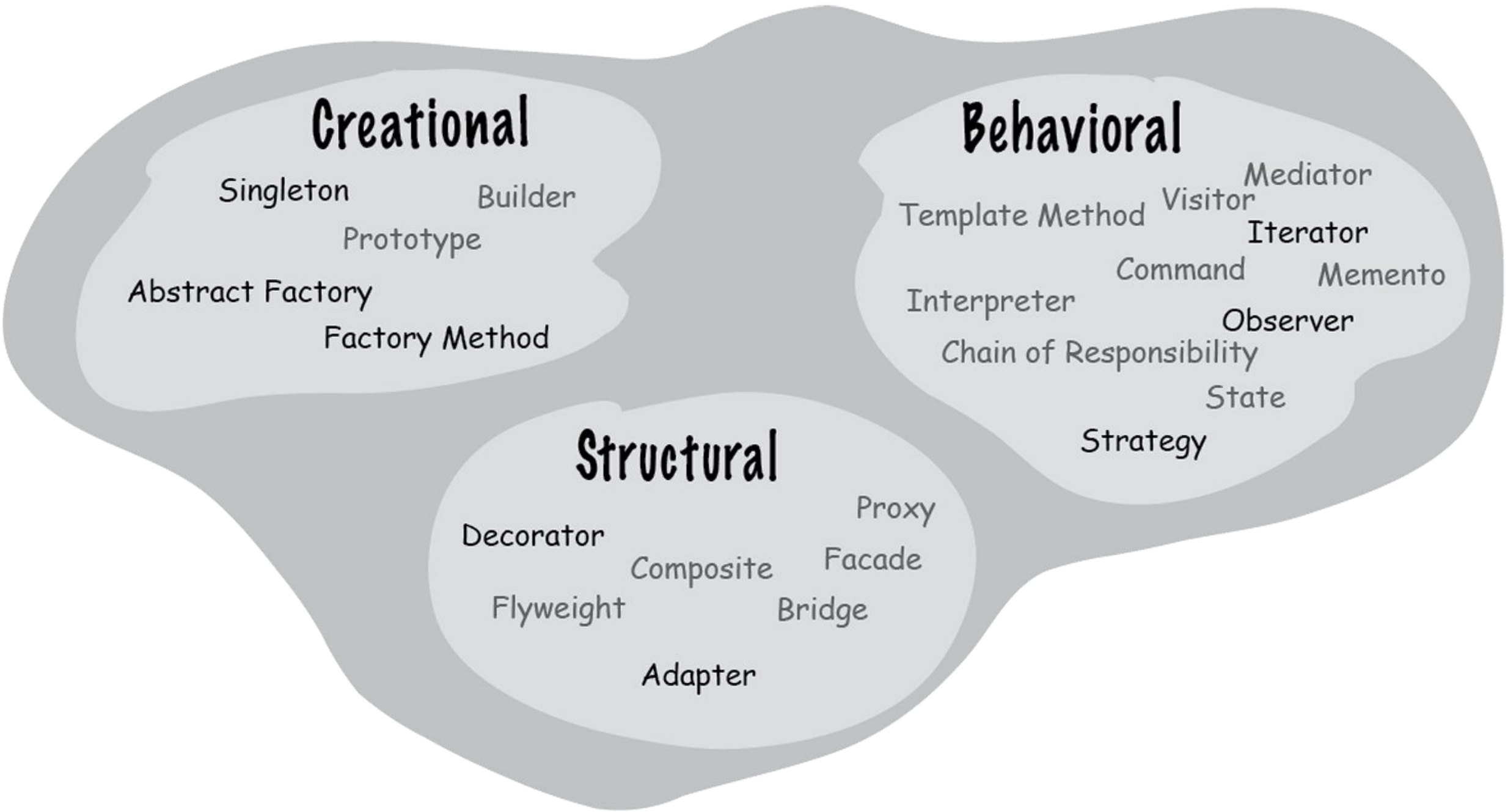
Use the Strategy pattern when:

* many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.

* you need different variants of an algorithm.

Structure

Pattern Categories



STRATEGY

Object Behavioral

Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Motivation

Many algorithms exist for breaking a stream of text into lines. Hard-wiring all such algorithms into the classes that require them isn't desirable for several reasons:

* Clients that need linebreaking get more complex if they include the line-breaking code. That makes clients bigger and harder to maintain, especially if they support multiple linebreaking algorithms.

* Different algorithms will be appropriate at different times. We don't want to support multiple linebreaking algorithms if we don't use them all.

Applicability

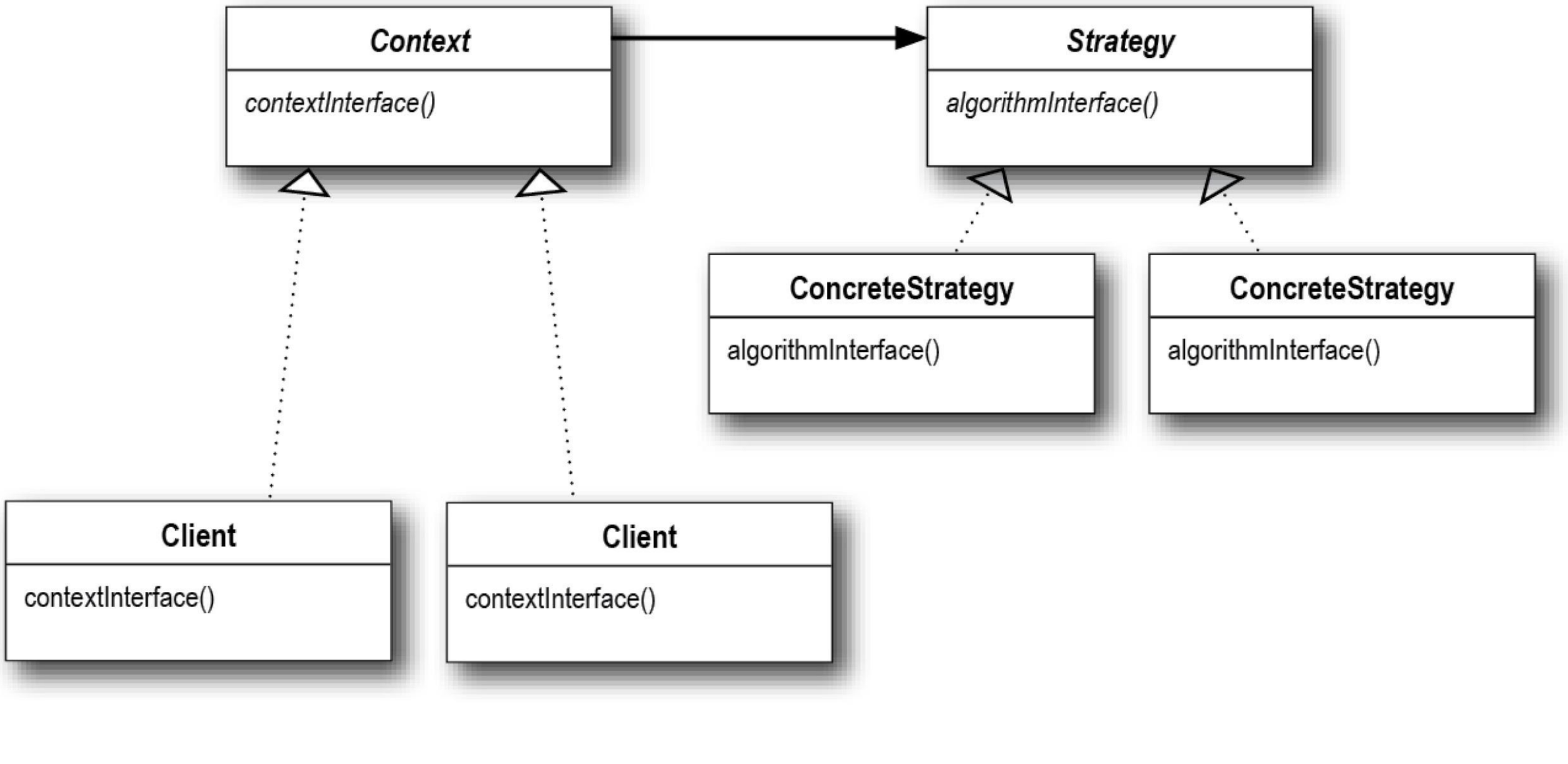
Use the Strategy pattern when:

* many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.

* you need different variants of an algorithm.

Structure

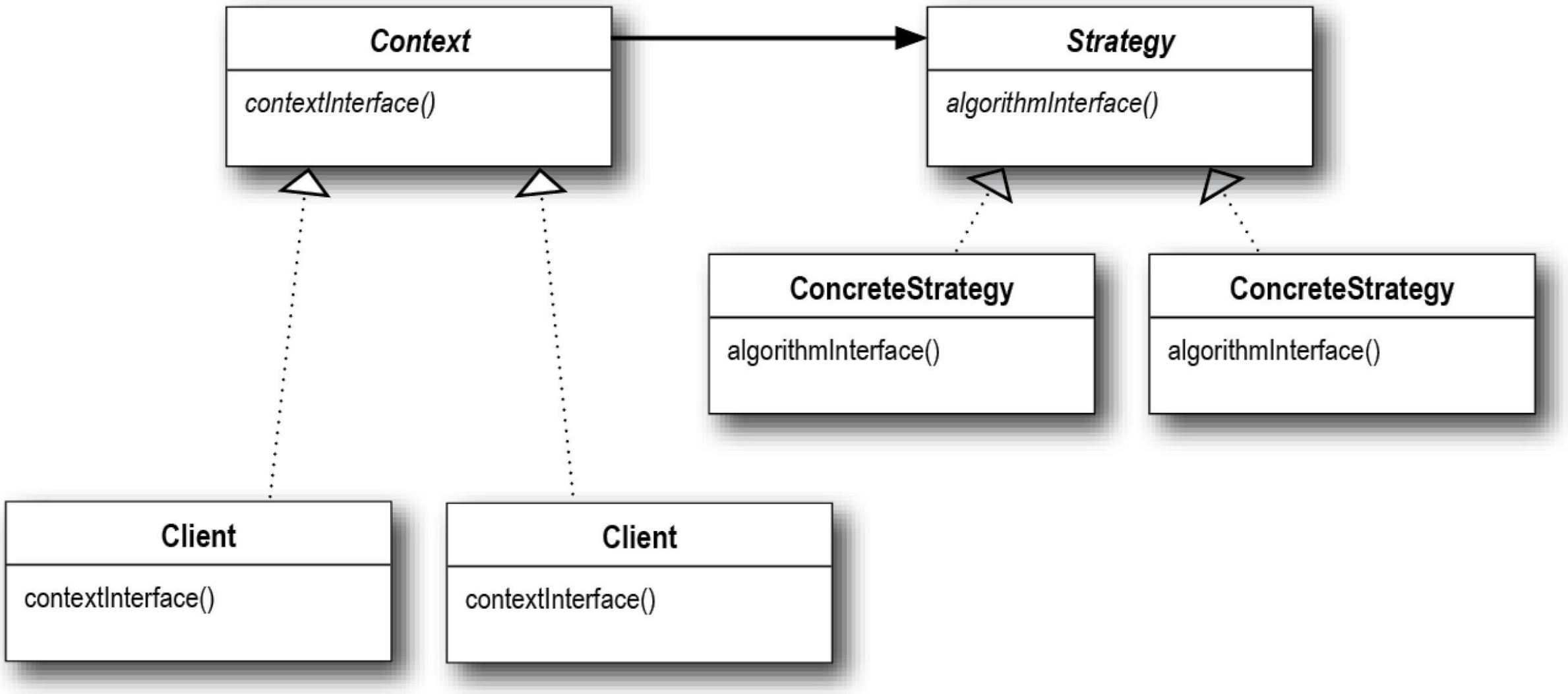
Structure


Context
contextInterface()
Strategy
algorithmInterface()
ConcreteStrategy
algorithmInterface()
ConcreteStrategy
algorithmInterface()
Client
contextInterface()
Client
contextInterface()

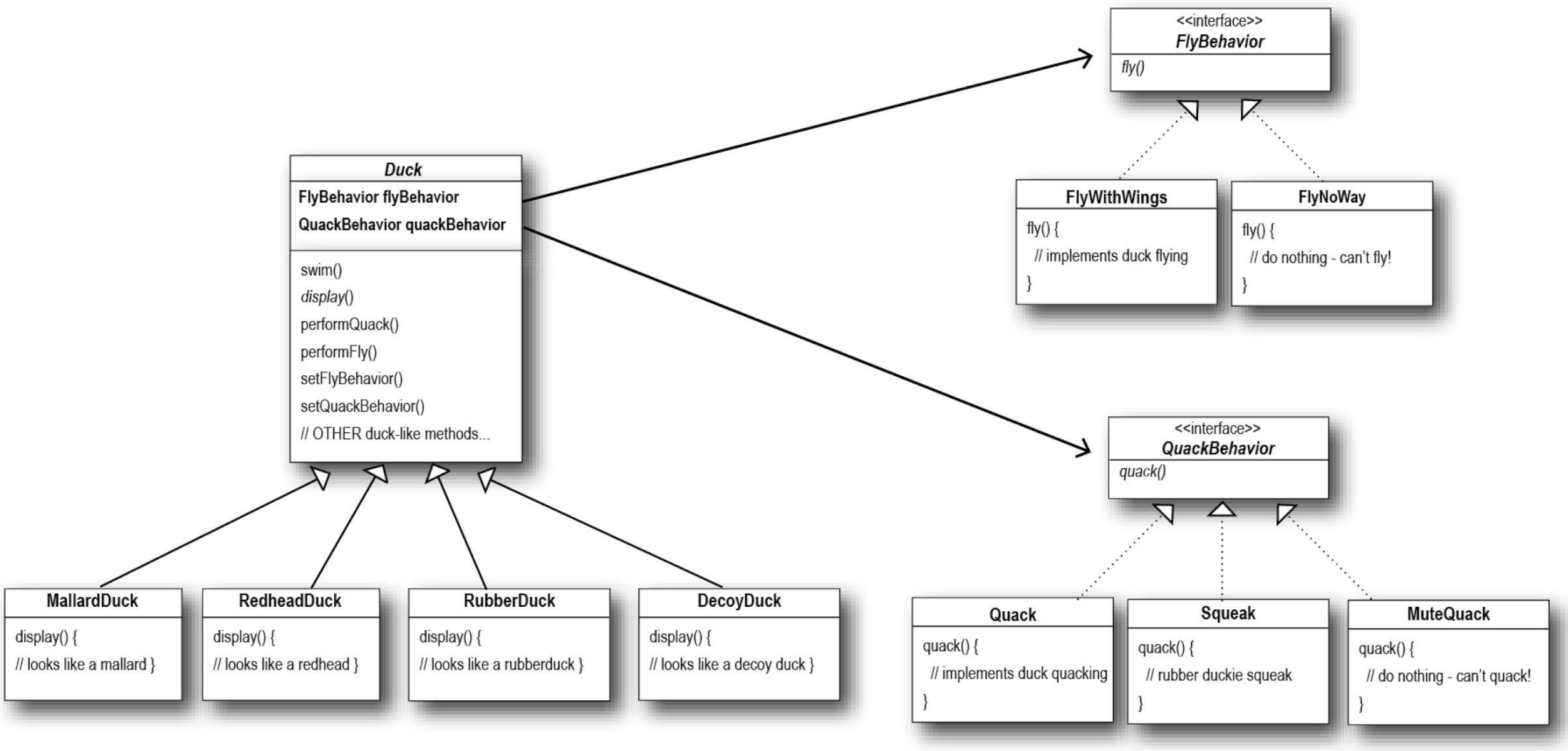
What aren’t Design Patterns?

- Design Patterns aren’t code libraries or modules.
- Design Patterns aren’t coding idioms.
- Design Patterns aren’t Design Principles.

Strategy Pattern Class Diagram



The SimUDuck Class Diagram

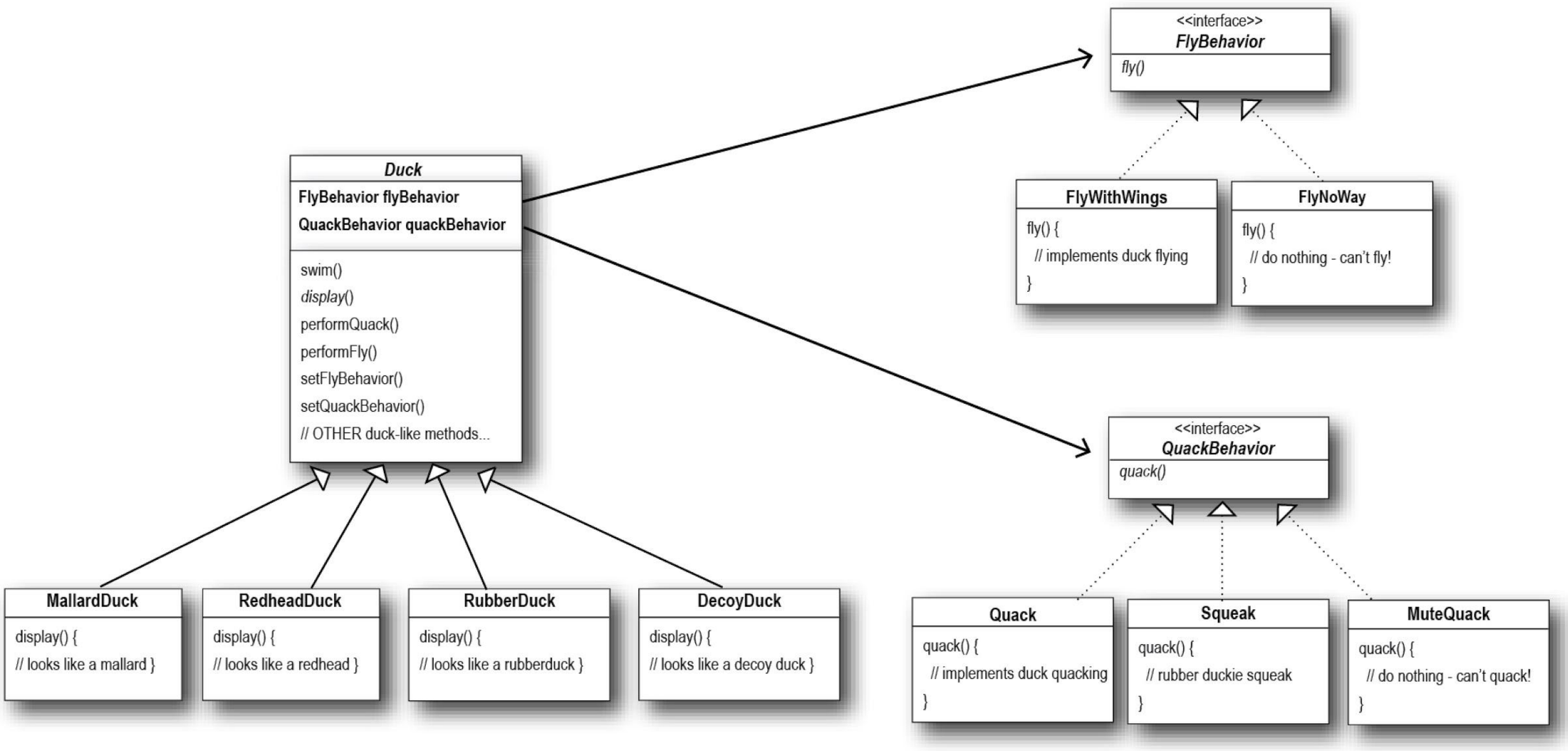


The Strategy Pattern

The Strategy Pattern

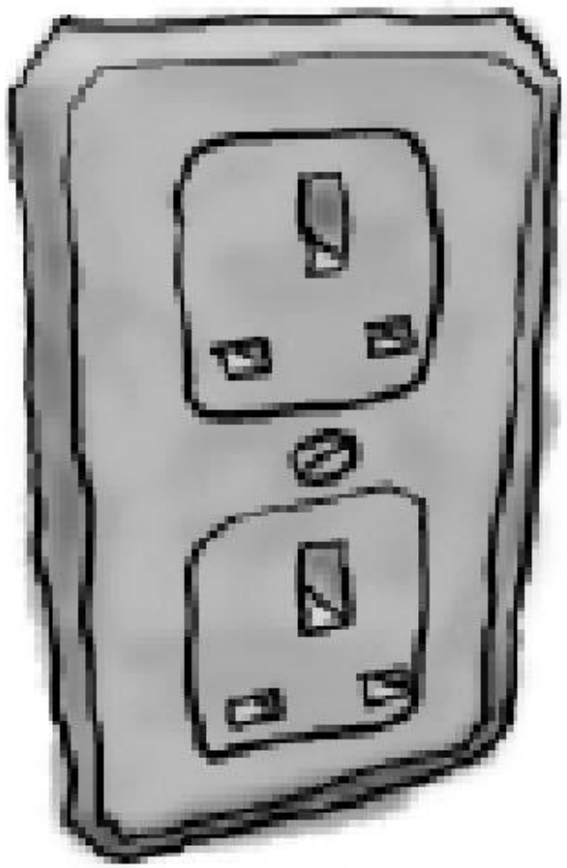
Defines a family of algorithms encapsulating each one, and makes them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

The SimUDuck Class Diagram



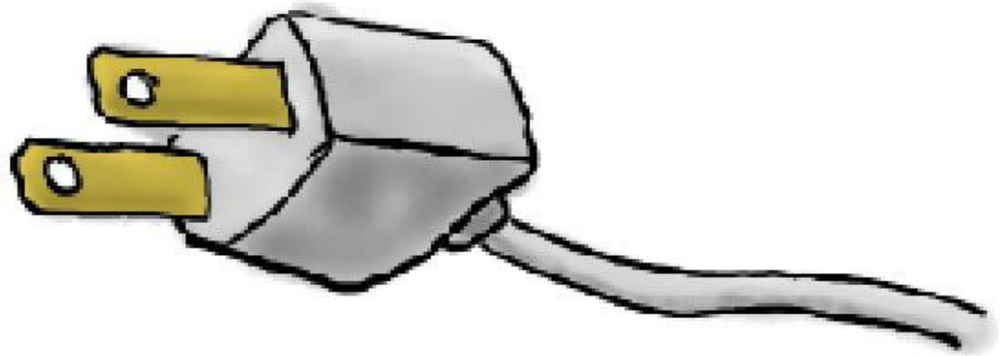
How to be Adaptive

British Wall Outlet

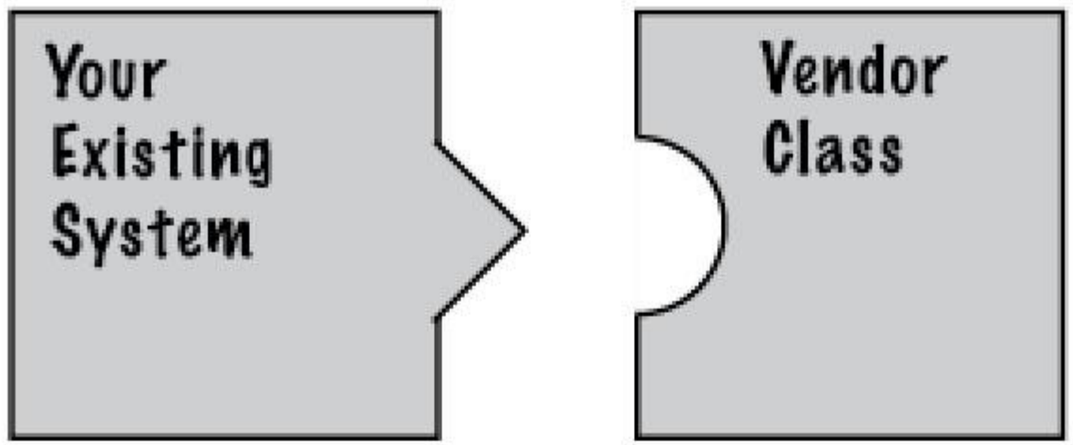


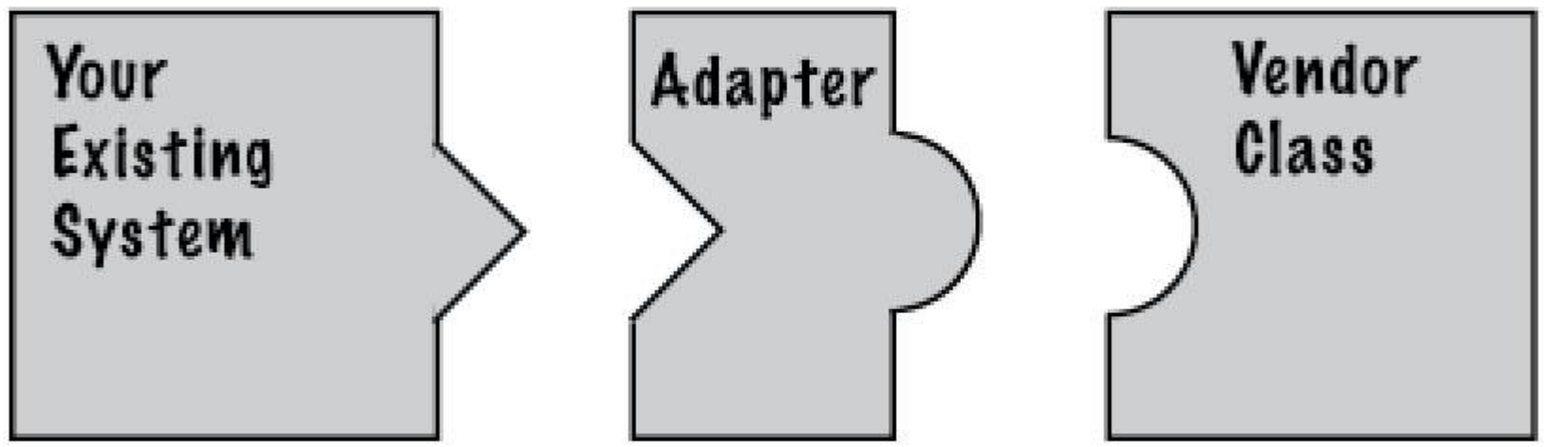
AC Power Adapter

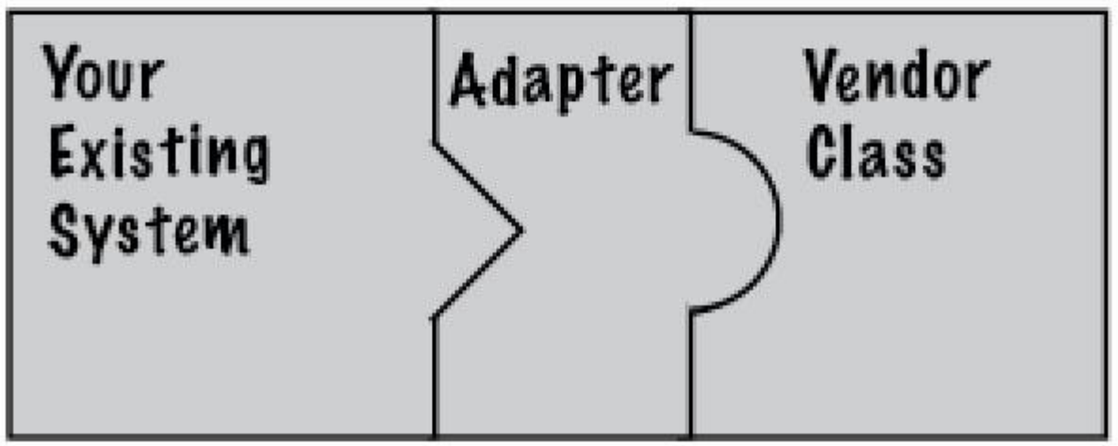
Standard AC Plug



Not like this!


Your
Existing
System
Vendor
Class


Your
Existing
System
Adapter
Vendor
Class


Your
Existing
System
Adapter
Vendor
Class

ADAPTER

Object Structural

Intent

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Motivation

Sometimes one class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires.

How can existing and unrelated classes work in an application that expects classes with a different and incompatible interface?

We could define a class so that it adapts one interface to another interface.

Applicability

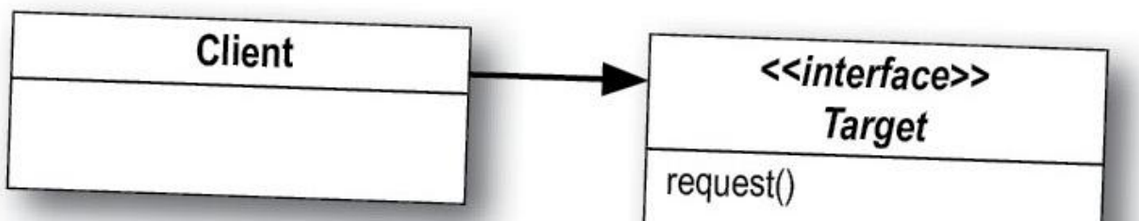
Use the Adapter pattern when:

* you want to use an existing class, and its interface does not match the one you need.

* you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.

* you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Structure



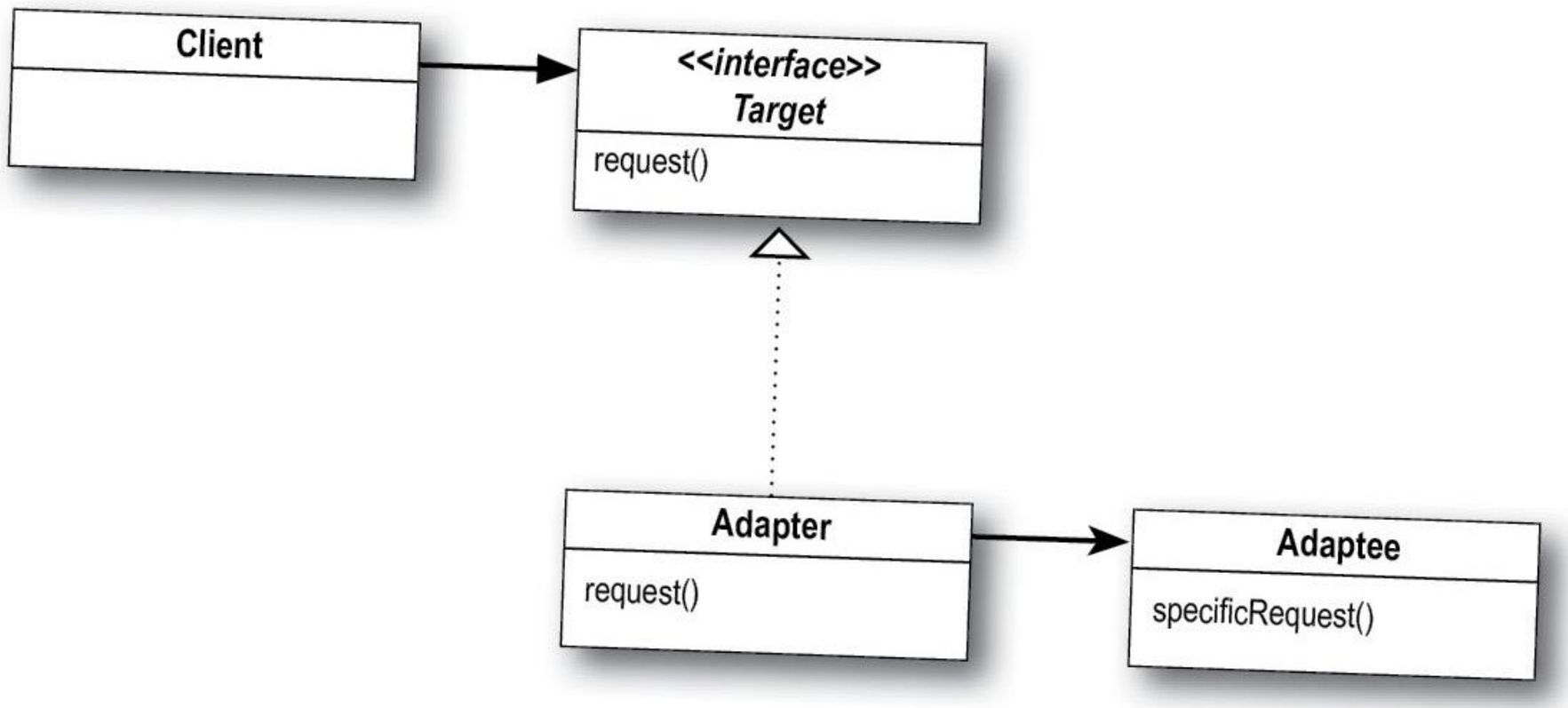
Use the Adapter pattern when:

* you want to use an existing class, and its interface does not match the one you need.

* you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.

* you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Structure



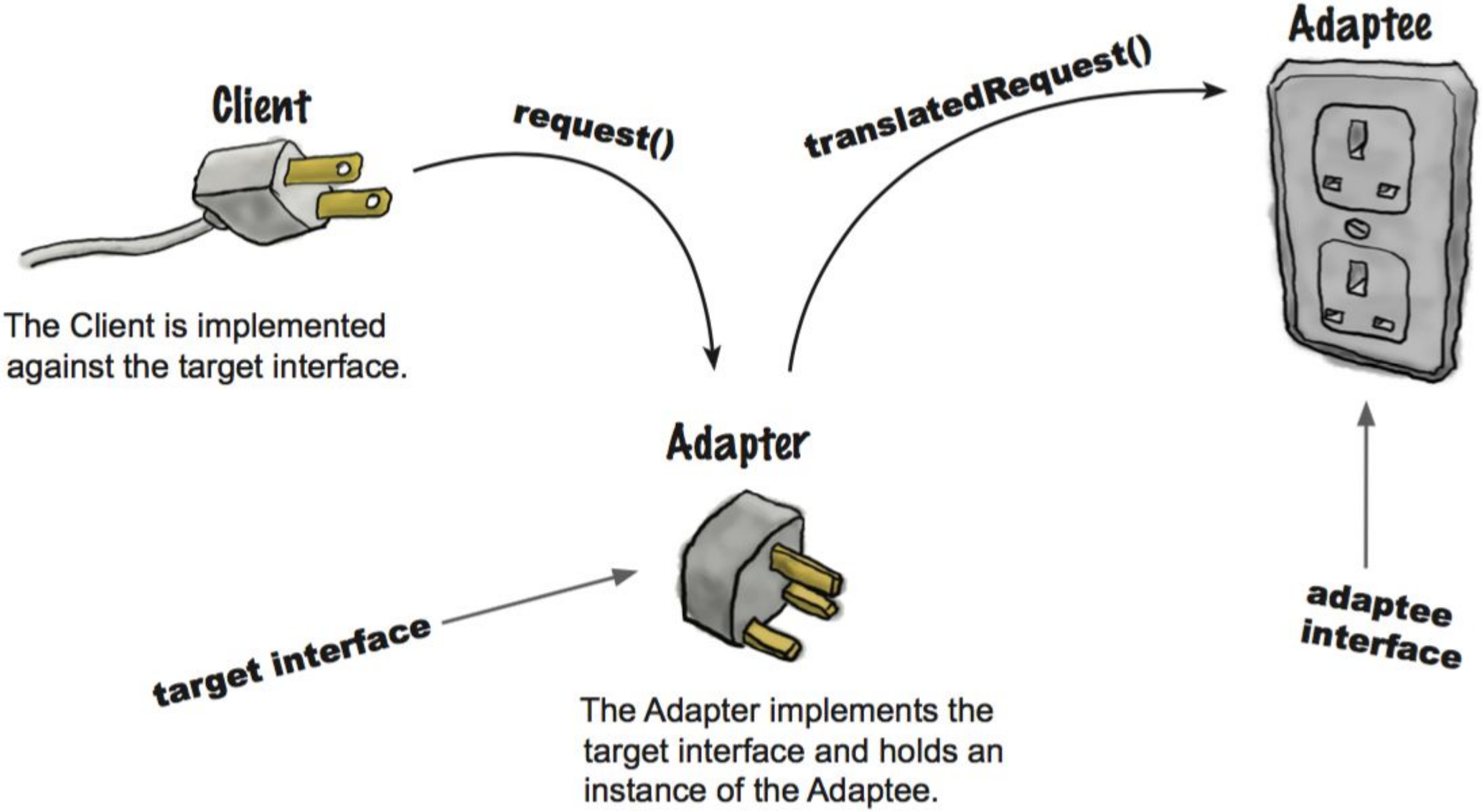
Participants

Target - defines the domain-specific interface that Client uses.

Client - collaborates with objects conforming to the Target interface.

Adaptee - defines an existing interface that needs adapting.

Adapter - adapts the interface of Adaptee to the Target interface.


Client
request()
translatedRequest()
Adaptee
The Client is implemented against the target interface.
Adapter
target interface
adaptee interface
The Adapter implements the target interface and holds an instance of the Adaptee.

Ducks, again

```
public interface Duck {
   public void quack();
   public void fly();
}

public class MallardDuck implements Duck {
   public void quack() {
      System.out.println("Quack");
   }

   public void fly() {
      System.out.println("I'm flying");
   }
}

public interface Turkey {
   public void gobble();
   public void fly();
}
```

Turkey

```
public class WildTurkey implements Turkey {
   public void gobble() {
      System.out.println("Gobble gobble");
   }
   public void fly() {
      System.out.println("I'm flying a short distance");
   }
}
```

Turkey Adapter

```
public class TurkeyAdapter implements Duck {
   Turkey turkey;

   public TurkeyAdapter(Turkey turkey) {
      this.turkey = turkey;
   }

   public void quack() {
      turkey.gobble();
   }

   public void fly() {
      for(int i=0; i < 5; i++) {
         turkey.fly();
      }
   }
}
```

Turkey Adapter

```
public class DuckTestDrive {
   public static void main(String[] args) {
      Duck duck = new MallardDuck();

      Turkey turkey = new WildTurkey();
      Duck turkeyAdapter = new TurkeyAdapter(turkey);

      System.out.println("\nThe Duck says...");
      testDuck(duck);

      System.out.println("\nThe TurkeyAdapter says...");
      testDuck(turkeyAdapter);
   }

   static void testDuck(Duck duck) {
      duck.quack();
      duck.fly();
   }
}
```

The Adapter Pattern

Convert the interface of a class into another interface clients expect.

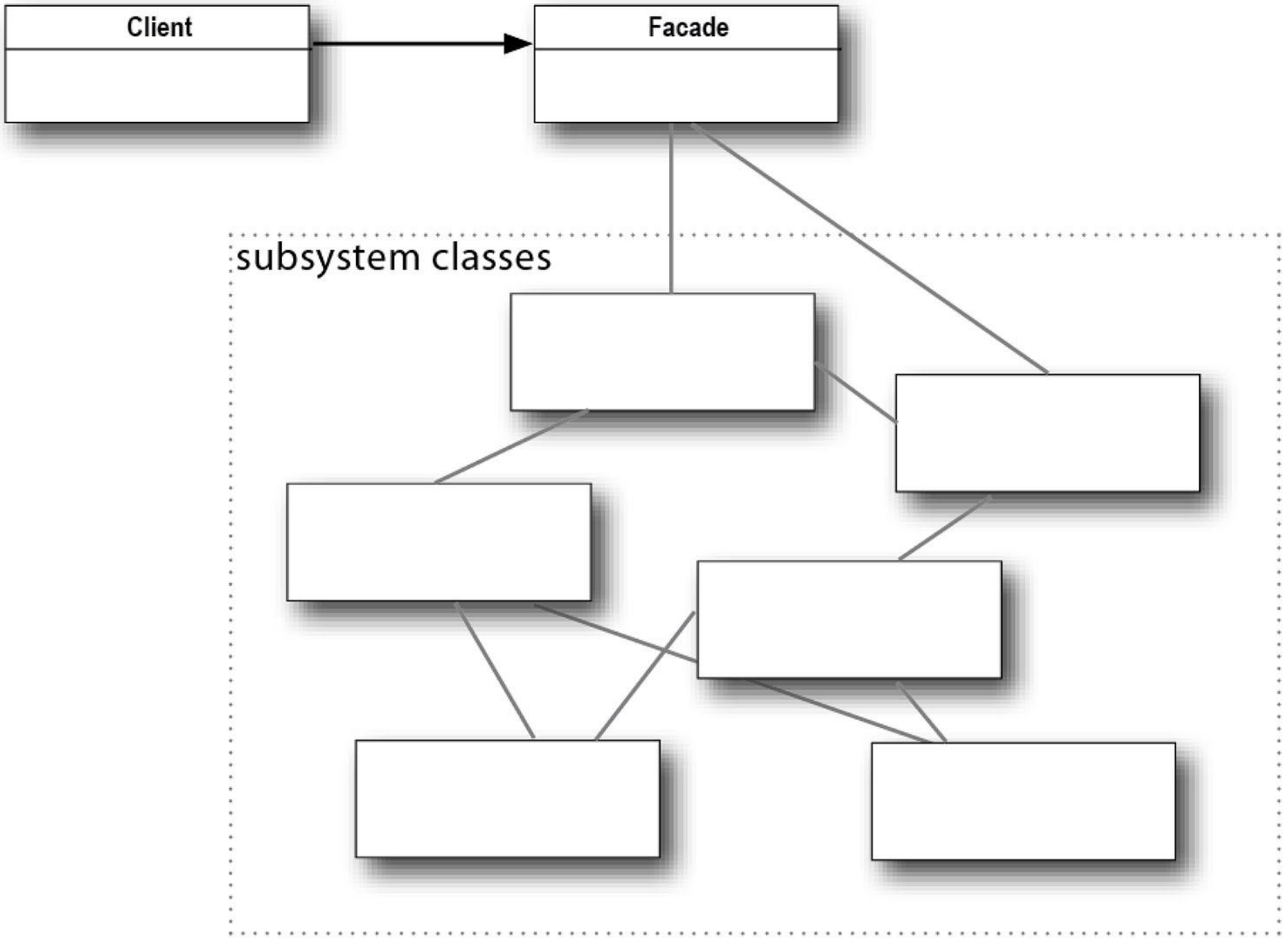
Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

Also check out...

The Facade Pattern

Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Facade's Structure



Design Principles

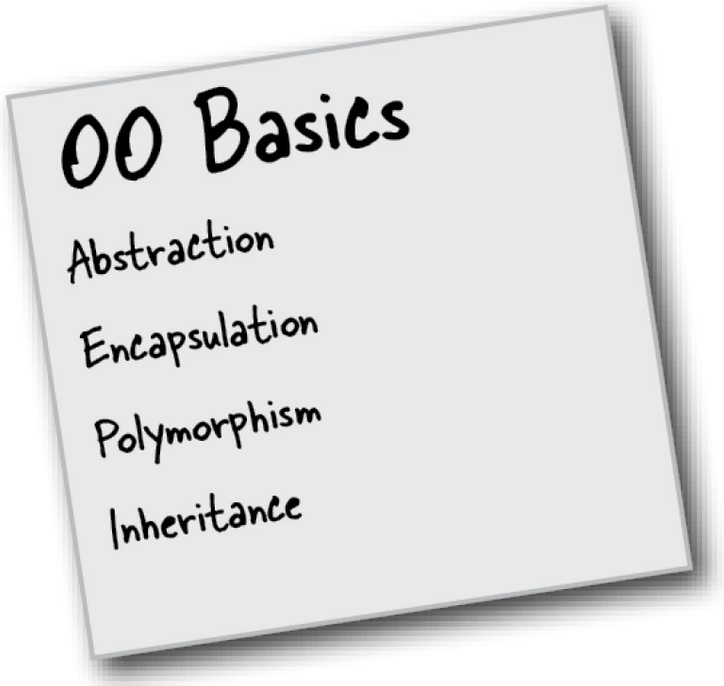
What are Design Principles?

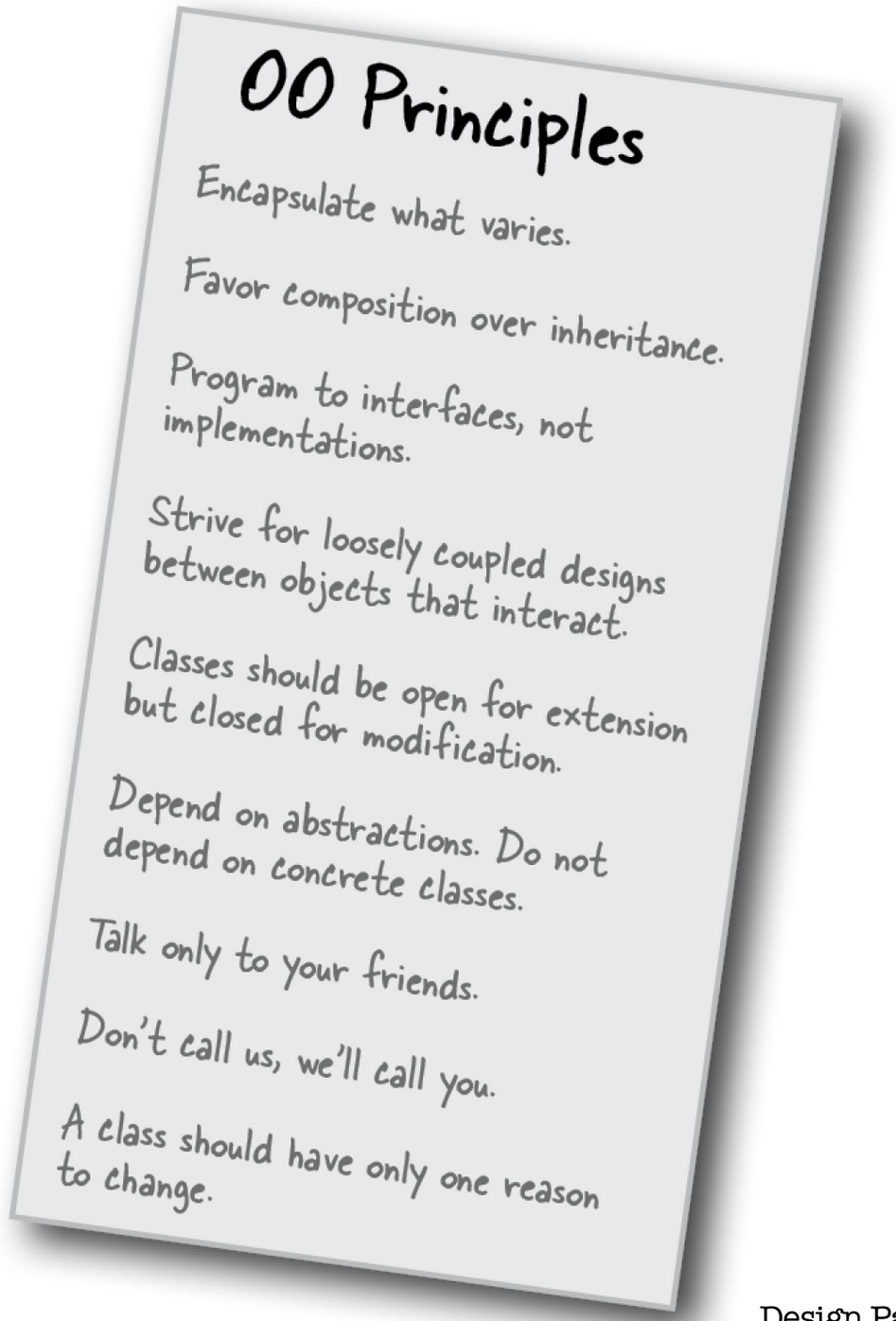
- A design principle is a general guideline that helps us avoid bad OOP designs.

- Typically, design principles address some aspect of keeping your designs flexible, resilient, and/or reuseable.

- There are many principles, some deemed more important than others by various OOP practitioners.

Getting Started with Principles



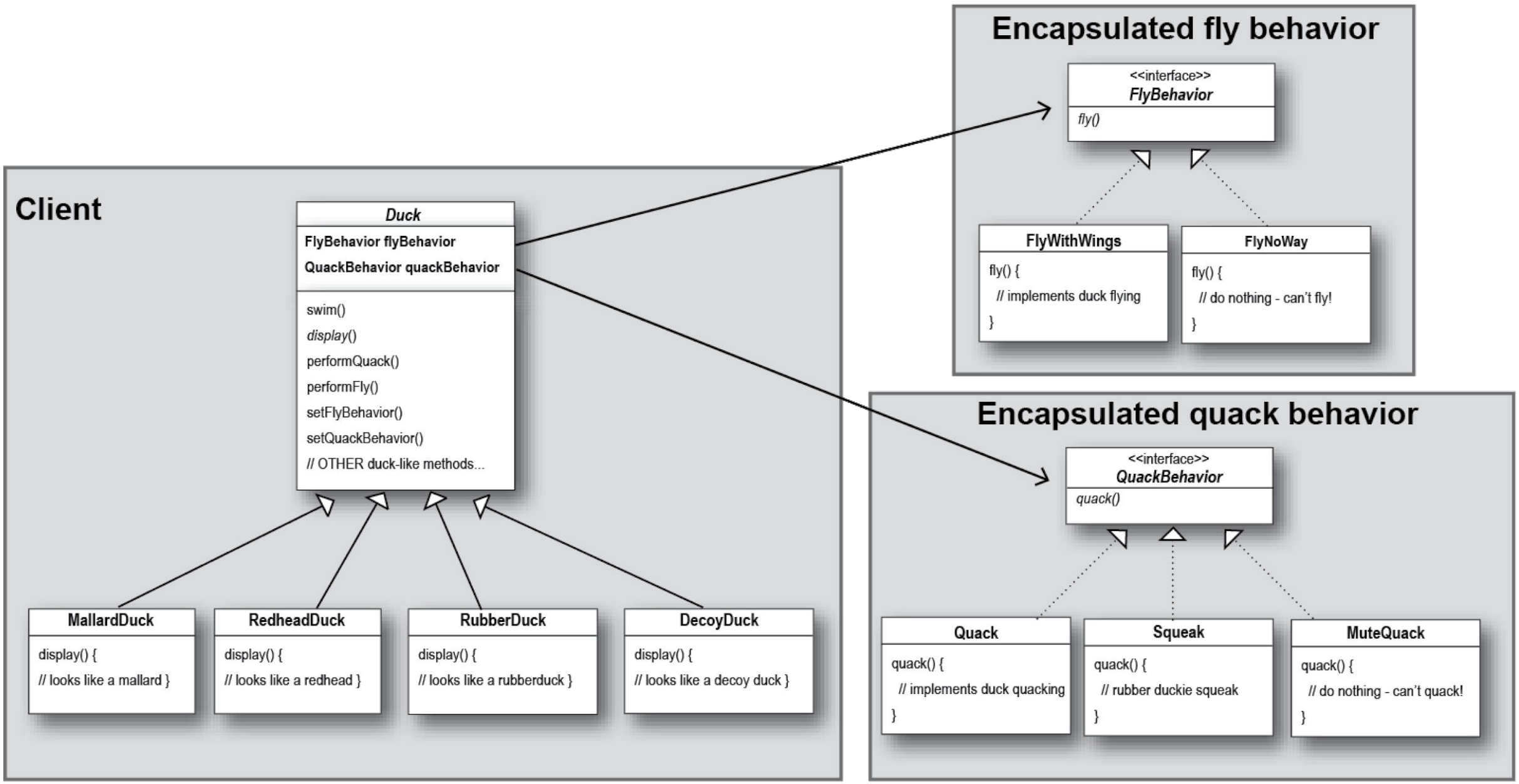


We've already seen two...

Design Principle

Encapsulate what varies

SimUDuck

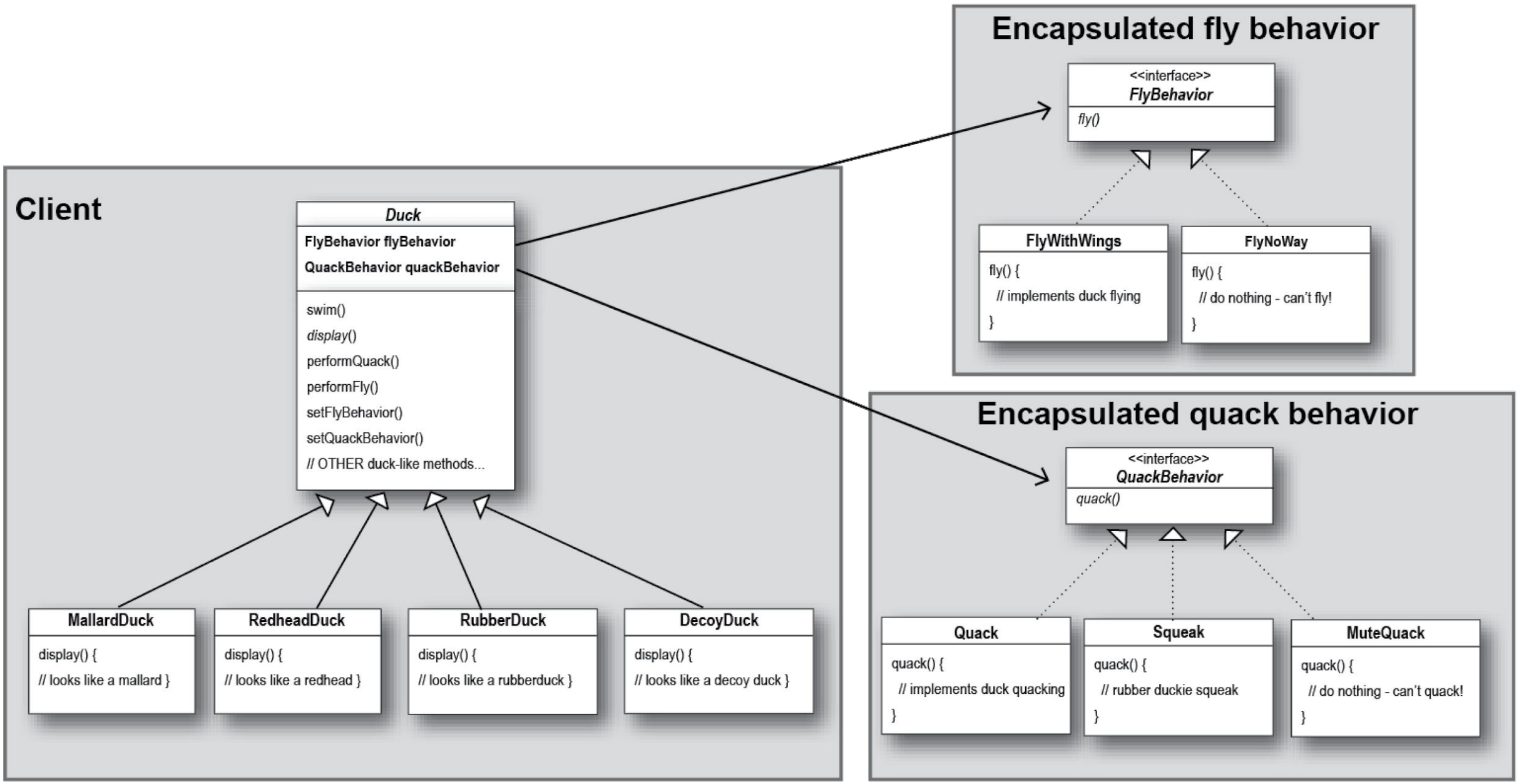

Encapsulated fly behavior
<<interface>>
FlyBehavior
fly()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Client
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
RubberDuck
display() {
// looks like a rubberduck }
DecoyDuck
display() {
// looks like a decoy duck }
Encapsulated quack behavior
<<interface>>
QuackBehavior
quack()
Quack
quack() {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}

HAS-A is better than IS-A
Or...

Design Principle

Favor composition over inheritance.

Take another look at the Strategy Pattern, does it support favoring composition over inheritance well?

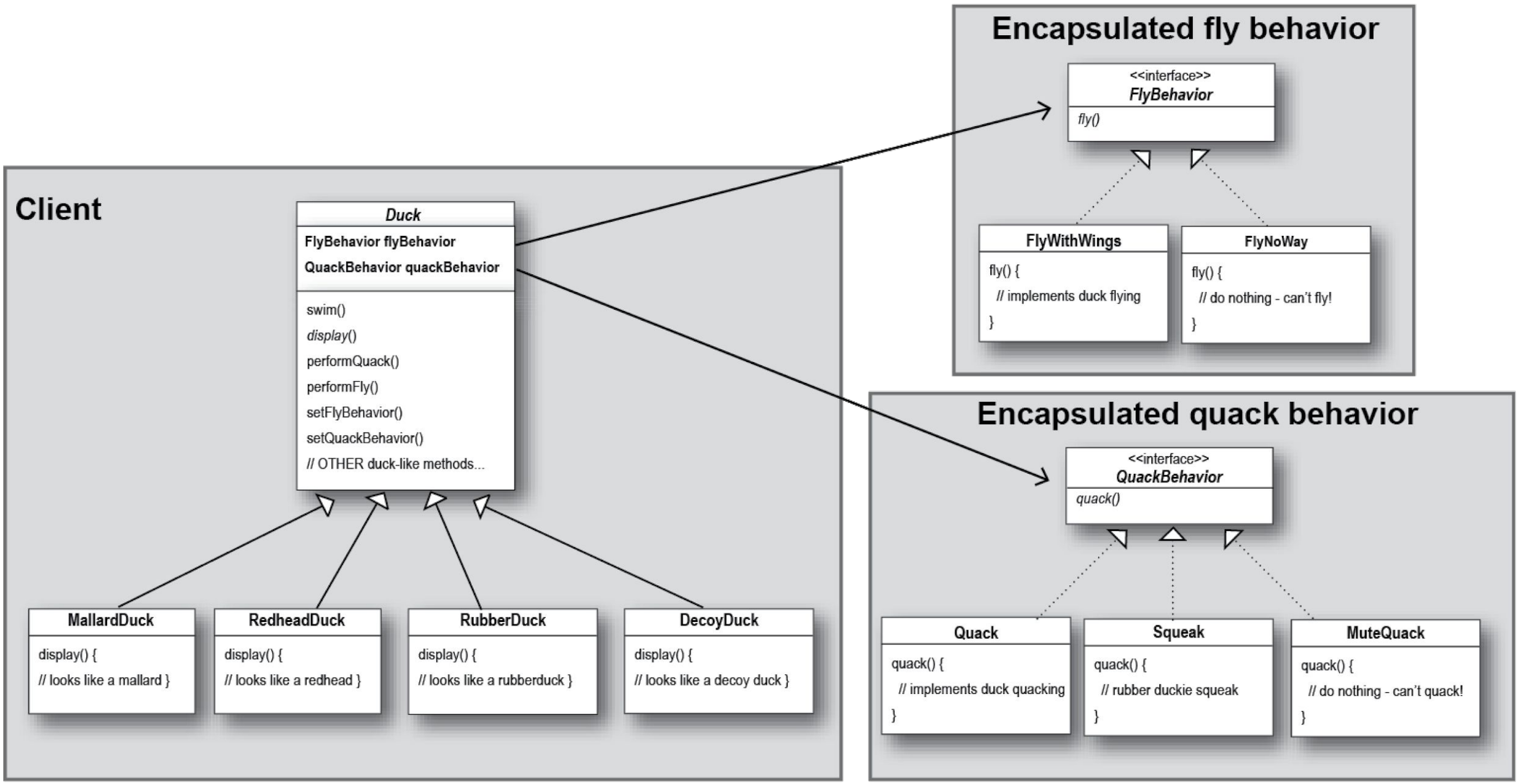


Program to interfaces

Design Principle

Program to an interface, not an implementation.

Take another look at the Strategy Pattern, does it support programming to interfaces well?



Open/Closed Principle

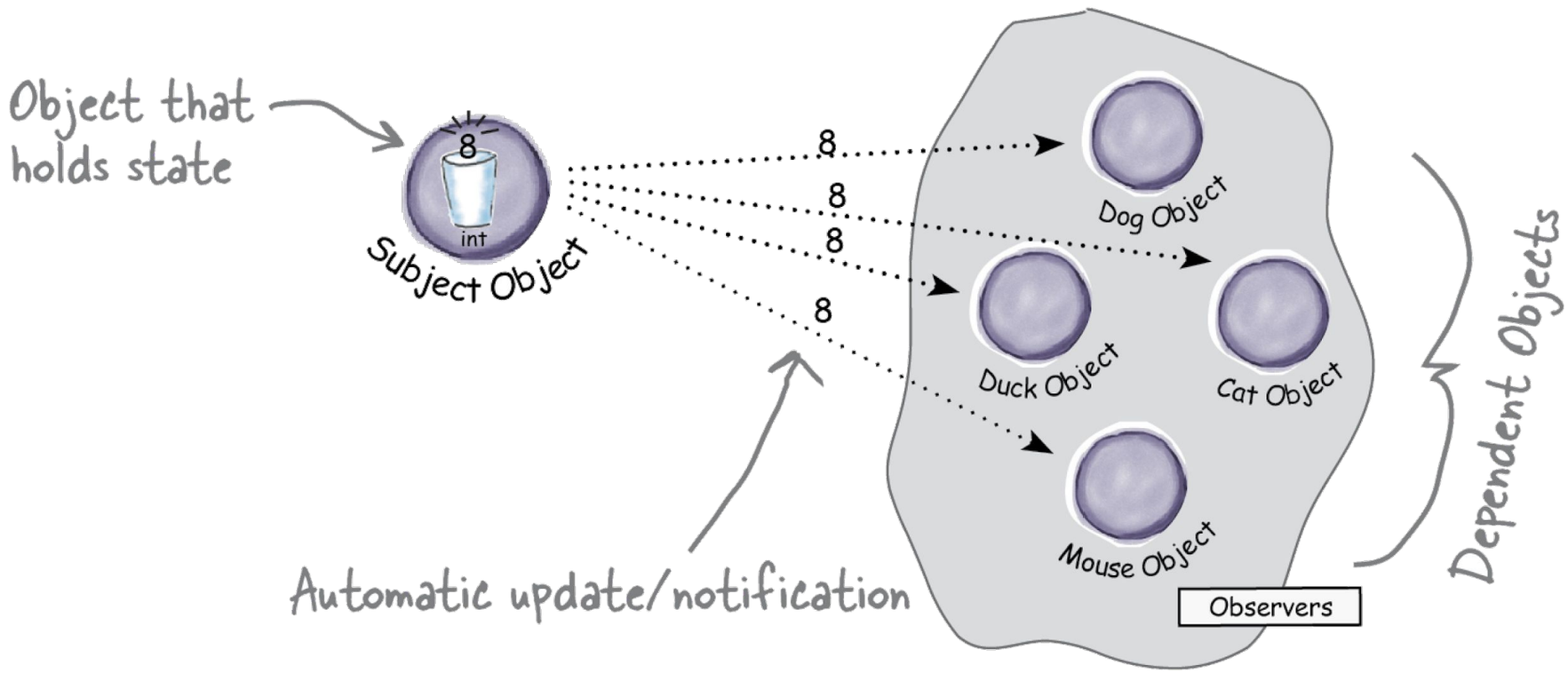
Design Principle

Classes should be open for extension but closed for modification.

Loose Coupling

Design Principle

Strive for loosely coupled designs between objects that interact.



Single Responsibility Principle

Design Principle

A class should have only one reason to change.

SOLID Principles

Single Responsibility Principle

Open/Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

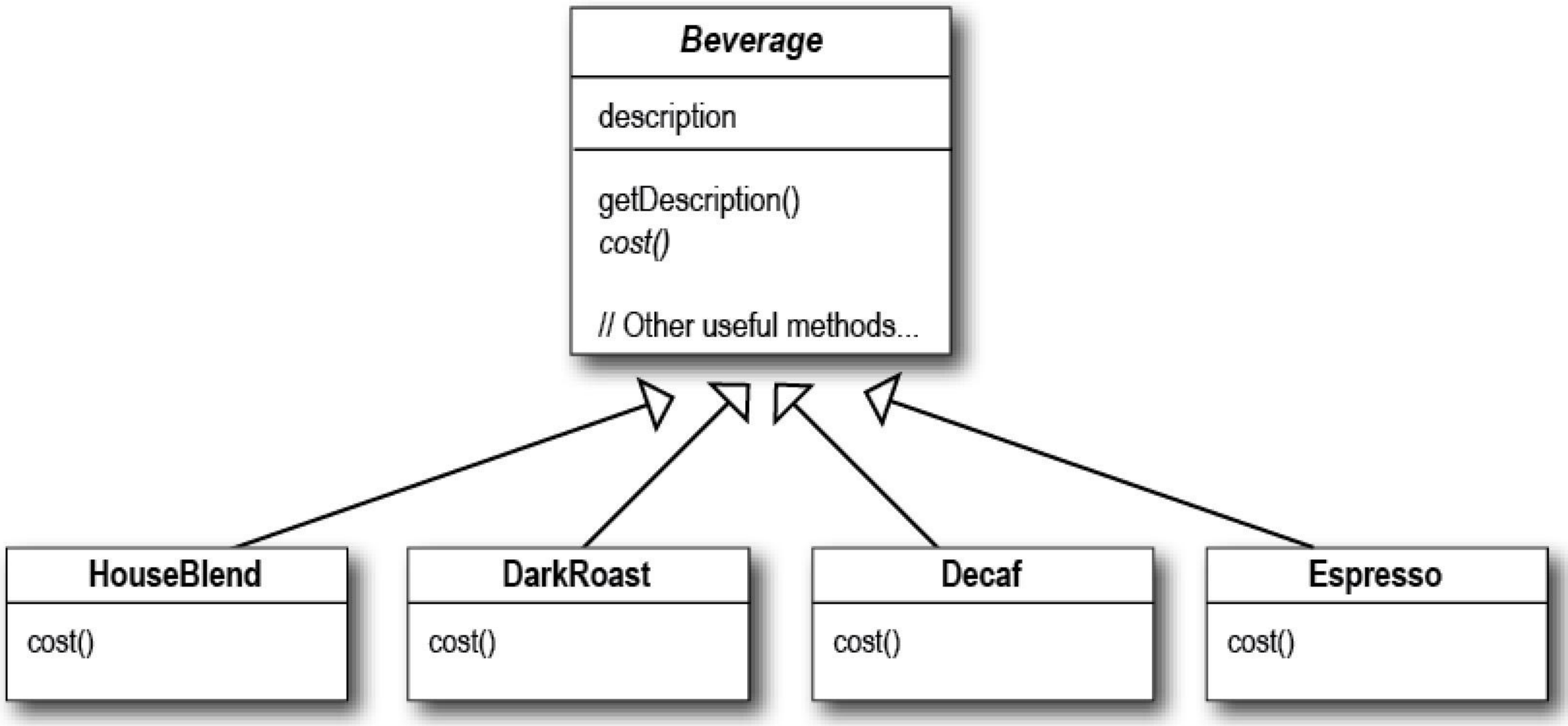
Design Patterns BOOTCAMP

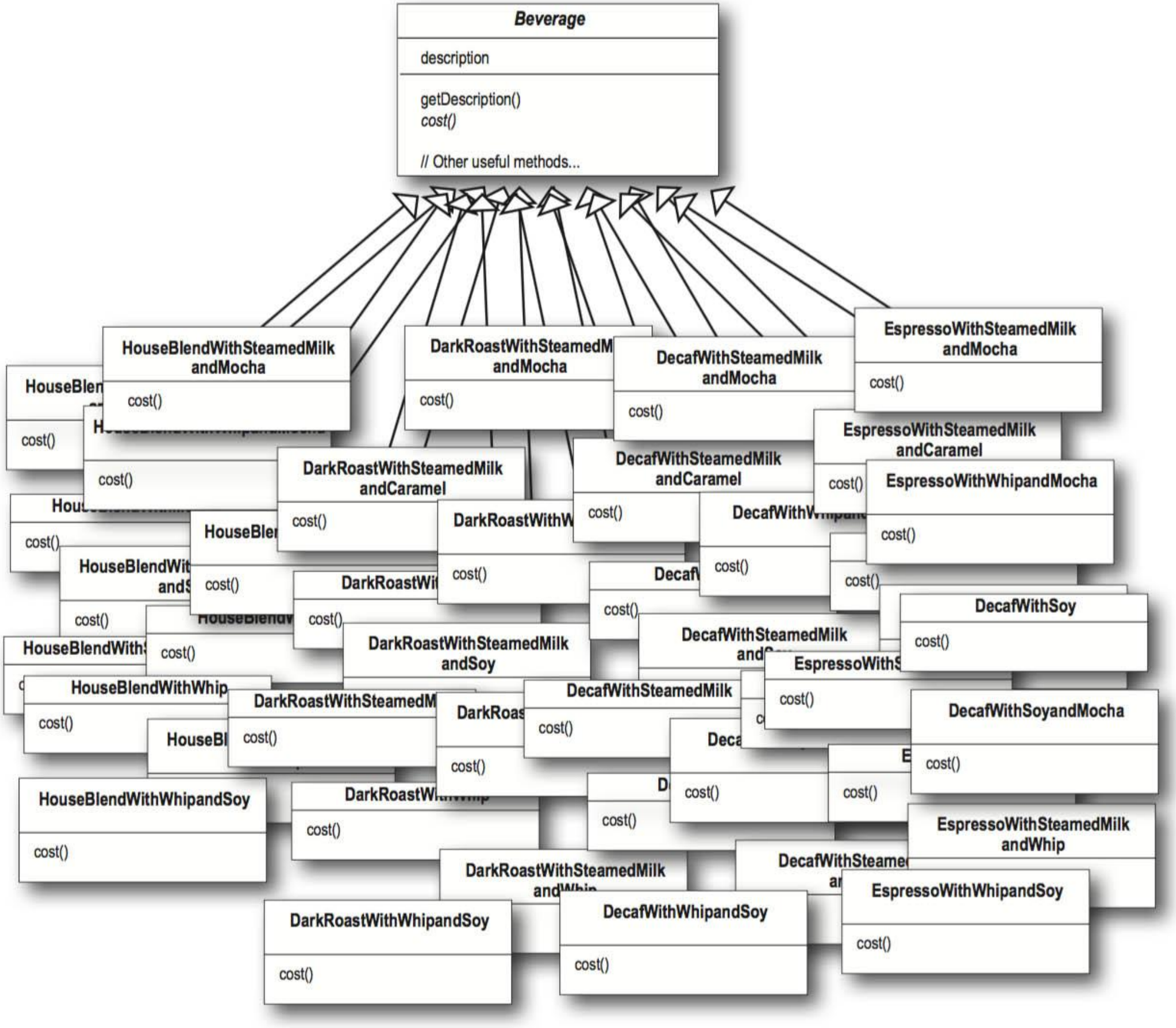
with Eric Freeman & Elisabeth Robson

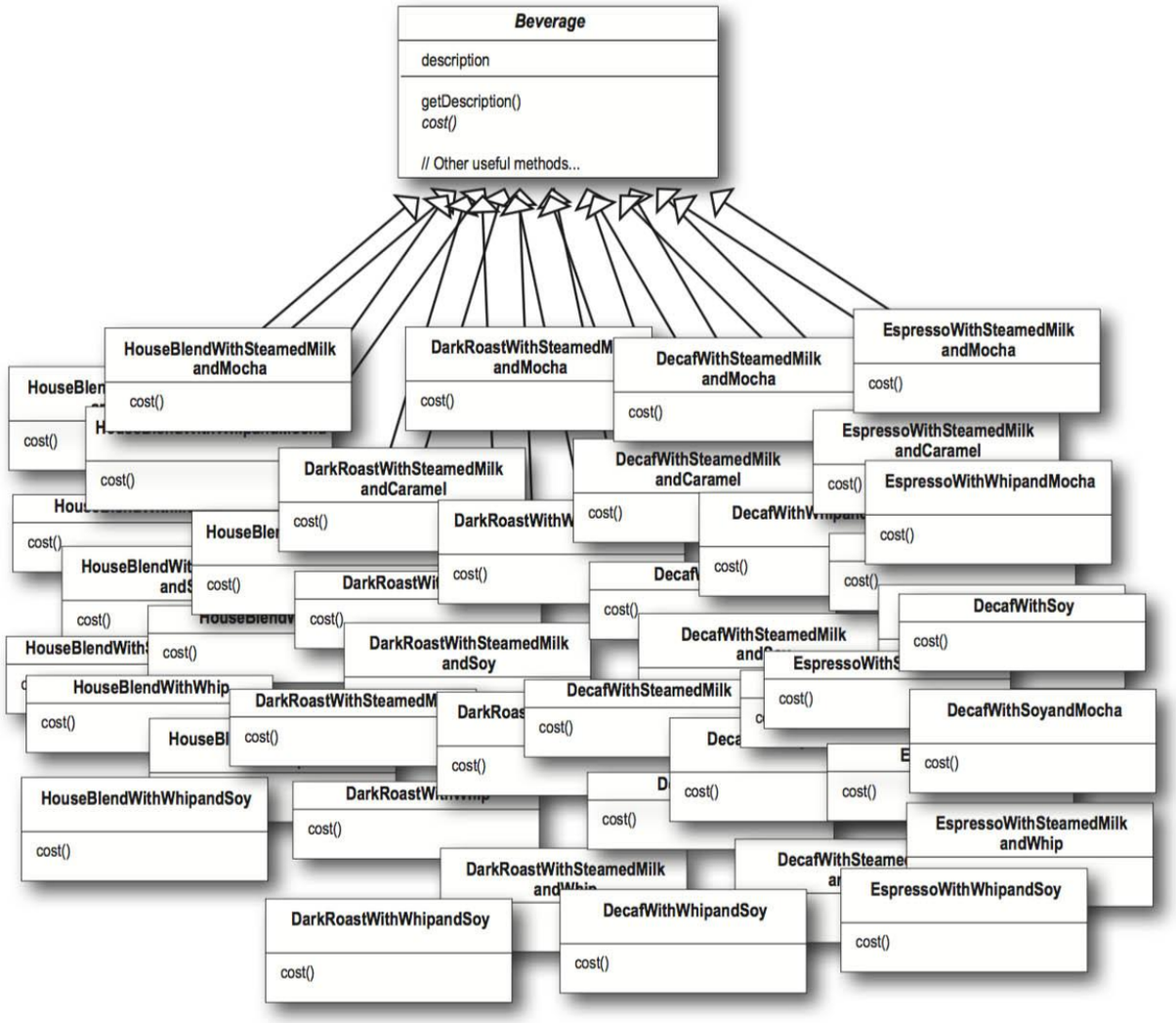
DAY 2

Decorating Objects

Starbuzz Coffee




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()
DecafWithSoyandMocha
cost()
HouseBlendWithWhipandSoy
cost()
DarkRoastWithSteamedMilk andSoy
DecafWithSteamedMilk
EspressoWithSteamedMilk andWhip
DarkRoastWithWhipandSoy
cost()
DecafWithWhipandSoy
cost()
EspressoWithWhipandSoy
cost()



OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

Classes should be open for extension but closed for modification.

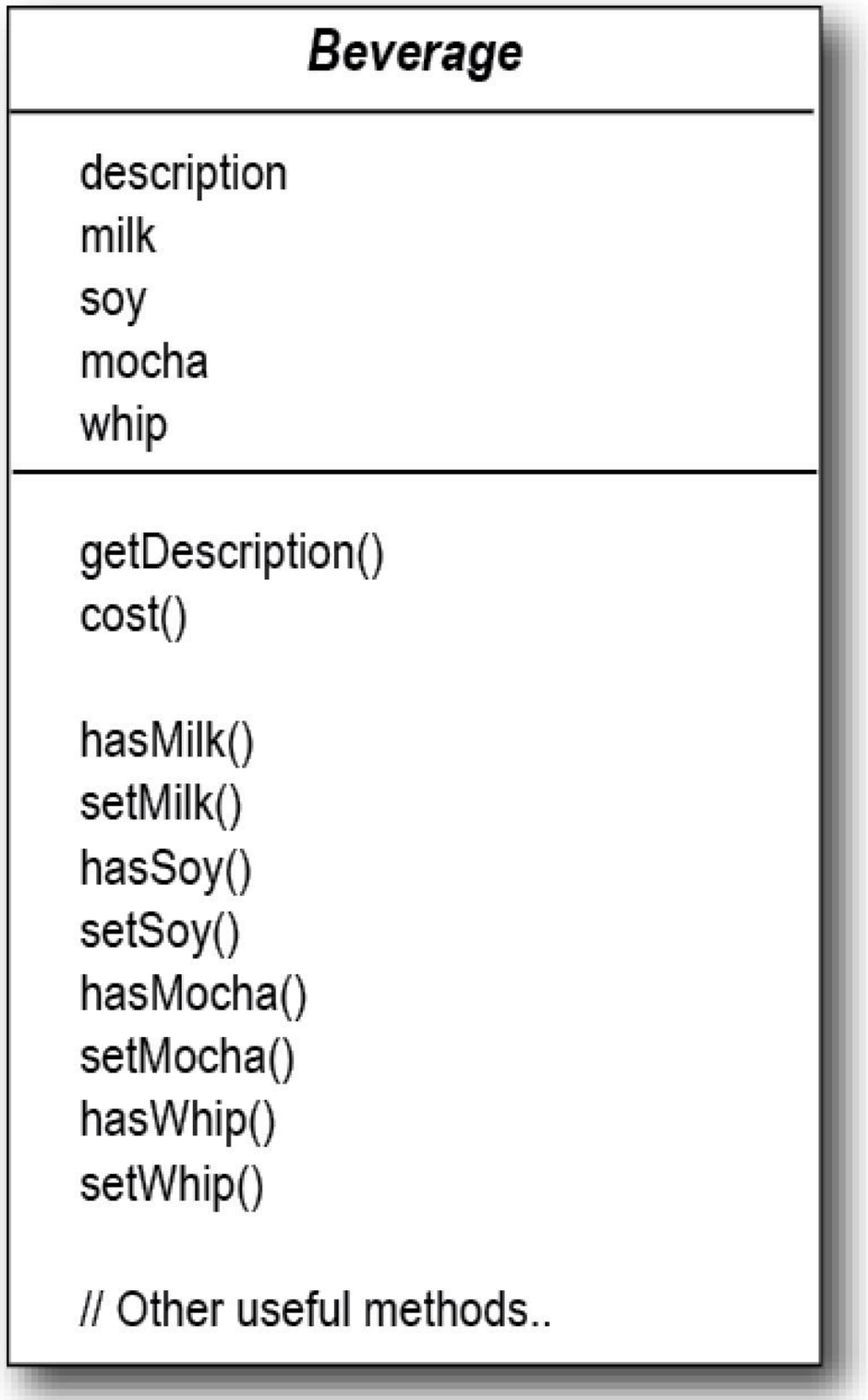
Depend on abstractions. Do not depend on concrete classes.

Talk only to your friends.

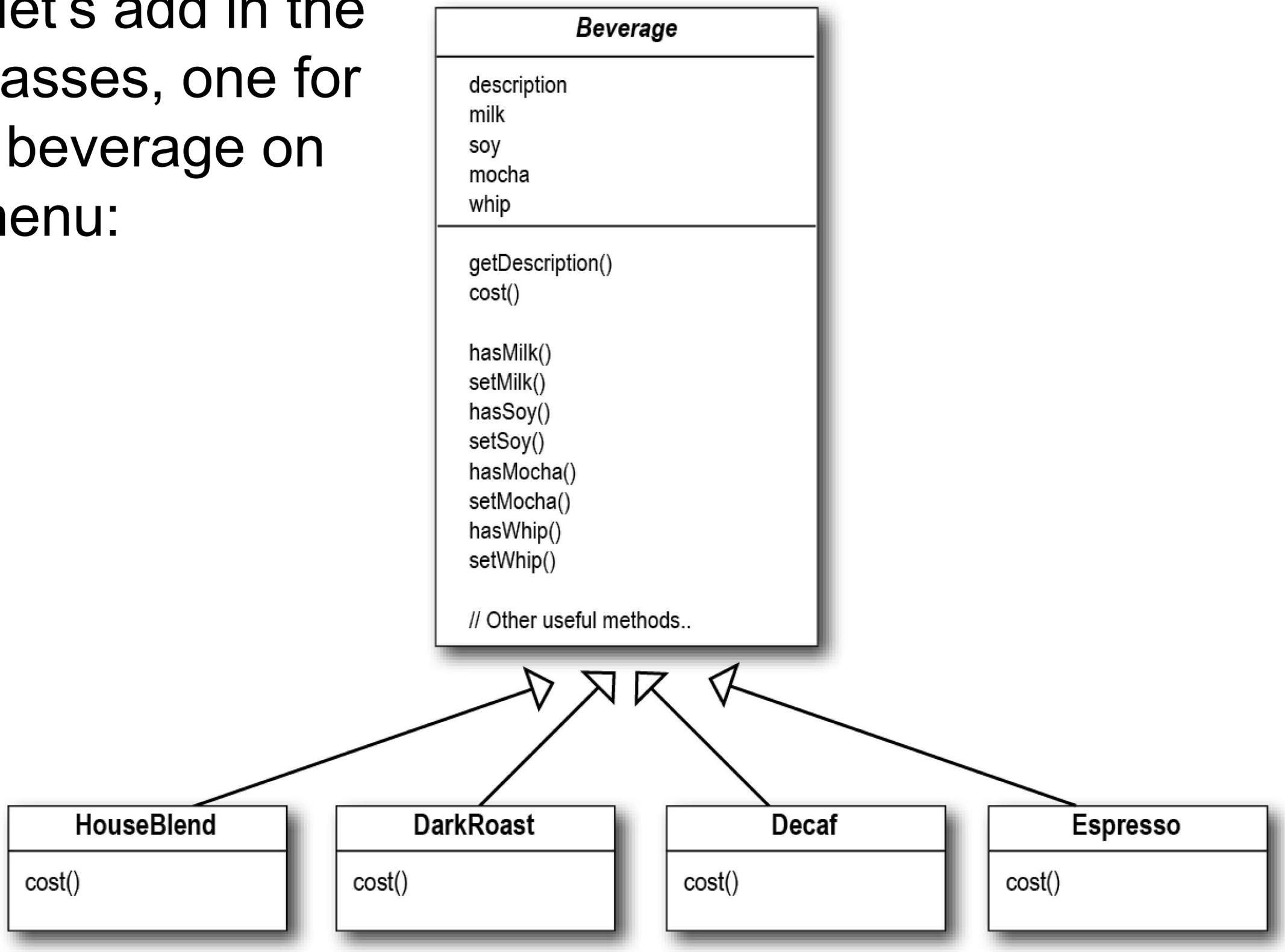
Don't call us, we'll call you.

A class should have only one reason to change.

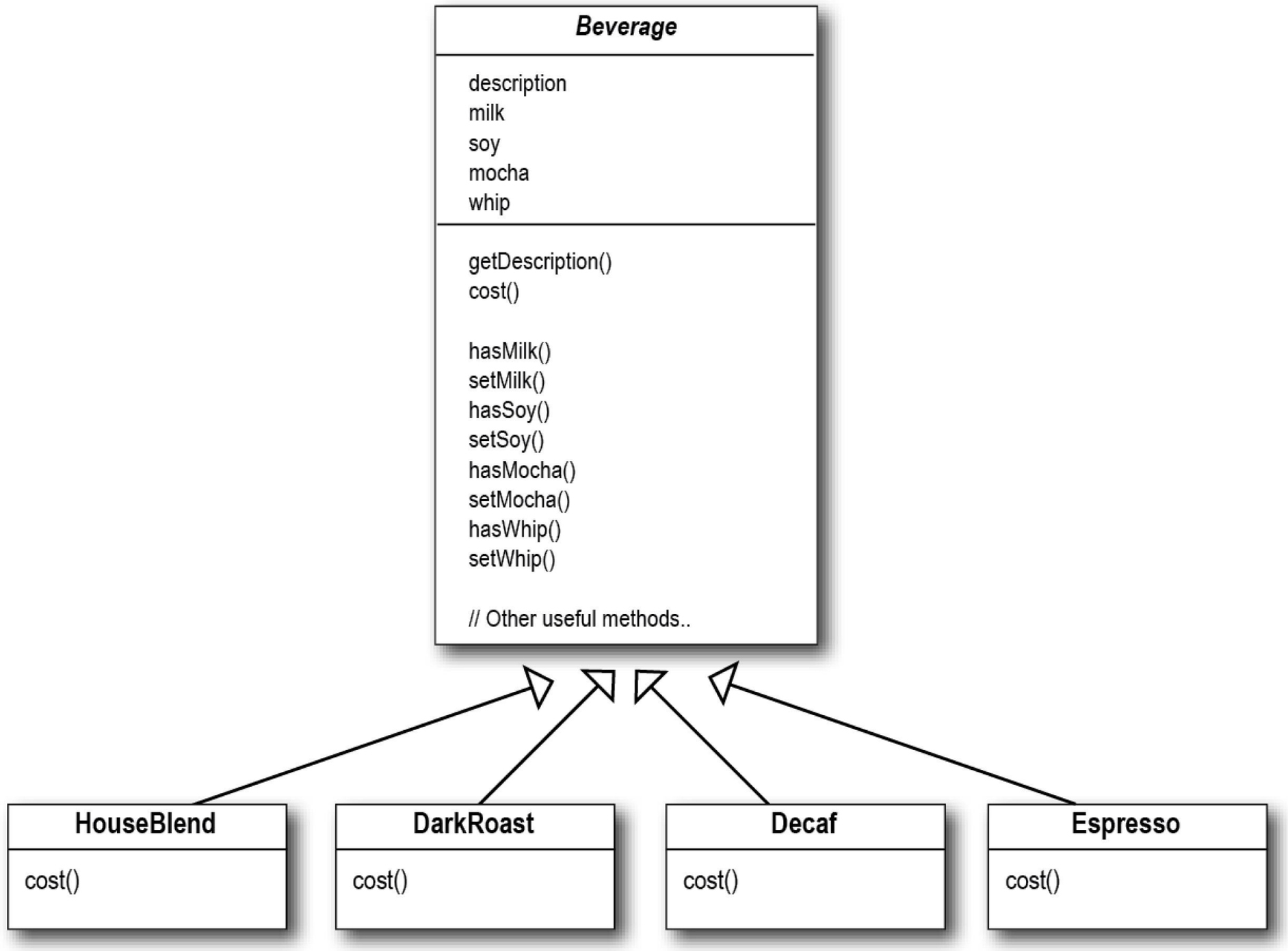
What about this approach?



Now let's add in the subclasses, one for each beverage on the menu:



Only five classes, not bad. Although, what problems might you encounter with this solution?



Only five classes, not bad. Although, what problems might you encounter with this solution?

A. Price changes will force us to alter existing code.

B. New condiments will require new methods and alter the cost method in the superclass.

C. New beverages like iced tea will be forced to inherit methods that don’t make sense.

D. What if a customer wants a double mocha?

DECORATOR

Object Structural

Intent

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

Motivation

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.

One way to add responsibilities is with inheritance. This is inflexible, however, because the client can't control how and when to add properites like a border to a component.

A more flexible approach is to enclode the component in a decorator object that adds the border property.

Applicability

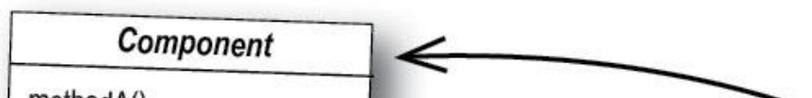
Use Decorator:

* to add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.

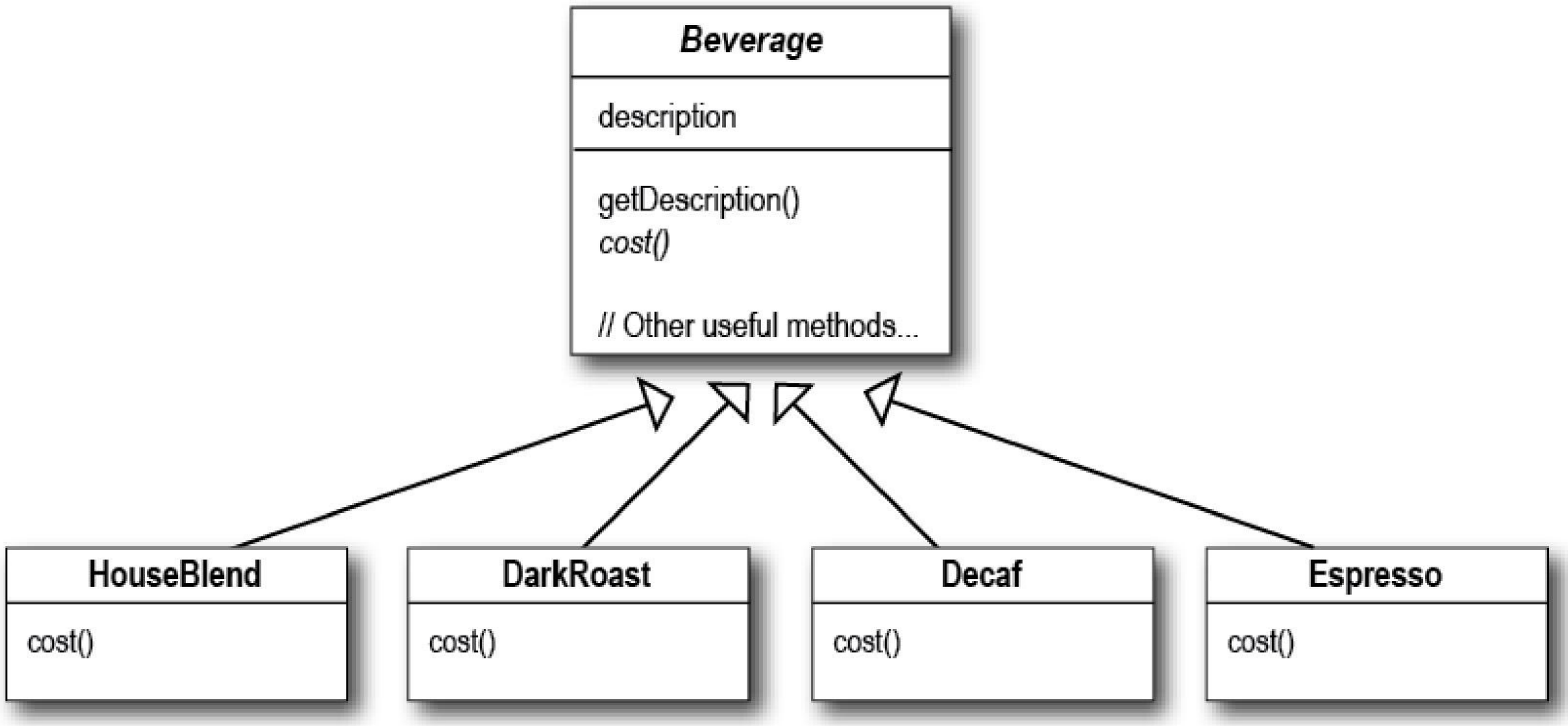
* for responsibilities that can be withdrawn.

* when extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for subclassing.

Structure

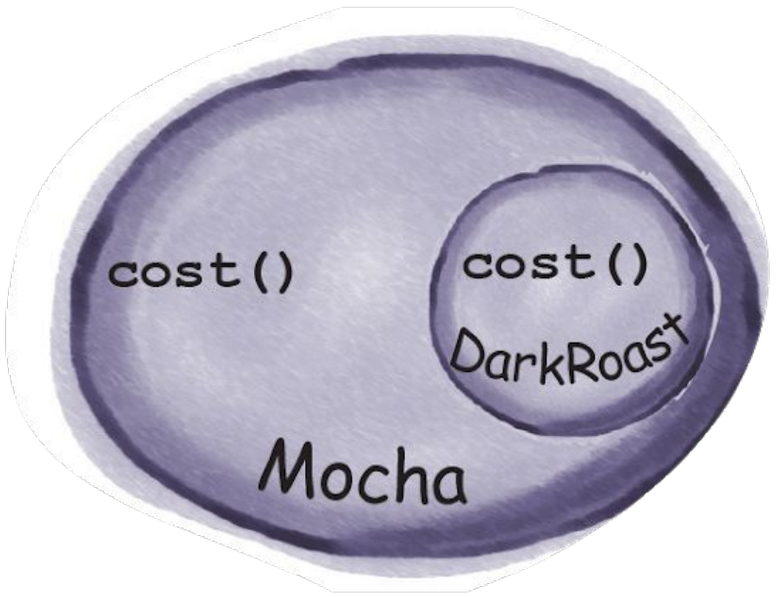


Starbuzz Coffee

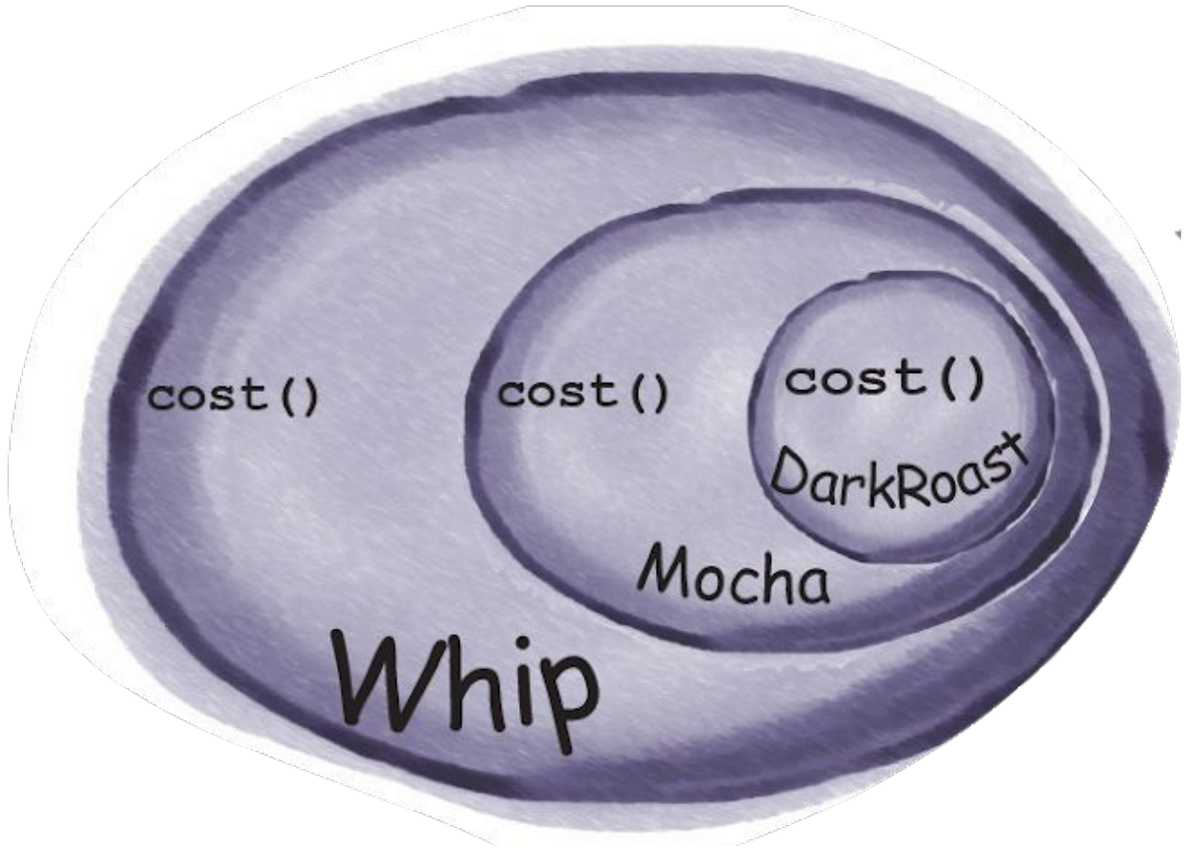


1 **We start with our DarkRoast object.**

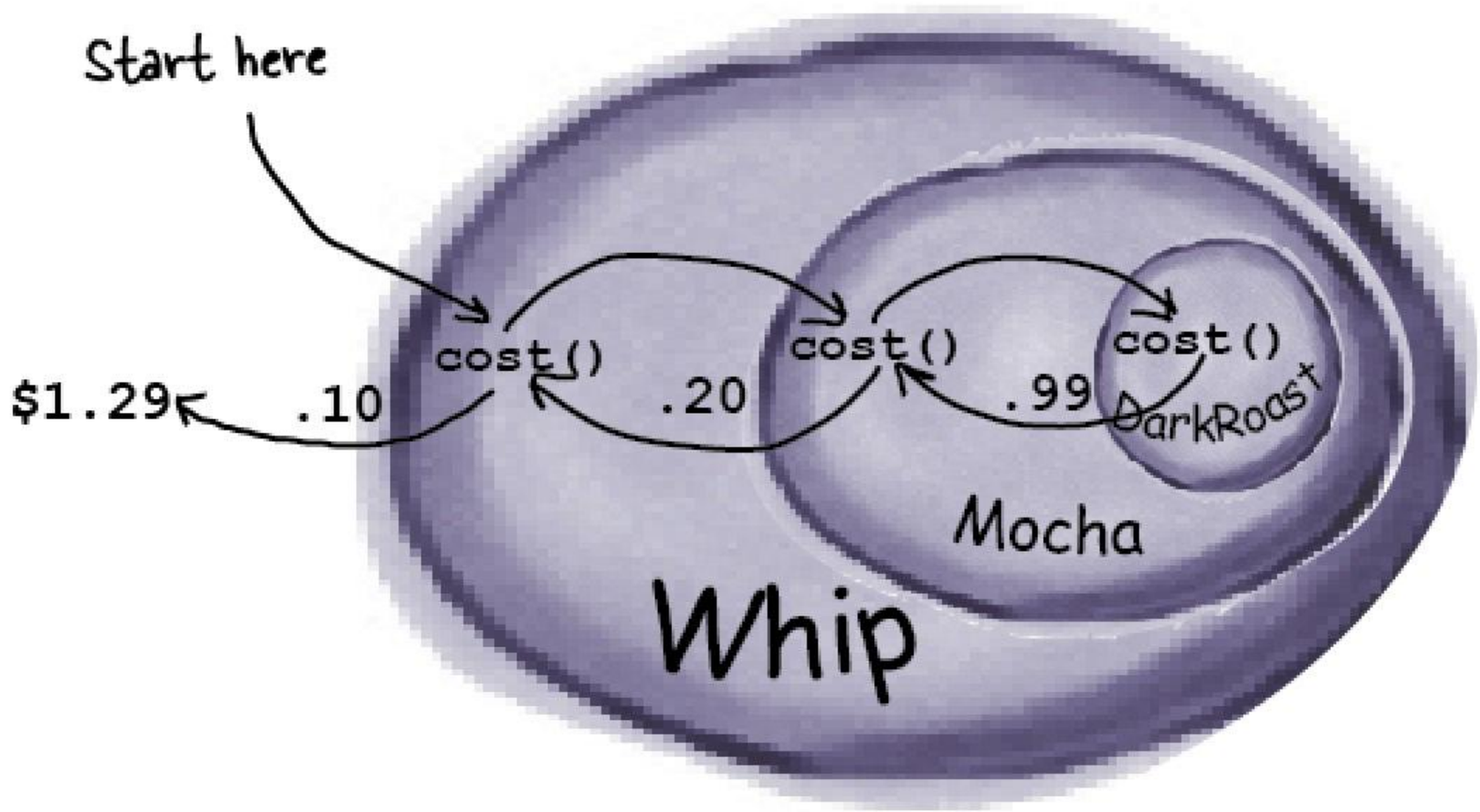
2 **The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.**



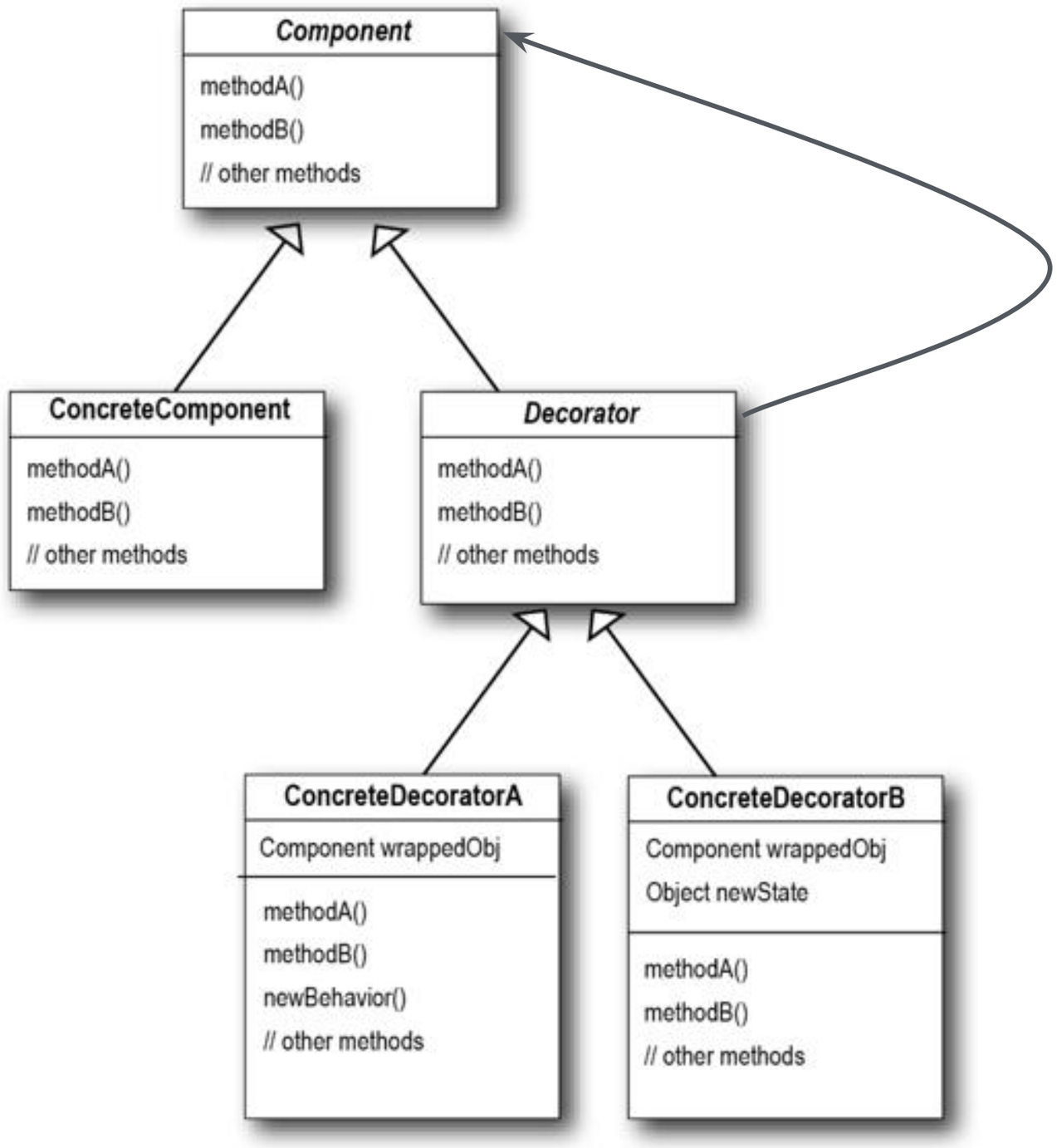
❸ The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.



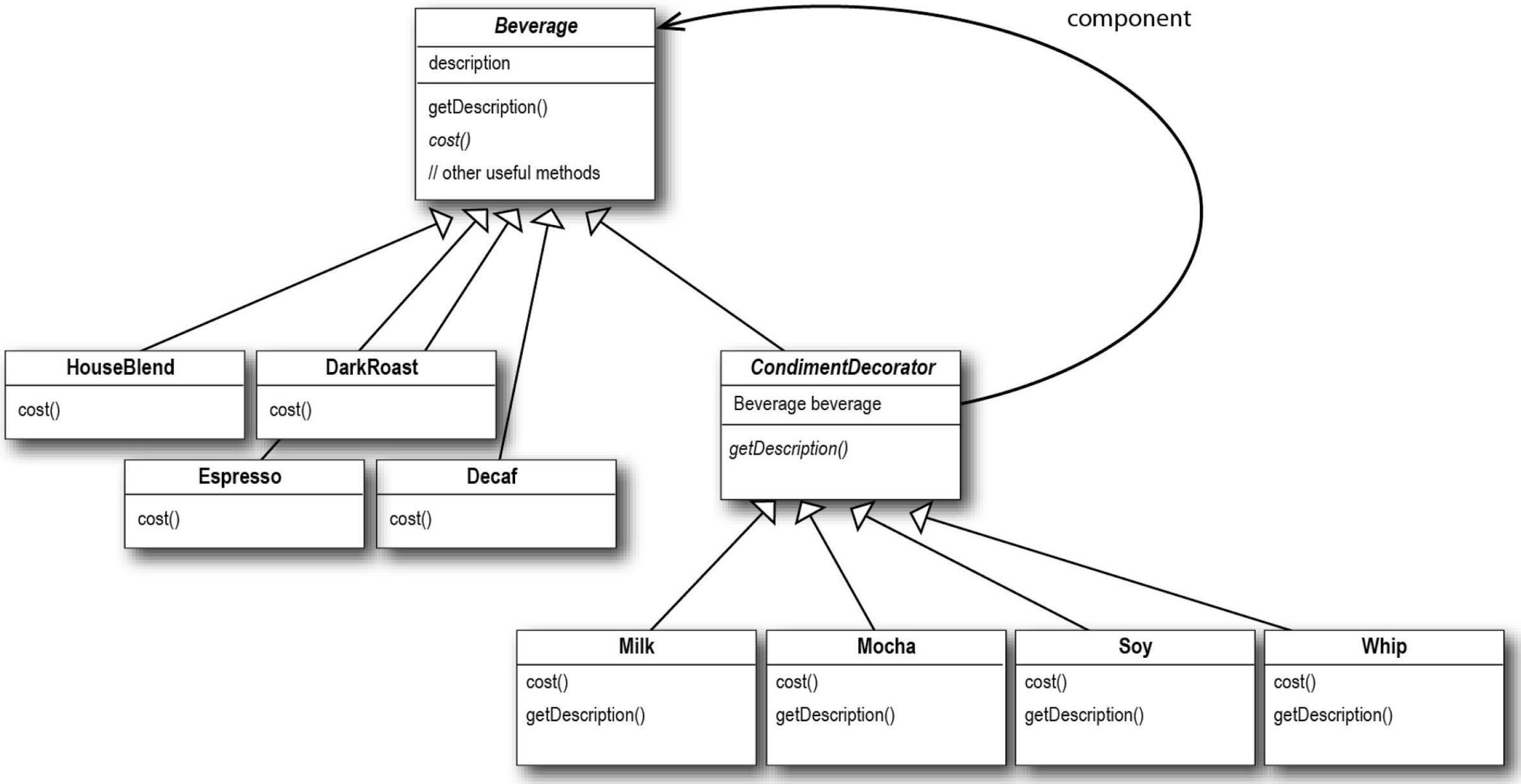
4. **Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.**



The Decorator Pattern



Starbuzz Coffee Design



Components

```
public abstract class Beverage {
   String description = "Unknown Beverage";

   public String getDescription() {
      return description;
   }

   public abstract double cost();
}

public class DarkRoast extends Beverage {
   public DarkRoast() {
      description = "Dark Roast Coffee";
   }

   public double cost() {
      return .99;
   }
}
```

Decorators

```
public abstract class CondimentDecorator extends Beverage {
   Beverage beverage;
   public abstract String getDescription();
}

public class Whip extends CondimentDecorator {

   public Whip(Beverage beverage) {
      this.beverage = beverage;
   }

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

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

Decorators

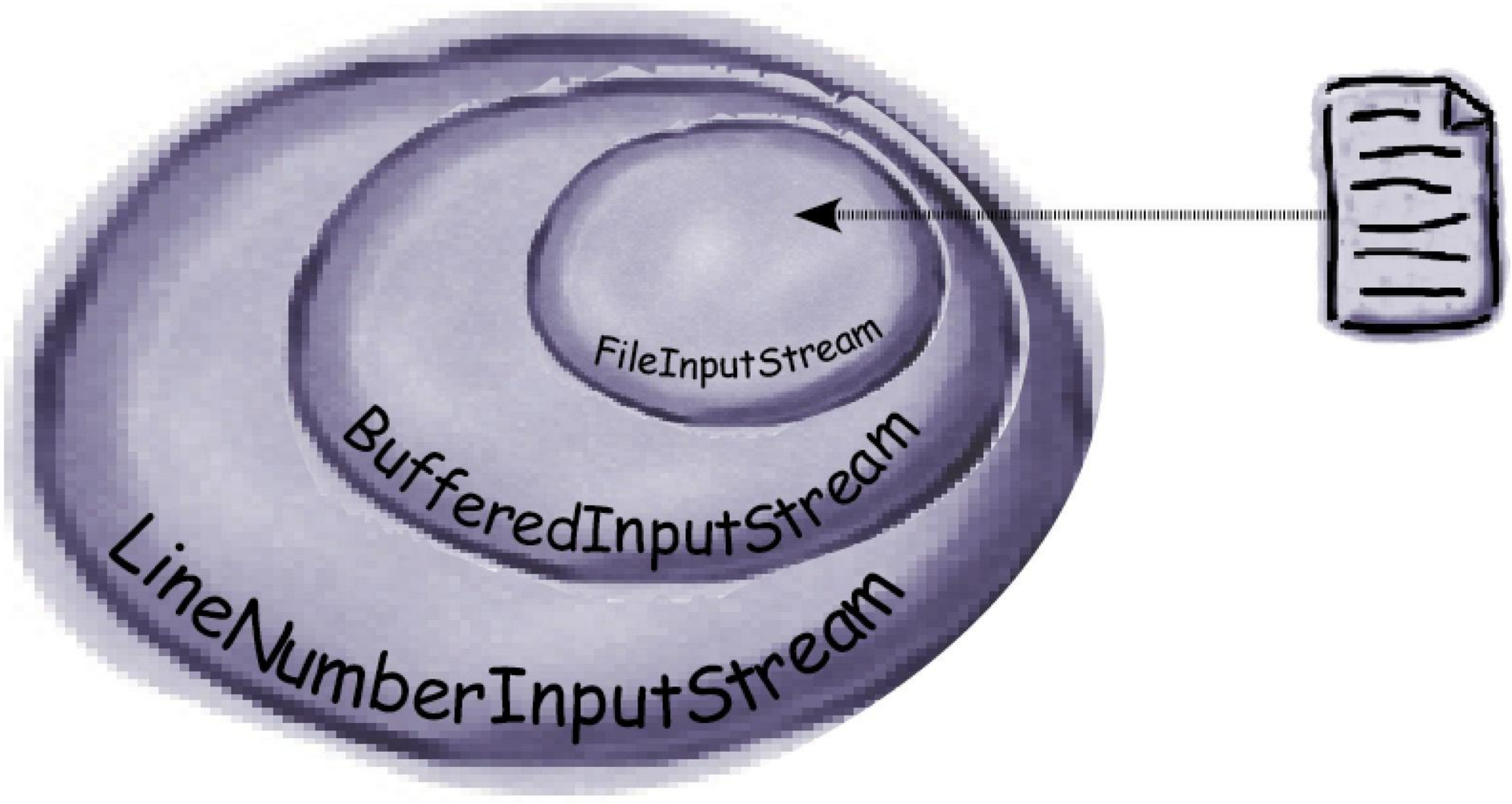
```
public class Mocha extends CondimentDecorator {

   public Mocha(Beverage beverage) {
      this.beverage = beverage;
   }

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

   public double cost() {
      return beverage.cost() + .20;
   }

}
```

Starbuzz

```
public class StarbuzzCoffee {

   public static void main(String args[]) {

      Beverage beverage = new DarkRoast();
      beverage = new Mocha(beverage);
      beverage = new Whip(beverage);
      System.out.println(beverage.getDescription()
          + " $" + beverage.cost());

   }
}
```

Quick Quiz: can anyone think of a Java library that uses the Decorator Pattern?

Quick Quiz: can anyone think of a Java library that uses a Decorator Pattern?



Staying Loosely Coupled

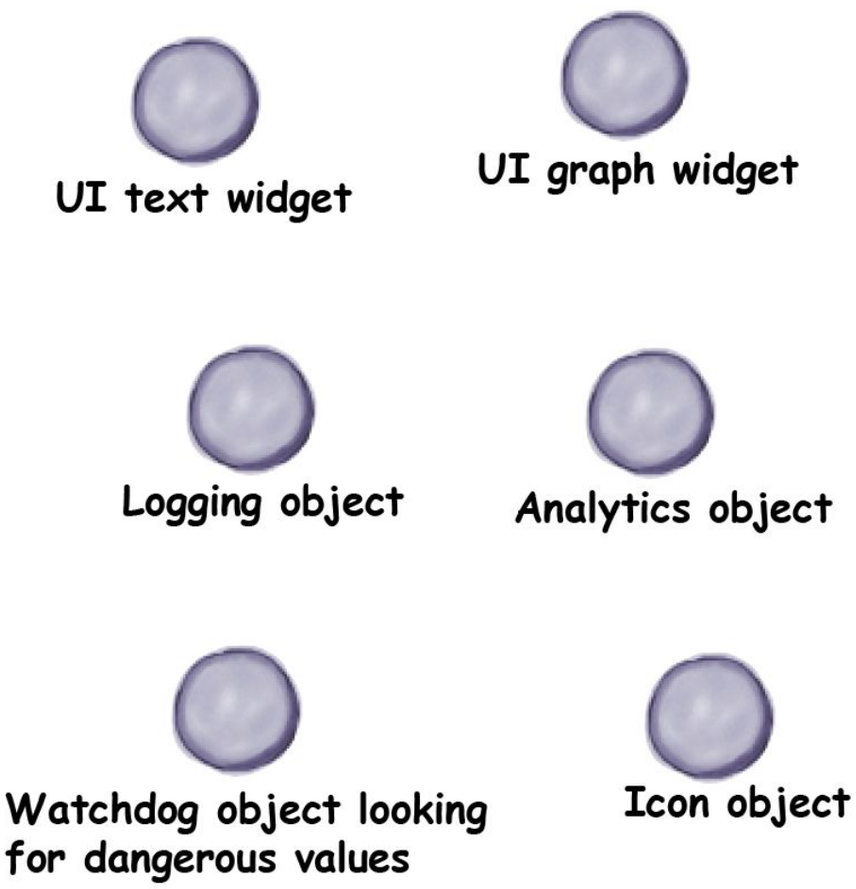
Loose Coupling

Design Principle

Strive for loosely coupled designs between objects that interact.

A Common Problem

An object with an Important Value

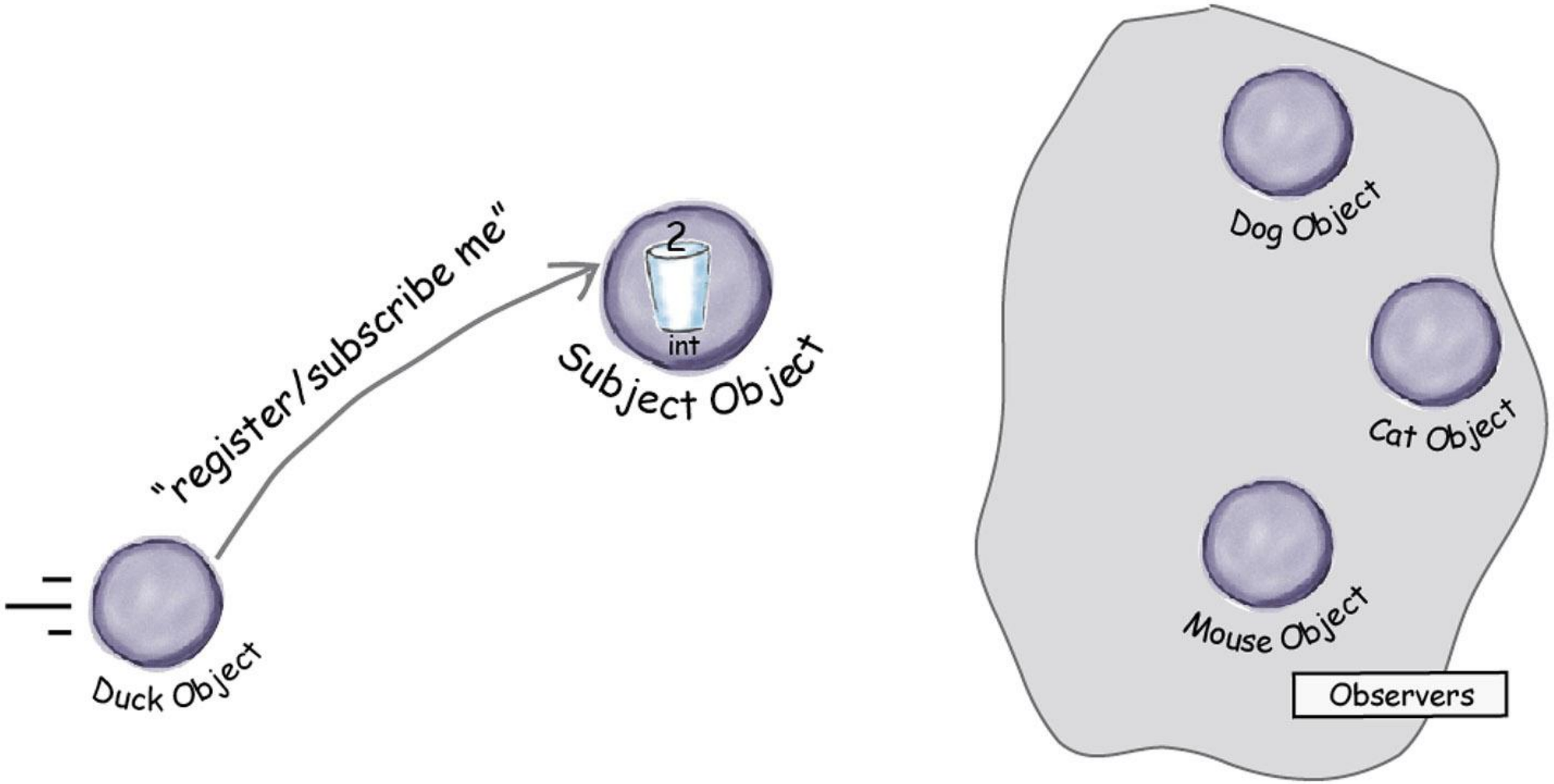


Objects that need to know the Important Value

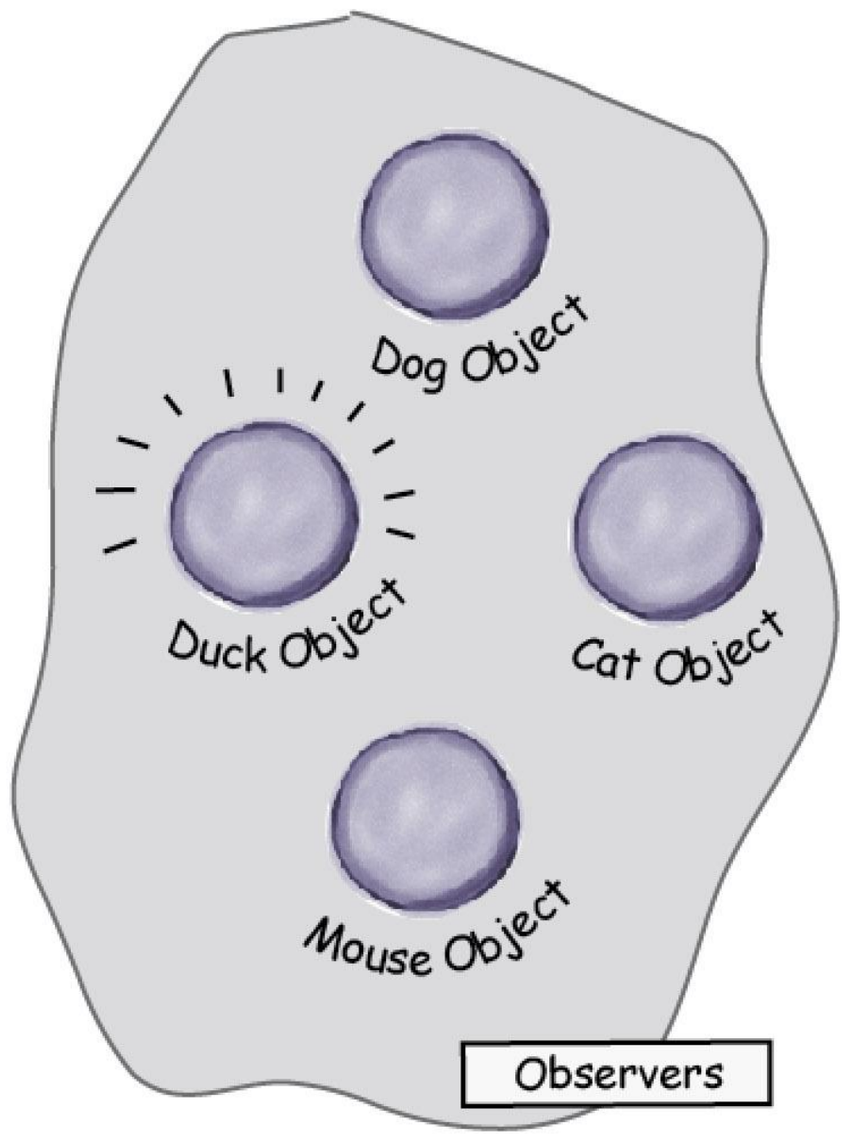
Use the Observer Pattern

- You subscribe to a news publisher.
- Whenever there's a new edition of the news (or a new blog post), you get it.
- If you don't want to get the paper (or the posts) anymore, you unsubscribe.
- Other subscribers continue to get the news.

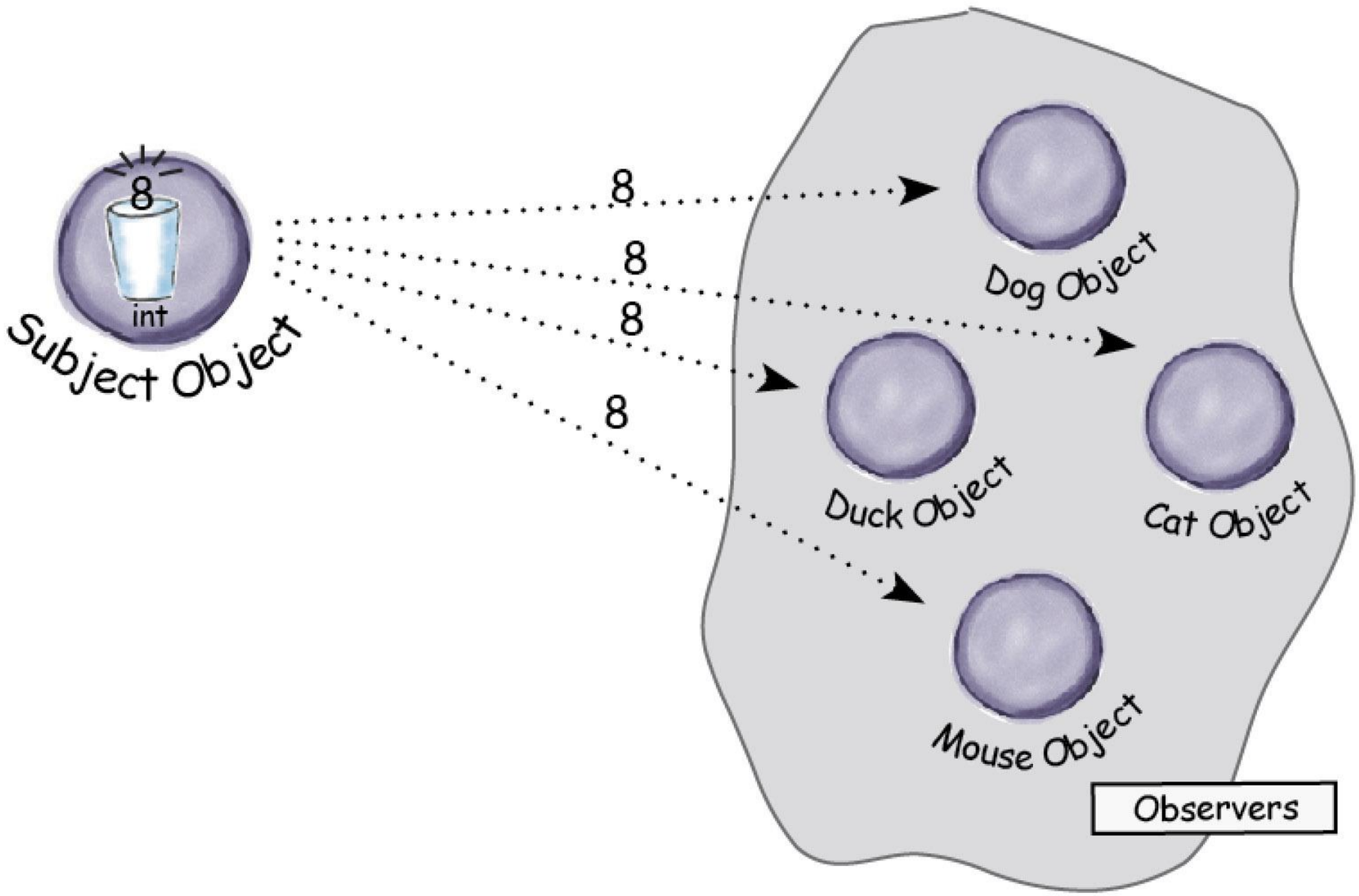
A day in the life of the Observer Pattern



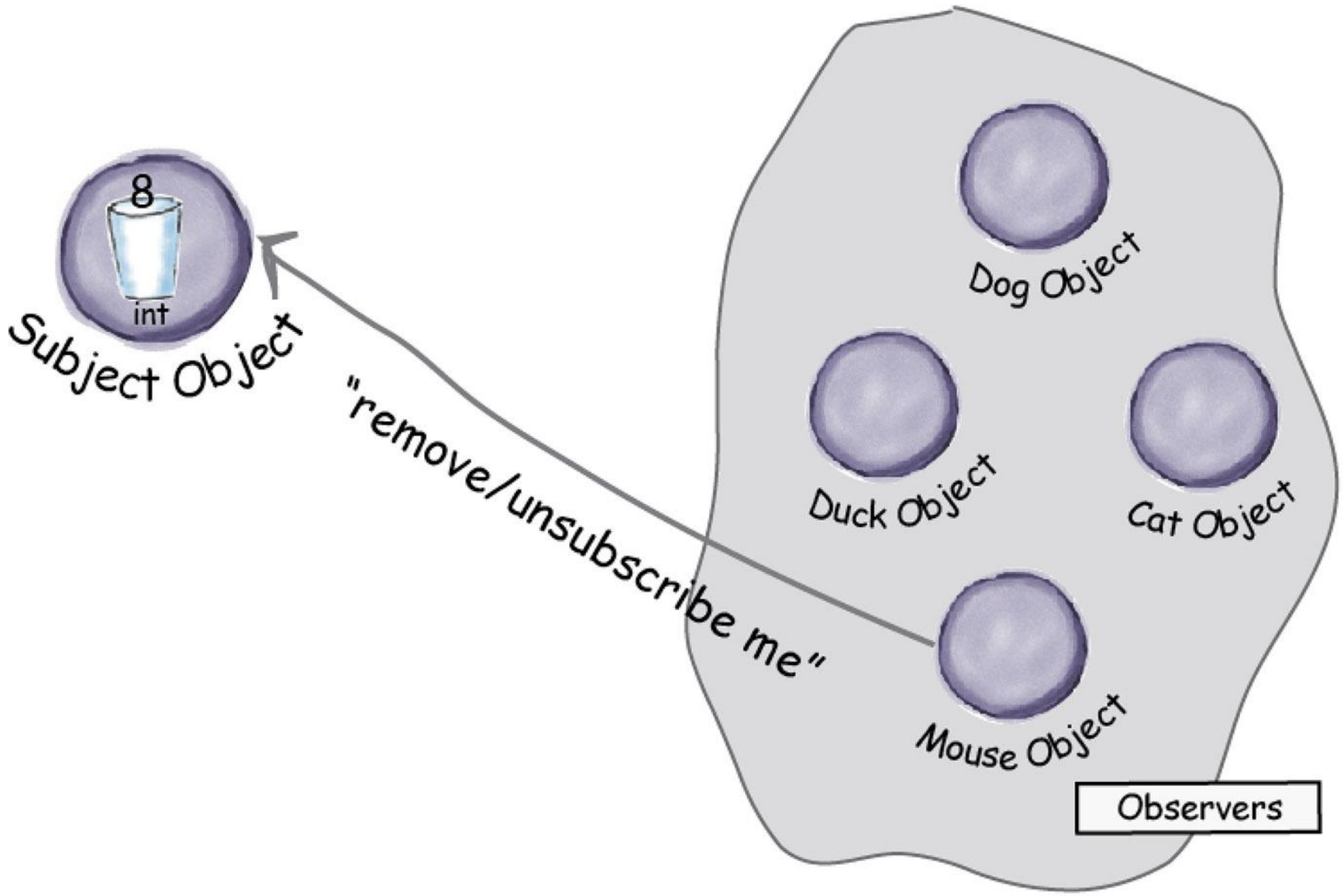
A day in the life of the Observer Pattern



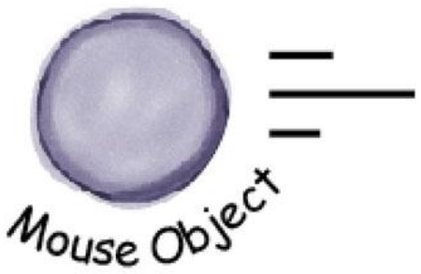
A day in the life of the Observer Pattern

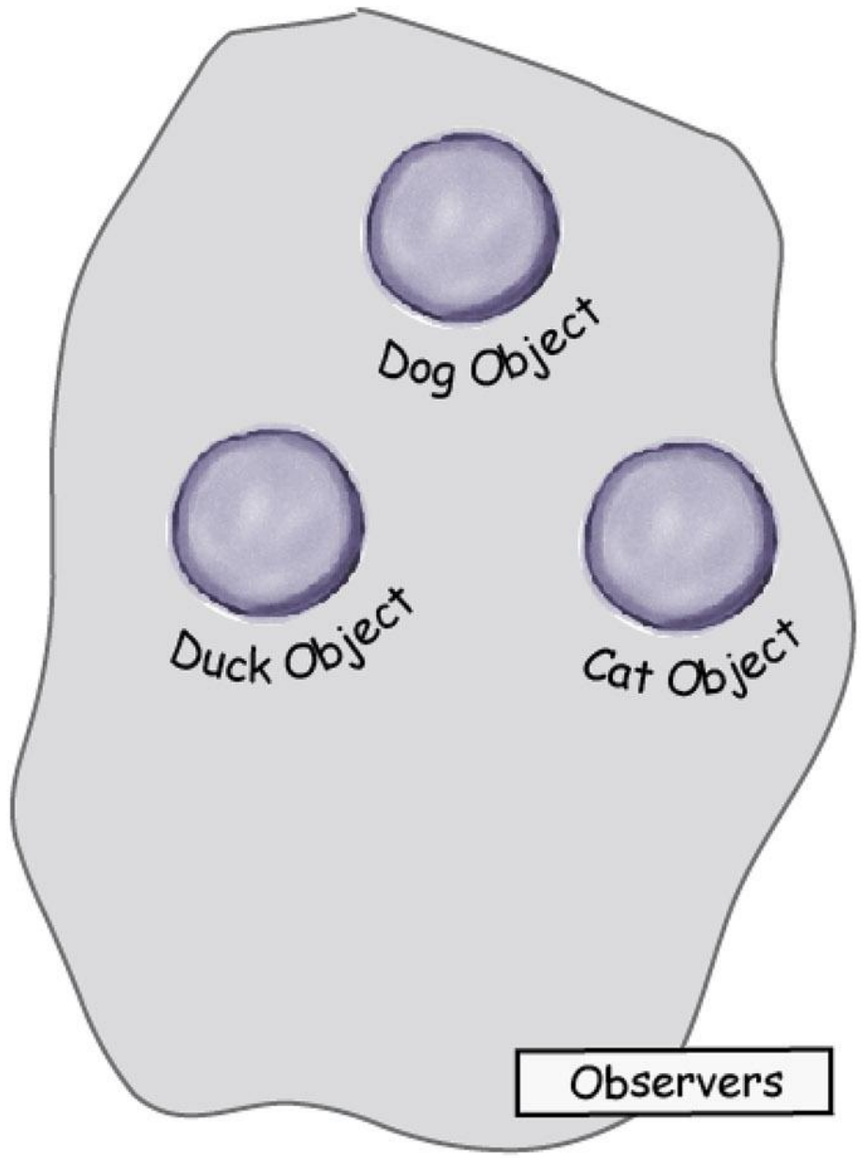


A day in the life of the Observer Pattern

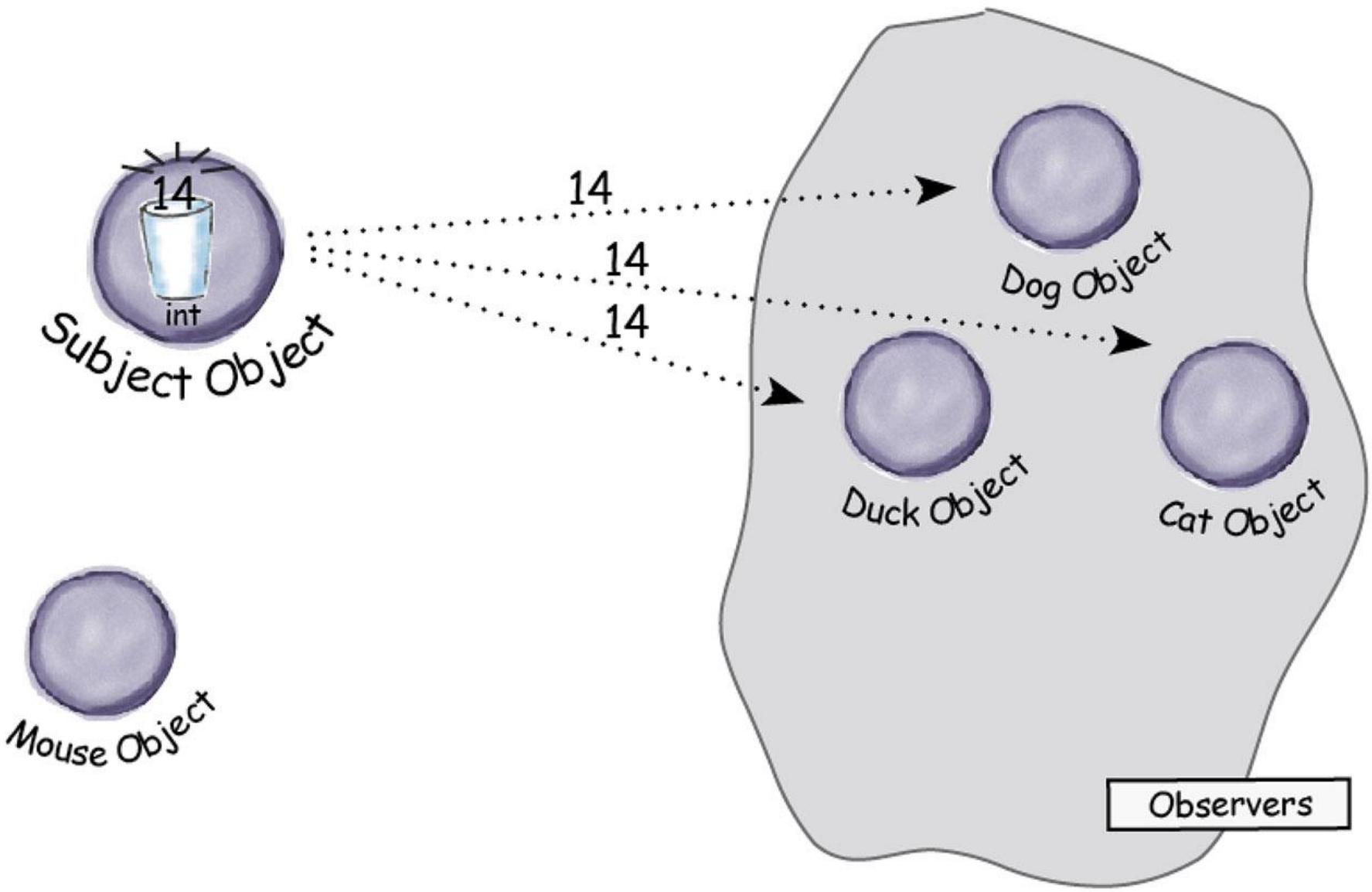


A day in the life of the Observer Pattern





A day in the life of the Observer Pattern



Observer Pattern, from Design Patterns: Elements of Reusable Object-Oriented Software.

OBSERVER

Object Behavioral

Intent

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

Motivation

A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.

The Observer pattern describes how to establish these relationships. They key objects in this pattern are the Subject and the Observer. A Subject may have any number of dependent Observers. All Observers are notified whenever the Subject undergoes a change in state. In response, each Observer will query the Subject to synchronize its state with the Subject's state.

(Also known as publish-subscribe.)

Applicability

Use the Observer pattern:

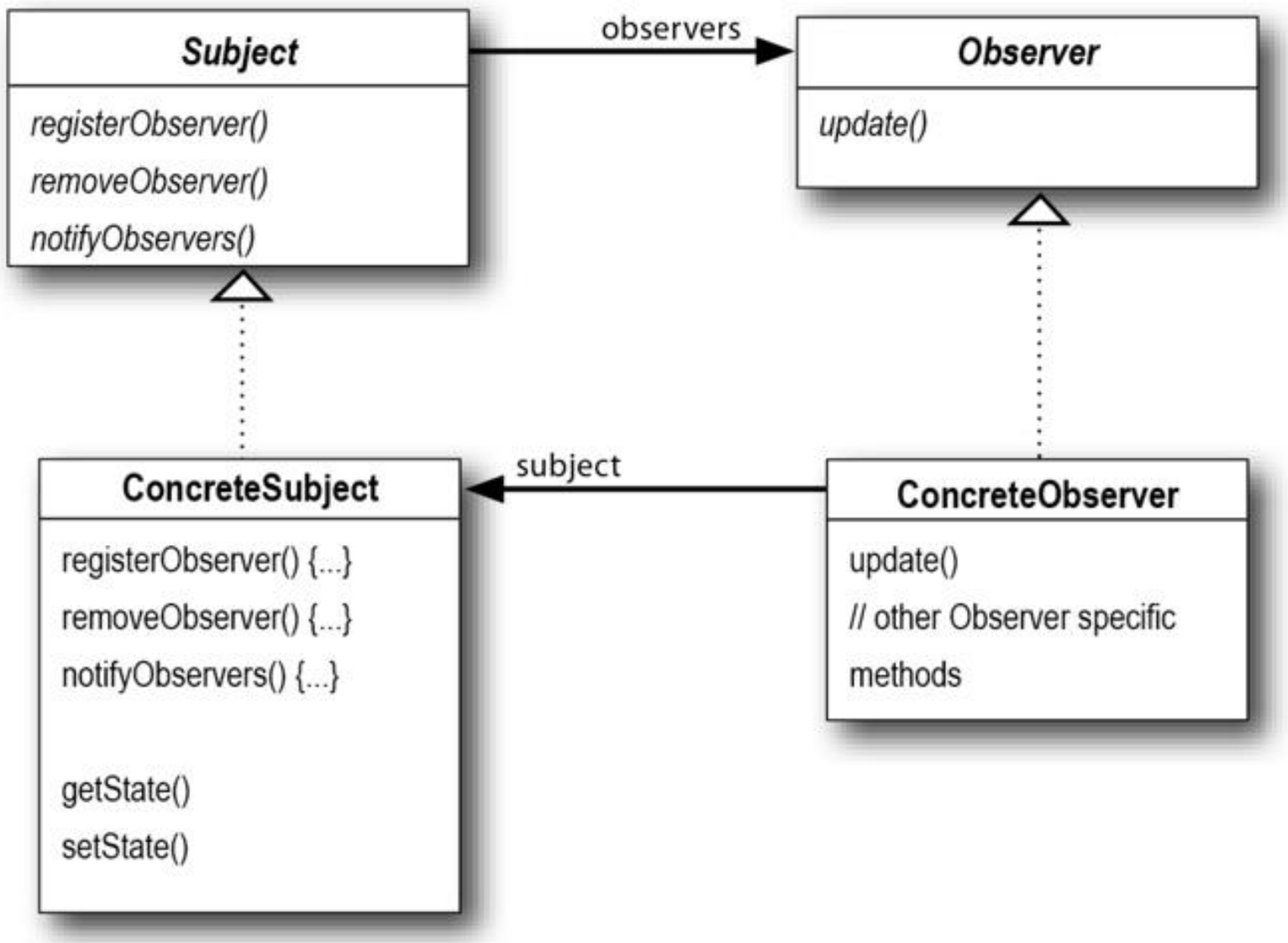
* when an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.

* when a change to one object requires changing others, and you don't know how many objects need to be changed.

* when an object should be able to notify other objects without making assumptions about who those objects are. In other words, you don't want those objects tightly coupled.

Structure

The Observer Pattern



The Subject Interface

```
public interface Subject {
   public void registerObserver(Observer o);
   public void removeObserver(Observer o);
   public void notifyObservers();
}
```

The Concrete Subject

```
public class SimpleSubject implements Subject {
   private ArrayList<Observer> observers;
   private int value = 0;

   public SimpleSubject() {
      observers = new ArrayList<Observer>();
   }

   public void registerObserver(Observer o) {
      observers.add(o);
   }

   public void removeObserver(Observer o) {
      ... // remove observer from the list
   }

   // more code here (next slide)
}
```

The Concrete Subject

```
public class SimpleSubject implements Subject {
   private ArrayList<Observer> observers;
   private int value = 0;

   // other code here

   public void notifyObservers() {
      for (Observer observer : observers) {
         observer.update(value);
      }
   }

   public void setValue(int value) {
      this.value = value;
      notifyObservers();
   }
}
```

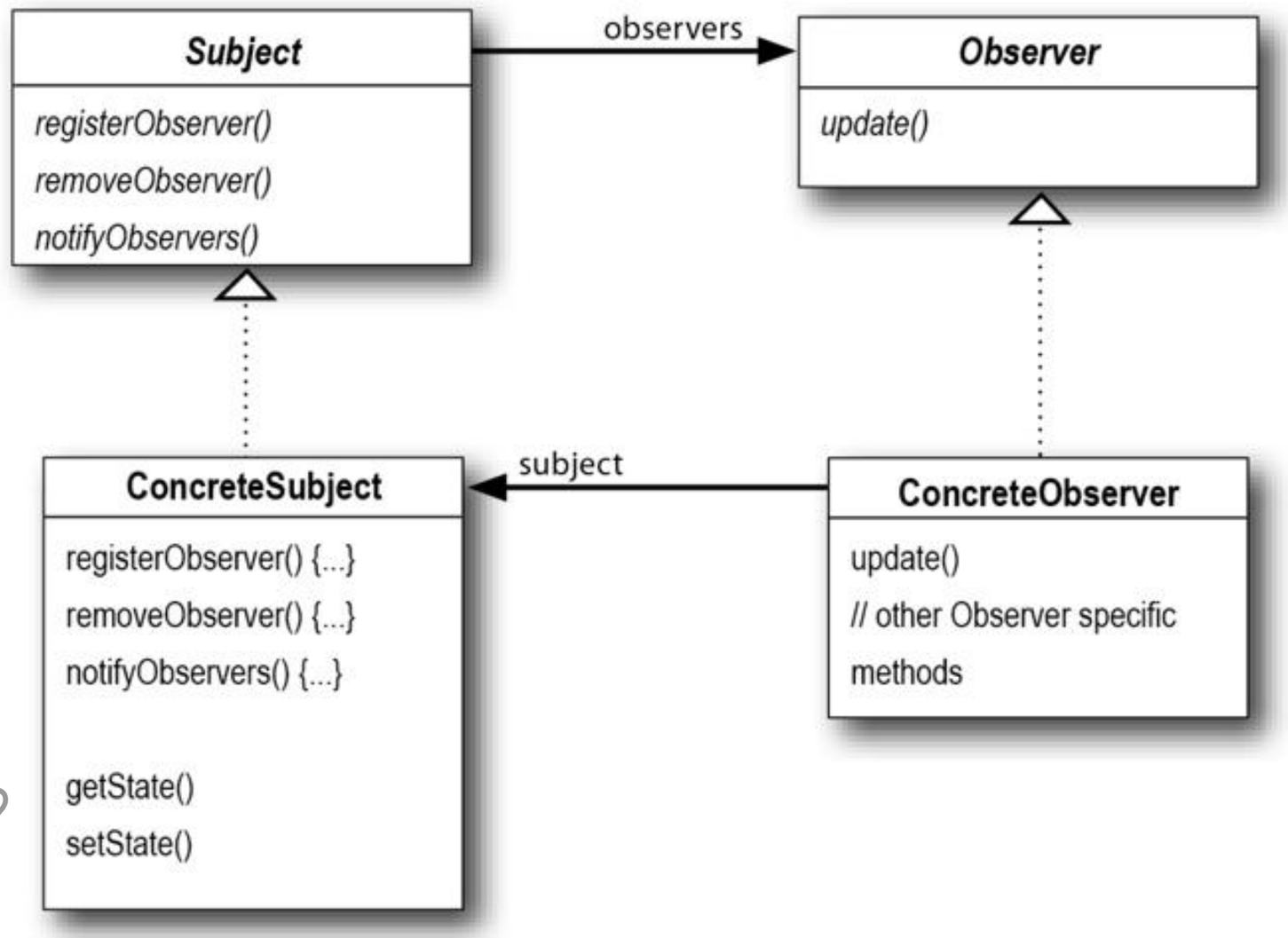
The Observer interface

```
public interface Observer {
   public void update(int value);
}
```

The Concrete Observer

```
public class SimpleObserver implements Observer {
   private int value;
   private Subject simpleSubject;

   public SimpleObserver(Subject simpleSubject) {
      this.simpleSubject = simpleSubject;
      simpleSubject.registerObserver(this);
   }

   public void update(int value) {
      this.value = value;
      display();
   }

   public void display() {
      System.out.println("Value: " + value);
   }
}
```

Push vs Pull



Observer can use getState() to get the state of the Subject.

Pull: Subject

```
public class SimpleSubject implements Subject {
   …
    private int value = 0;

   …
    public void notifyObservers() {
       for (Observer observer : observers) {
          observer.update();
       }
    }

    public int getValue() {
       return value;
    }
}
```

Pull: Observer

```
public interface Observer {
   public void update();
}

public class SimpleObserver implements Observer {
    …
    int value;

   public void update() {
      this.value = subject.getValue();
      display();
   }
    …
}
```

Event handling

JavaScript:

```
buttonElement.addEventListener(“click”, function() {
   console.log(“Come on, do it!”);
}, false);
```

Java:

```
frame = new JFrame();
JButton button = new JButton("Should I do it?");

button.addActionListener(event ->
   System.out.println("Don't do it, you might regret it!")
);
button.addActionListener(event ->
   System.out.println("Come on, do it!")
);
```

Encapsulating Iteration

Breaking News: Objectville Diner and Objectville Pancake House Merge

Design Patterns
BOOTCAMP

Which would be great, except...

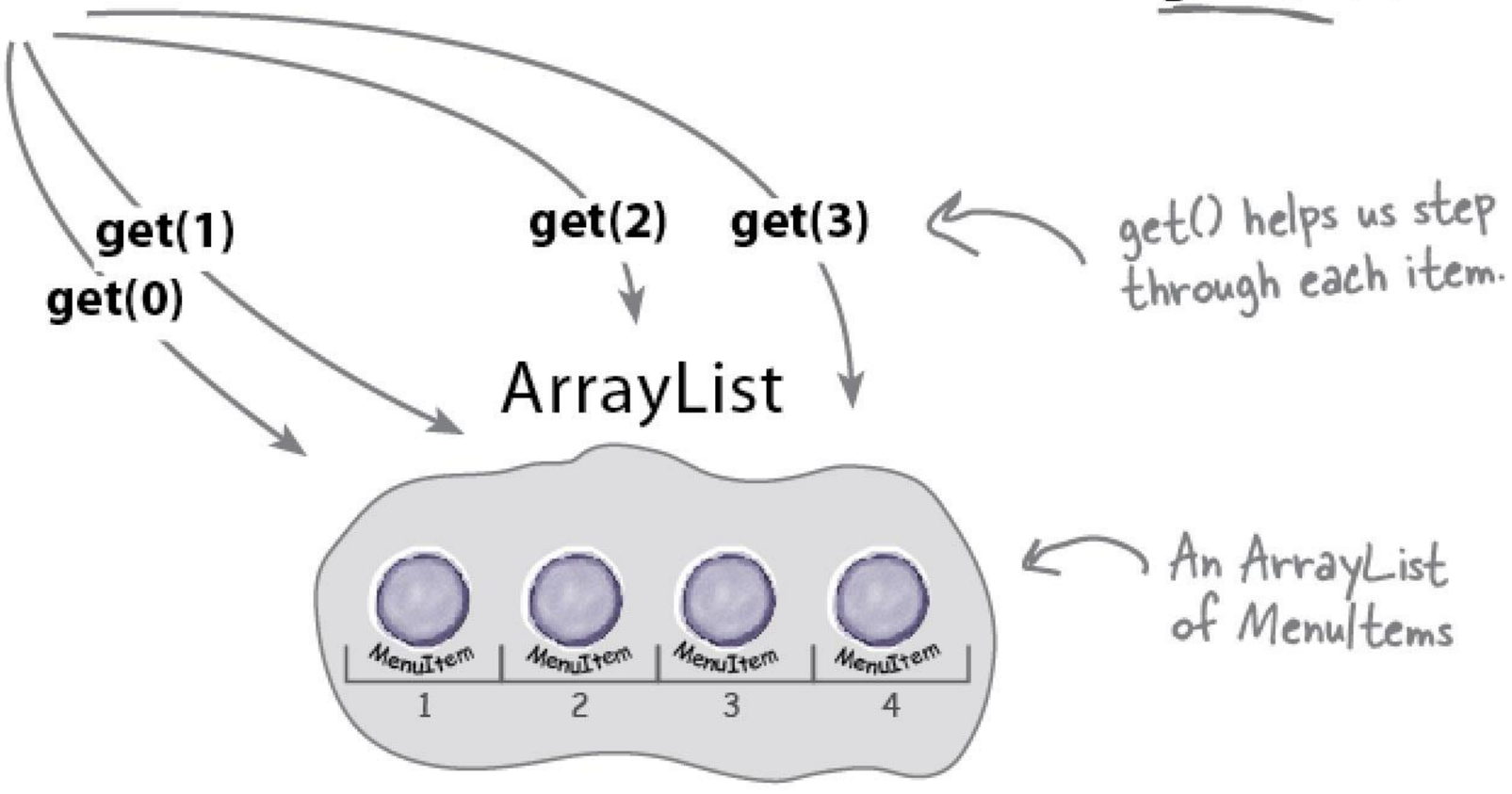
We've got different Data Structures

```
MenuItem[] menuItems;
```


```
ArrayList<MenuItem> menuItems;
```

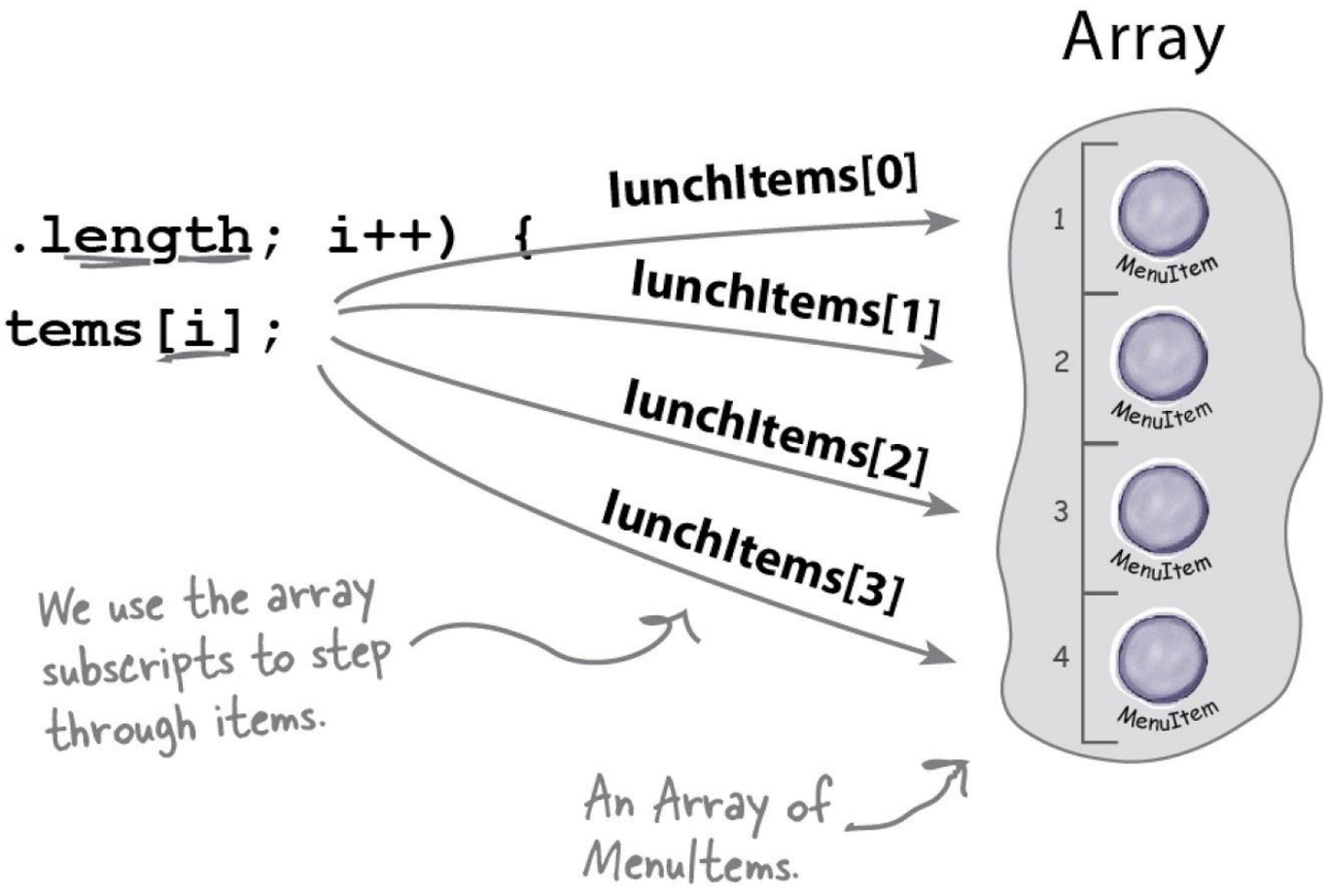
Iterating with an array list

```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = breakfastItems.get(i);
}
```



Iterating with an array

```
for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
}
```



So we have some problems with the waitress...

A. We have duplicate code: a loop for each type of menu.

B. The Waitress has to know how each menu type is implemented.

C. We are coding to concrete implementations, if we ever want to replace the data structure we have a lot of coding to do (we're not closed for modification!).

D. Waitress has dual responsibilities (not a single one).

ITERATOR — Object Behavioral

Intent

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Motivation

An aggregate object such as a list should give you a way to access its elements without exposing its internal structure. Moreover you might want to traverse the list in different ways, depending on what you want to accomplish. But you don't want to bloat the List interface with operations for different traversals, even if you could anticipate the ones you will need.

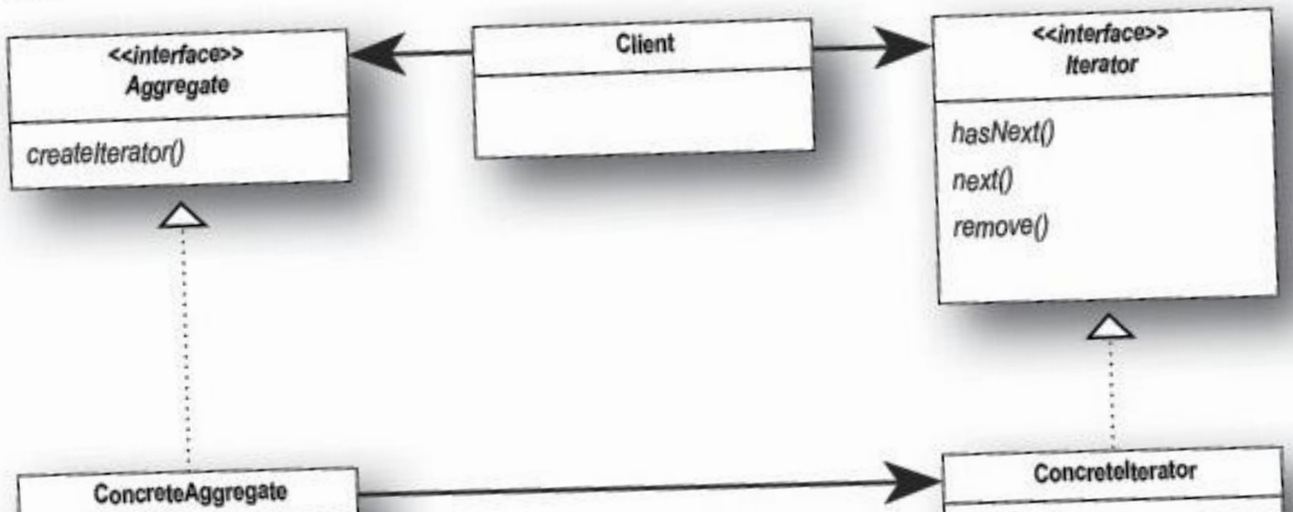
The Iterator pattern lets you do all this. The key idea in this pattern is to take the responsibility for access and traversal out of the list object and put it into an iterator object. The Iterator class defines an interface for accessing the list's elements. An iterator object is responsible for keeping track of the current element; that is, it knows which elements have been traversed already.

Applicability

Use the Iterator pattern:

* to access an aggregate object's contents without exposing its internal reprsentation.
* to support multiple traversals of aggregate objects.
* to provide a uniform interface for traversing different aggregate structures.

Structure



Iterator Pattern, from Design Patterns: Elements of Reusable Object-Oriented Software.

ITERATOR

Object Behavioral

Intent

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Motivation

An aggregate object such as a list should give you a way to access its elements without exposing its internal structure. Moreover you might want to traverse the list in different ways, depending on what you want to accomplish. But you don't want to bloat the List interface with operations for different traversals, even if you could anticipate the ones you will need.

The Iterator pattern lets you do all this. The key idea in this pattern is to take the responsibility for access and traversal out of the list object and put it into an iterator object. The Iterator class defines an interface for accessing the list's elements. An iterator object is responsible for keeping track of the current element; that is, it knows which elements have been traversed already.

Applicability

Use the Iterator pattern:

* to access an aggregate object's contents without exposing its internal reprsentation.

* to support multiple traversals of aggregate objects.

* to provide a uniform interface for traversing different aggregate structures.

Structure

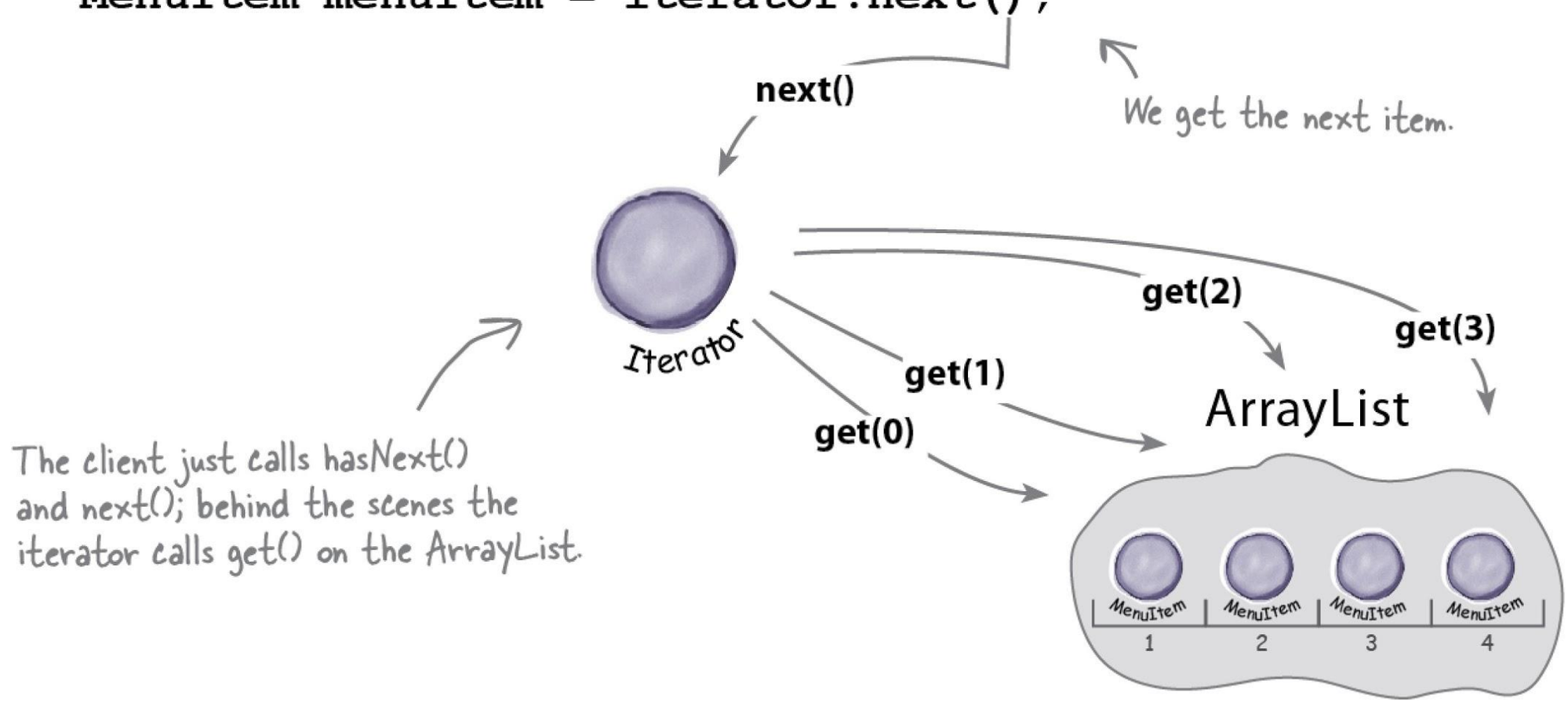
Using an Iterator for ArrayList

We ask the breakfastMenu for an iterator of its MenuItems.

```
Iterator iterator = breakfastMenu.createIterator();

while (iterator.hasNext()) {
    MenuItem menuItem = iterator.next();
}
```

And while there are more items left...

We get the next item.



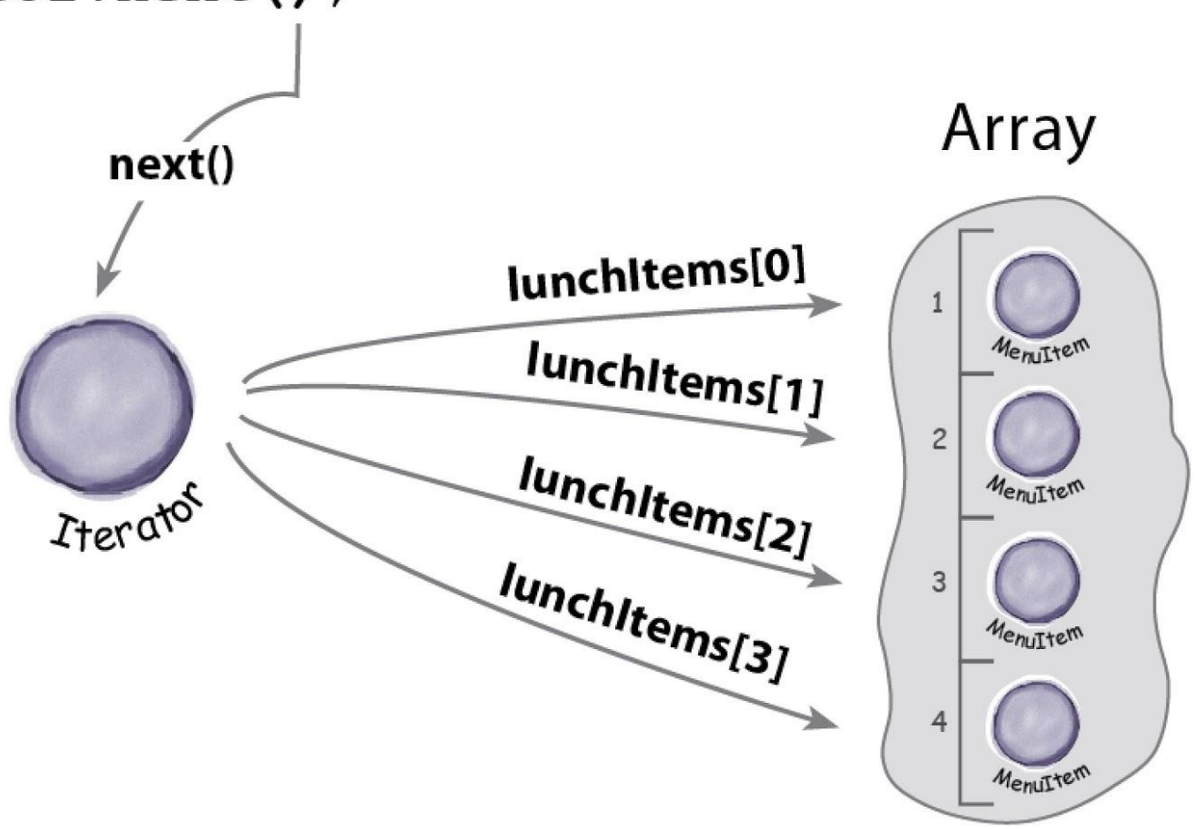
The client just calls hasNext() and next(); behind the scenes the iterator calls get() on the ArrayList.

Using an Iterator for Array

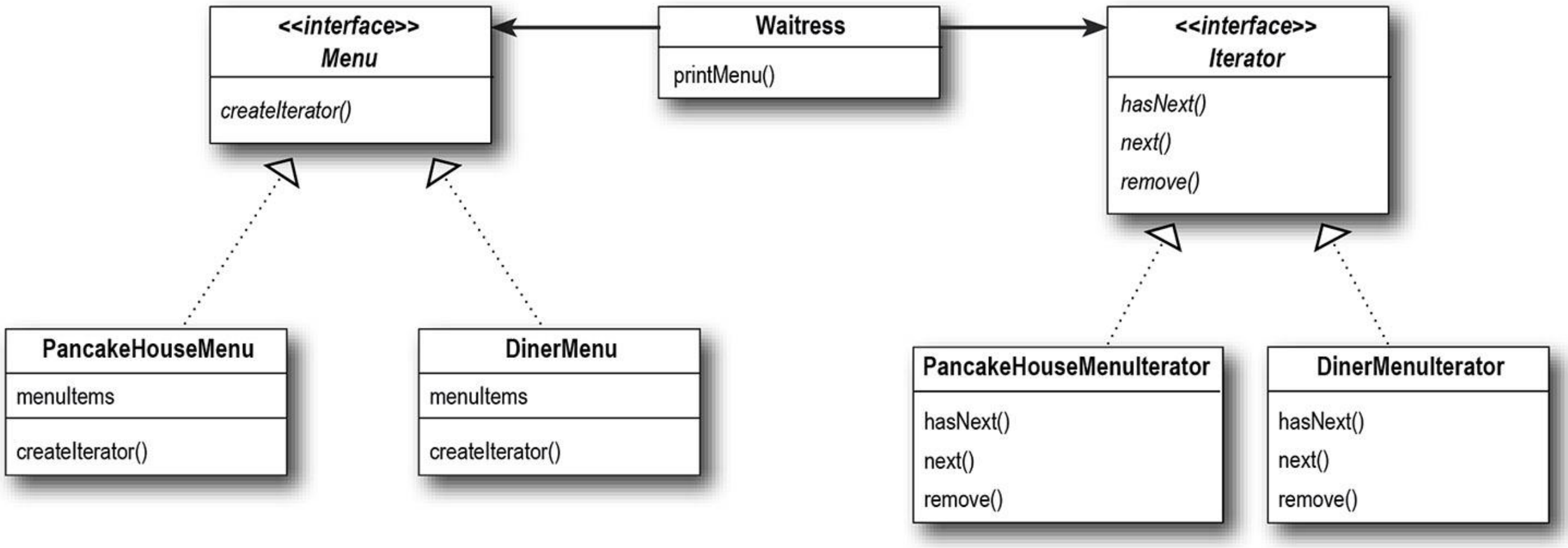
```
Iterator iterator = lunchMenu.createIterator();

while (iterator.hasNext()) {
    MenuItem menuItem = iterator.next();
}
```

Wow, this code is exactly the same as the breakfastMenu code.

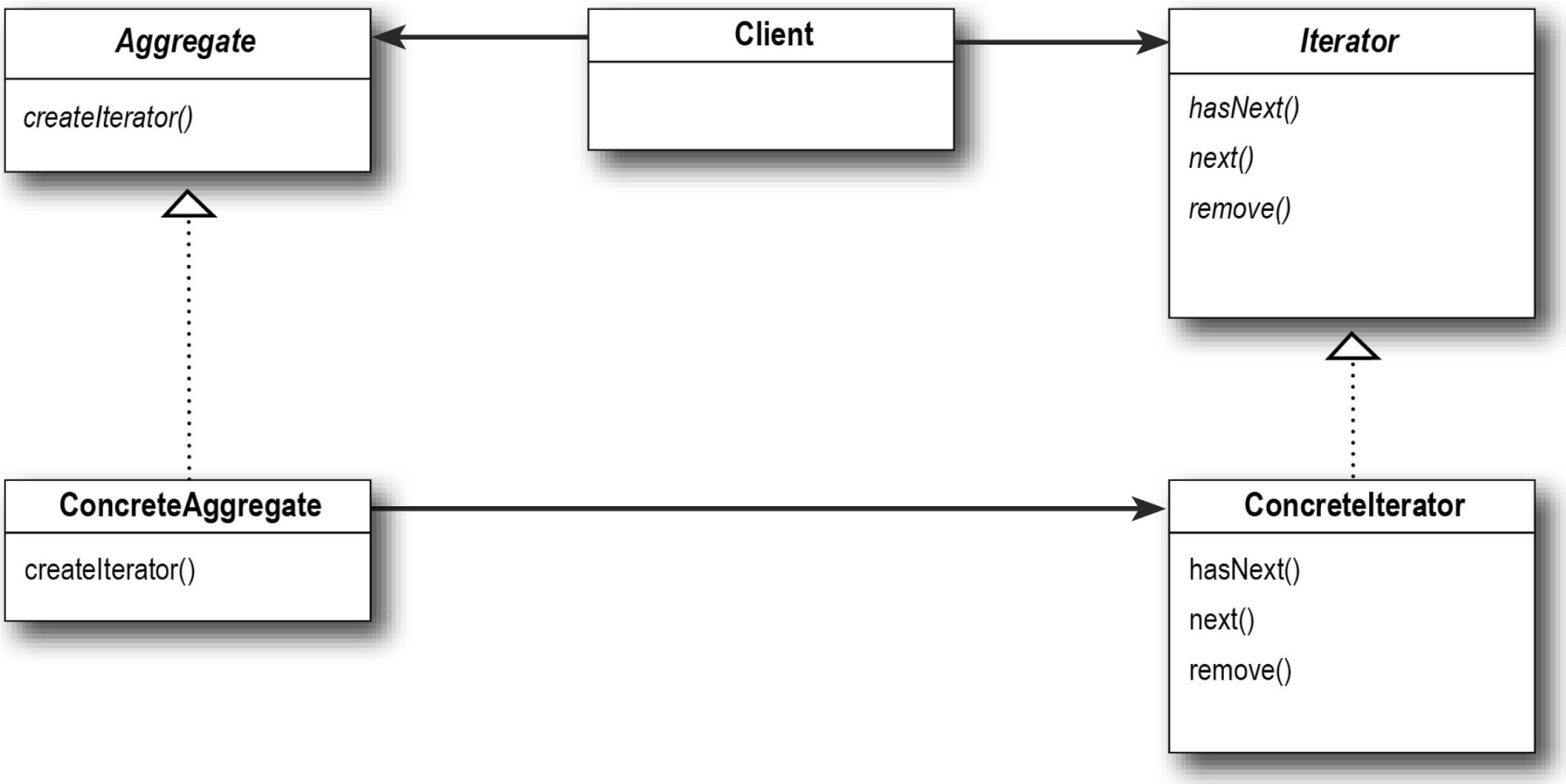
Same situation here: the client just calls hasNext() and next(); behind the scenes, the iterator indexes into the Array.



The new, improved design



The Iterator Pattern



The Diner Iterator

```
public class DinerMenuIterator implements Iterator<MenuItem> {
   MenuItem[] list;
   int position = 0;

   public DinerMenuIterator(MenuItem[] list) {
      this.list = list;
   }
   public MenuItem next() {
      MenuItem menuItem = list[position];
      position = position + 1;
      return menuItem;
   }
   public boolean hasNext() {
      if (position >= list.length || list[position] == null) {
         return false;
      } else {
         return true;
      }
   }

   ...
}
```

New and improved waitress code

```
public class Waitress {
   Menu pancakeHouseMenu;
   Menu dinerMenu;

   public Waitress(Menu pancakeHouseMenu, Menu dinerMenu) {
      this.pancakeHouseMenu = pancakeHouseMenu;
      this.dinerMenu = dinerMenu;
   }

   public void printMenu(Iterator<MenuItem> iterator) {
      while (iterator.hasNext()) {
         MenuItem menuItem = iterator.next();
         System.out.print(menuItem.getName() + ", ");
         System.out.print(menuItem.getPrice() + " -- ");
         System.out.println(menuItem.getDescription());
      }
   }

   …
}
```

Using built-in iterators

To create your own menu iterator:

```
public Iterator<MenuItem> createIterator() {
   return new PancakeHouseMenuIterator(menuItems);
}
```

To use a built-in iterator:

```
public Iterator<MenuItem> createIterator() {
   return menuItems.iterator();
}
```

More on treating objects uniformly

COMPOSITE

Object Structural

Intent

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Motivation

Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components. The user can group components to form larger components. A simple implementation could define classes for graphical primitives such as Text and Lines plus other classes that act as containers for these primitives.

But there's a problem with this approach: Code that uses these classes must treat primitive and container objects differently, even if most of the time the user treats them identically. The Composite pattern describes how to use recursive composition so that clients don't have to make this distinction.

Applicability

Use the Composite pattern when:

* You want to represent part-whole hierarchies of objects.

* You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite struture uniformly.

Structure

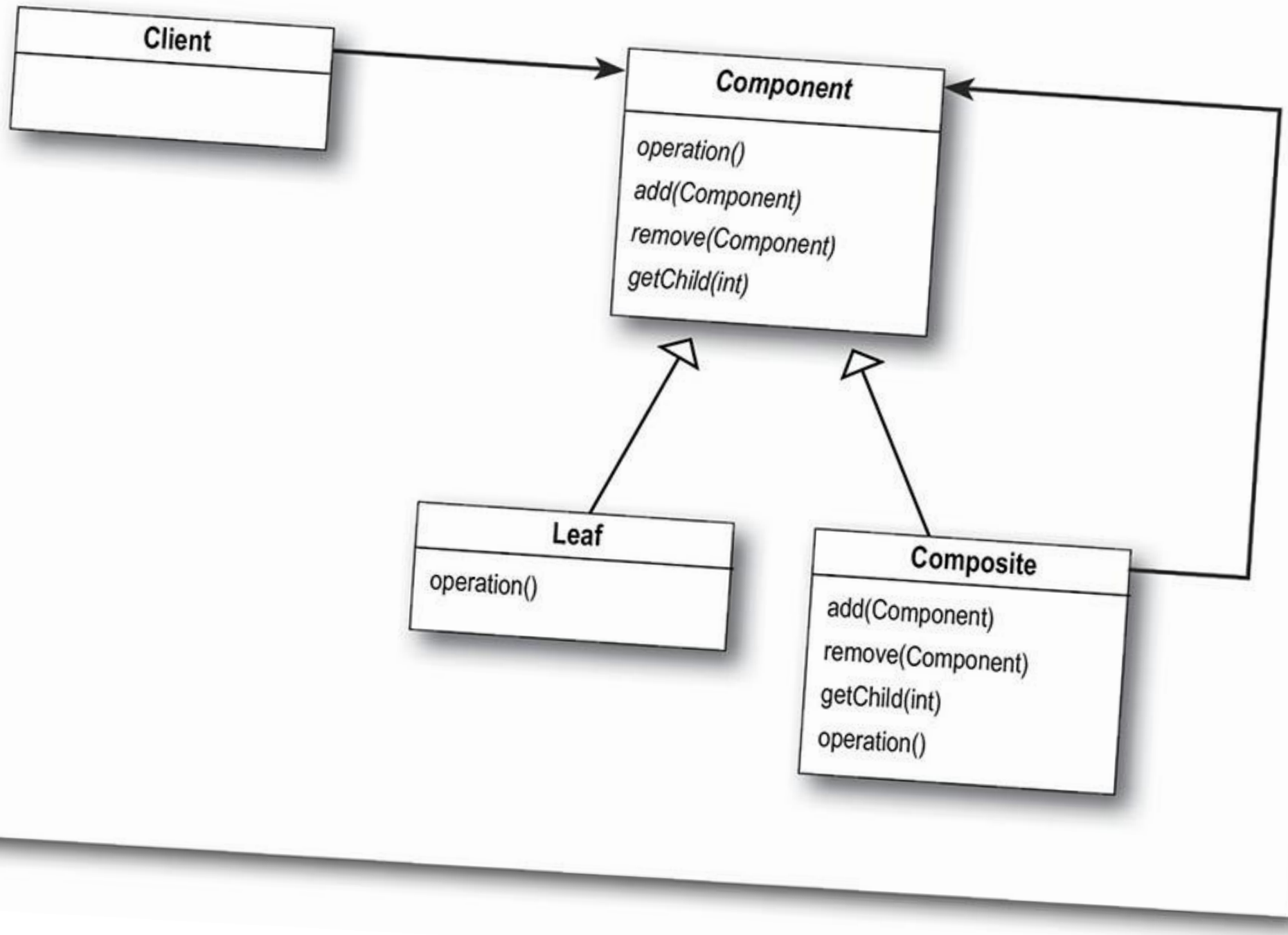
Applicability

Use the Composite pattern when:

* You want to represent part-whole hierarchies of objects.

* You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite struture uniformly.

Structure



The One of a Kind Pattern

SINGLETON

Object Creational

Intent

Ensure a class only has one instance, and provide a global point of access to it.

Motivation

It's important for some classes to have exactly one instance. How do we ensure that a class has only one instance and that the instance is easily accessible? A global variable makes an object accessible, but it doesn't keep you from instantiating multiple objects.

A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance. This is the Singleton pattern.

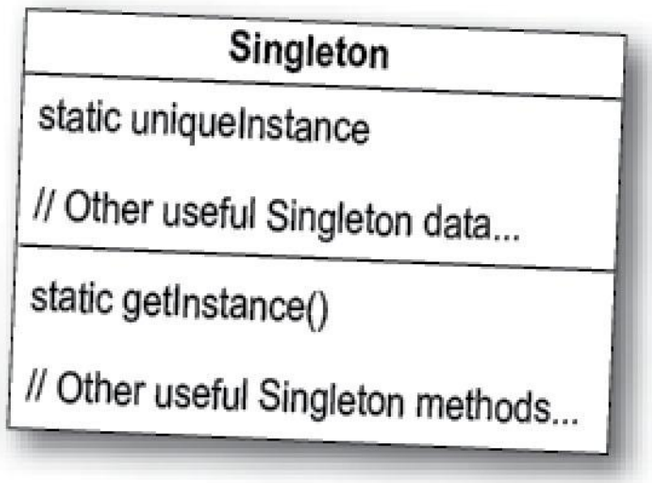
Applicability

Use the Singleton pattern when:

* there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.

* when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Structure



Participants

Singleton - defines an Instance operation that lets cli

How do we create a single object?

```
new MyObject();
```

How do we create a second object?

```
new MyObject();
new MyObject();
```

Can we always create another object using new?

```
new MyObject();
new MyObject();
new MyObject();
```

In Java, a public class typically has a public constructor method:

```
public MyClass {
    public MyClass() {
      // code here
    }

    // other methods here
}
```

You can make a constructor private instead.

```
public MyClass {

    private MyClass() {
      // code here
    }

    // other methods here
}
```

To make an instance, we have to call the constructor from MyClass

```
public MyClass {

    private MyClass() { … }

    public static MyClass getInstance() {
      return new MyClass();
    }

    // other methods here
}
```

To call a class method:

```
public MyClass {
    private MyClass() { … }
    public static MyClass getInstance() {
      return new MyClass();
    }
    ...
}

MyClass singleton = MyClass.getInstance();
```

Make sure there's only one!

```
public MyClass {
  private static MyClass uniqueInstance;

  private MyClass() {}

  public static MyClass getInstance() {
    if (uniqueInstance == null) {
      uniqueInstance = new MyClass();
    }
    return uniqueInstance;
  }
}
```

Be careful in multi-threaded environments

```
public MyClass {
  private static MyClass uniqueInstance;

  private MyClass() {}

  public static synchronized MyClass getInstance()
  {
    if (uniqueInstance == null) {
      uniqueInstance = new MyClass();
    }
    return uniqueInstance;
  }
}
```

An easier way in Java

```
public enum Singleton {
   UNIQUE_INSTANCE;
   public String getDescription() {
      return "I'm a thread safe Singleton!";
   }
}

public class SingletonClient {
   public static void main(String[] args) {
      Singleton singleton = Singleton.UNIQUE_INSTANCE;
      System.out.println(singleton.getDescription());
   }
}
```

Watch your design principles...

Design Principle

Strive for loosely coupled designs between objects that interact.

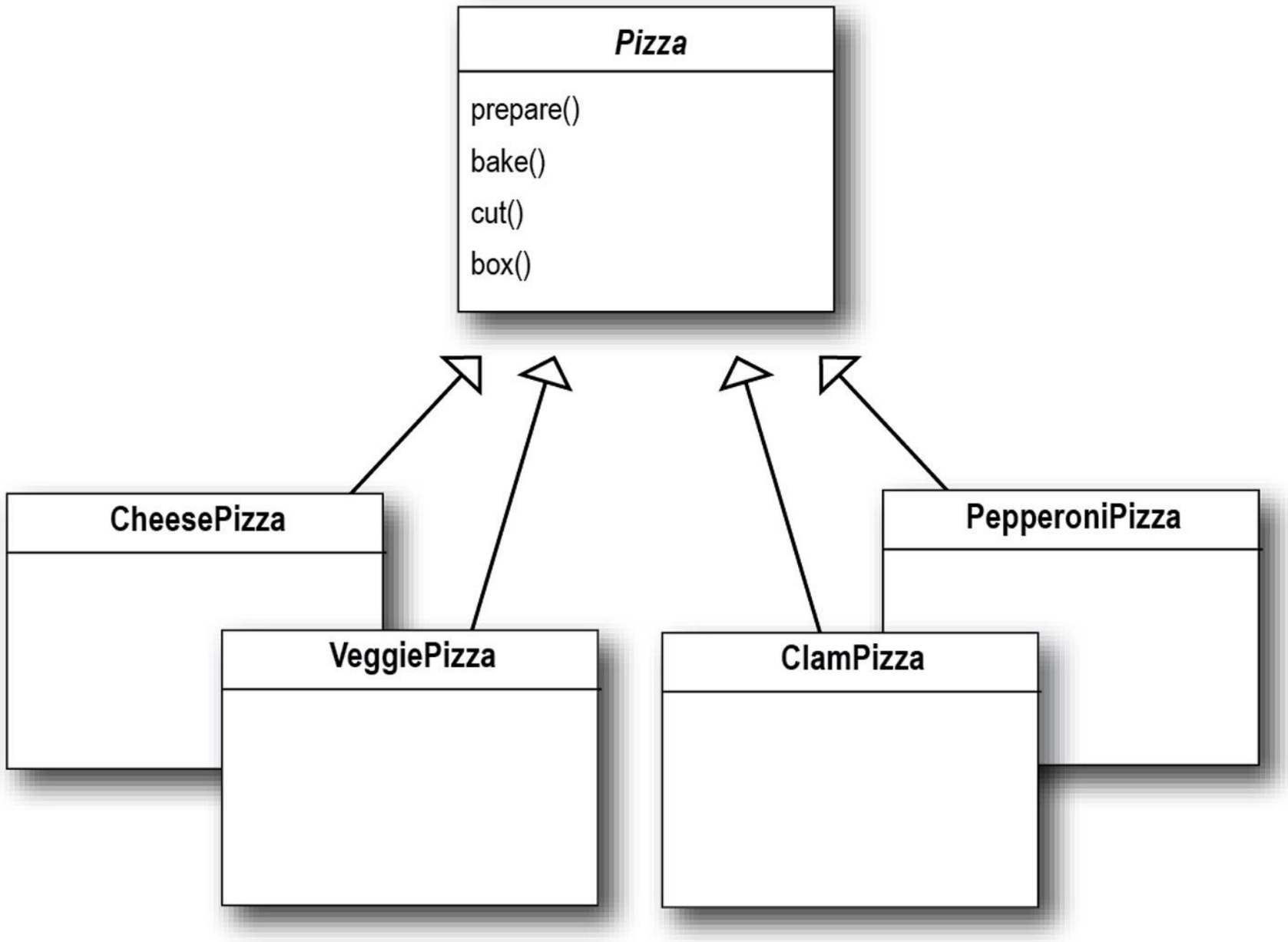
Encapsulating Object Creation

Design Principle

Program to an interface
not an implementation.

Question: We aren't supposed to program to an implementation, but every time we use **`new`**, that's exactly what we're doing, right?

The **new** Conundrum



The **new** Conundrum

```
Pizza orderPizza() {
   Pizza pizza = new Pizza();

   pizza.prepare();
   pizza.bake();
   pizza.cut();
   pizza.box();
   return pizza;
}
```

...but our concrete classes are CheesePizza, VeggiePizza, etc.

Coping Strategy

```
Pizza orderPizza(String type) {
   Pizza pizza;
   if (type.equals("cheese")) {
      pizza = new CheesePizza();
   } else if (type.equals("pepperoni")) {
      pizza = new PepperoniPizza();
   } else if (type.equals("clam")) {
      pizza = new ClamPizza();
   } else if (type.equals("veggie")) {
      pizza = new VeggiePizza();
   }
   pizza.prepare();
   pizza.bake();
   pizza.cut();
   pizza.box();
   return pizza;
}
```

What about Change?

```
Pizza orderPizza(String type) {
   Pizza pizza;
   if (type.equals("cheese")) {
      pizza = new CheesePizza();
   } else if (type.equals("clam")) {
      pizza = new ClamPizza();
   } else if (type.equals("greek") {
      pizza = new GreekPizza();
   } else if (type.equals("pepperoni") {
      pizza = new PepperoniPizza();
   } else if (type.equals("veggie") {
      pizza = new VeggiePizza();
   }
   // IMPORTANT CODE HERE (order, prepare, etc)
}
```

How might you take all the parts of your application that instantiate concrete classes and separate or encapsulate them from the rest of your application?

Let's Encapsulate Object Creation

```
if (type.equals("cheese")) {
   pizza = new CheesePizza();
} else if (type.equals("pepperoni") {
   pizza = new PepperoniPizza();
} else if (type.equals("veggie") {
   pizza = new VeggiePizza();
}
```

```
Pizza orderPizza(String type) {
   Pizza pizza;

   pizza.prepare();
   pizza.bake();
   pizza.cut();
   pizza.box();
   return pizza;
}
```

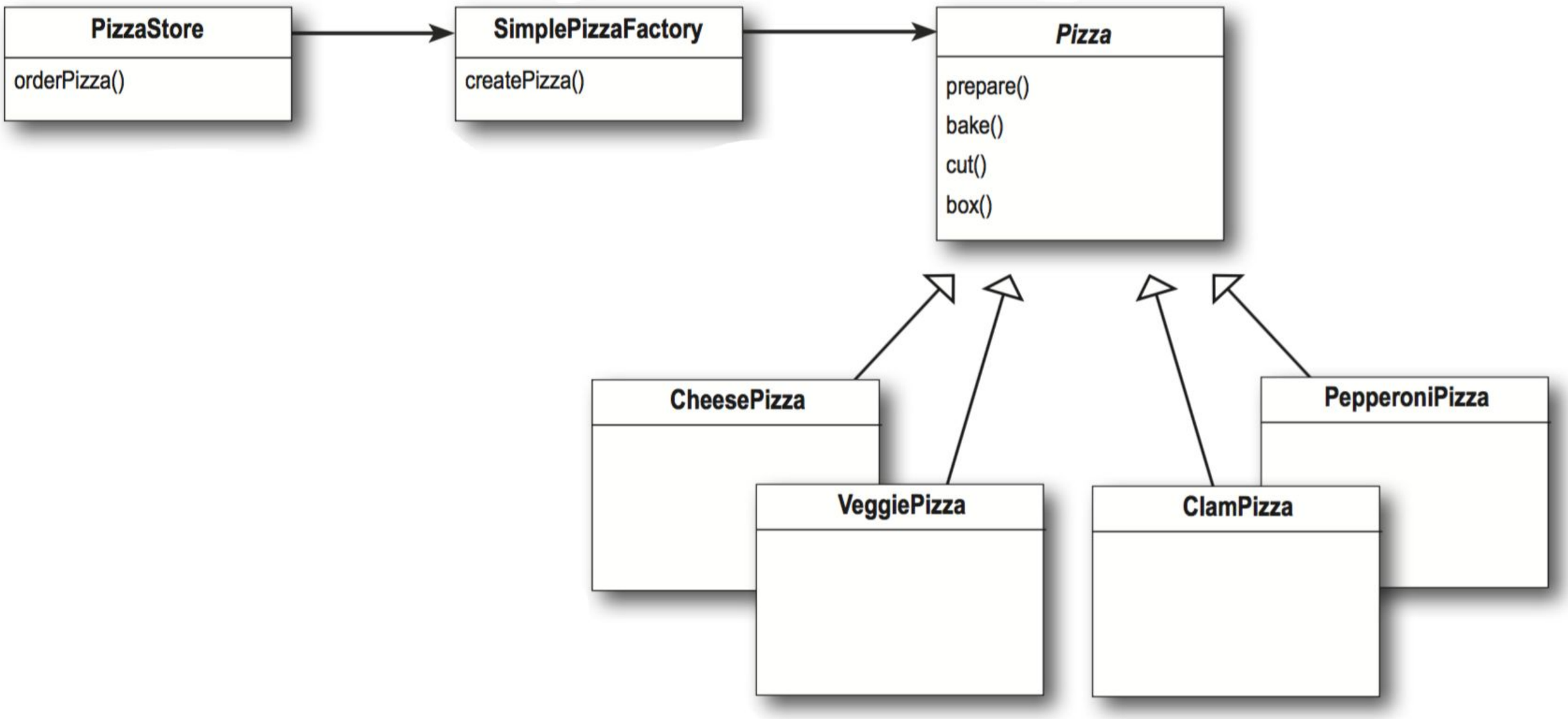
The Pizza Factory

```
public class SimplePizzaFactory {

    public Pizza createPizza(String type) {
        Pizza pizza = null;

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        return pizza;
    }
}
```

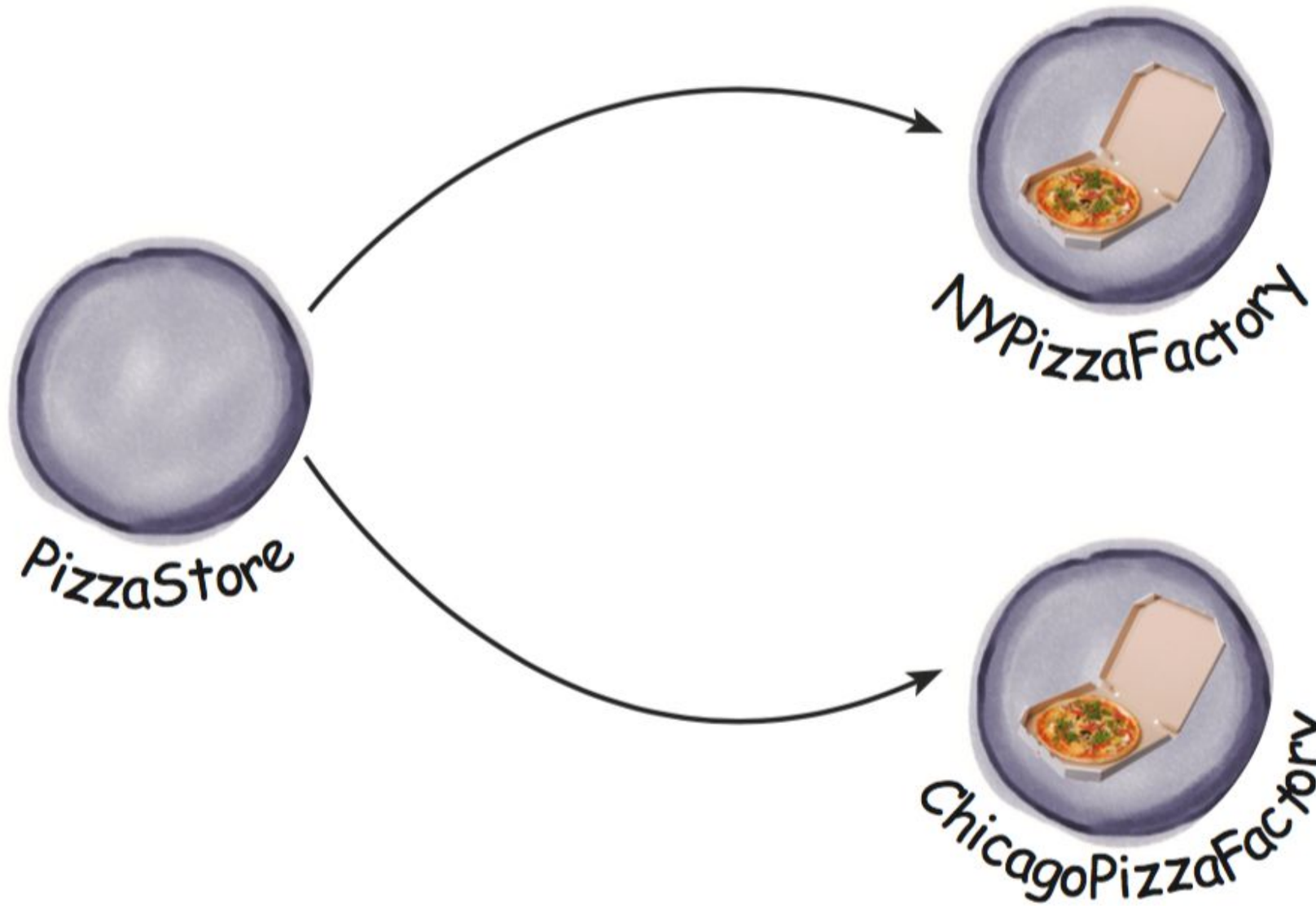
The New Pizza Store

```
public class PizzaStore {
    SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza;
        pizza = factory.createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

The Simple Factory “Pattern”



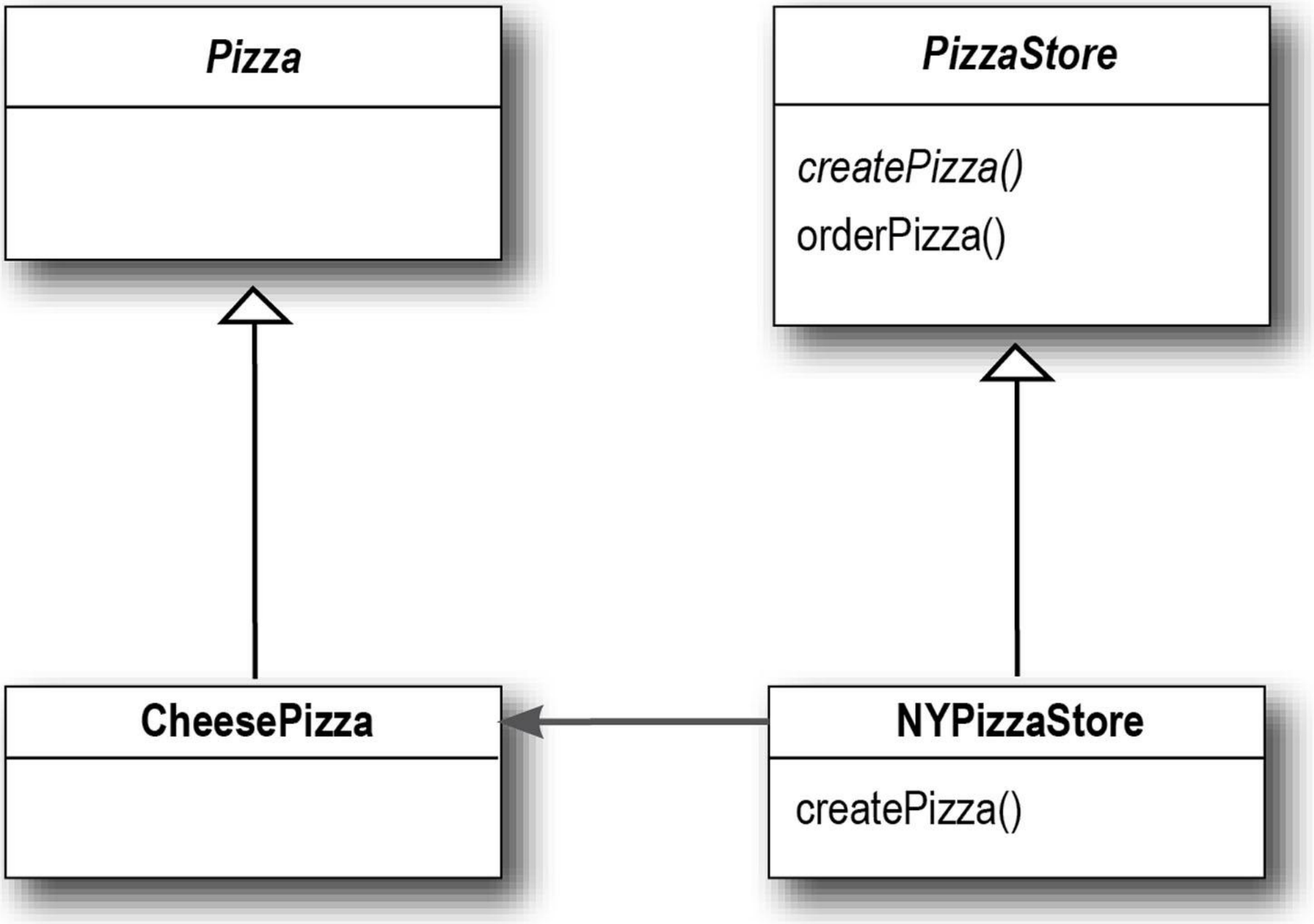
What if we want to franchise?



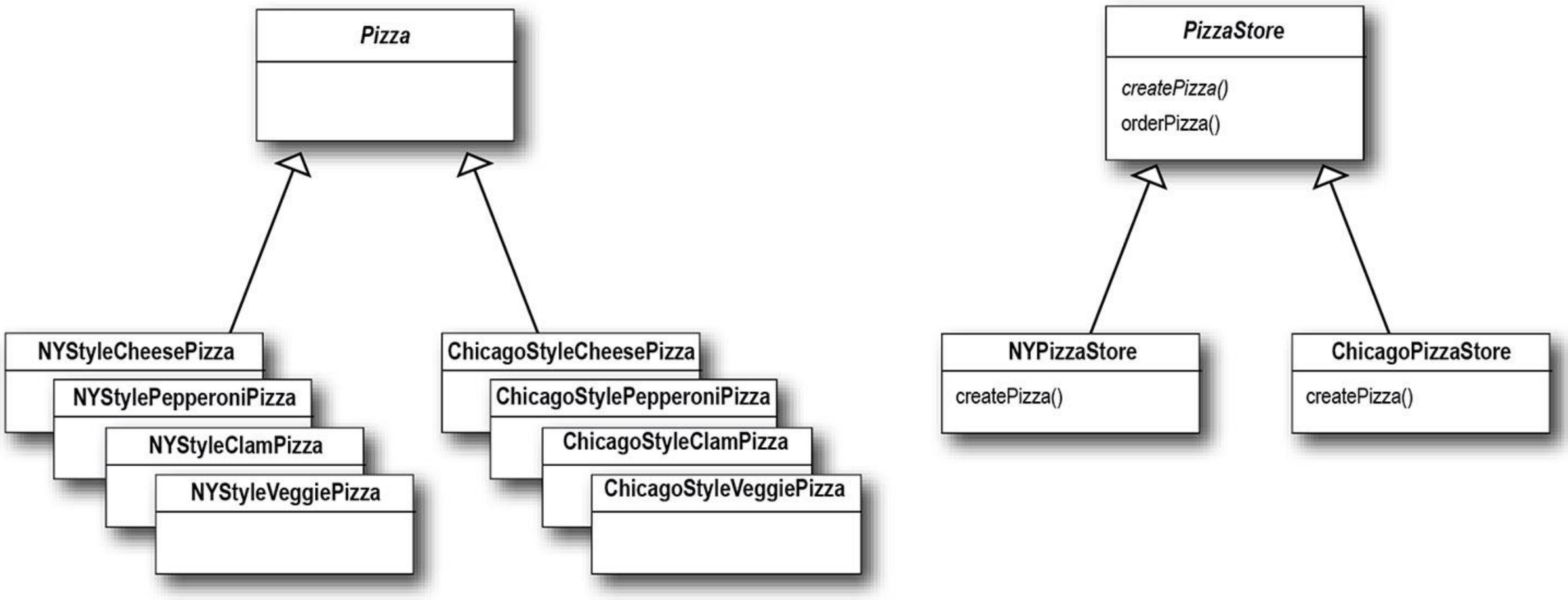
Factory Method Pattern

Factory Method Pattern: Define an interface for creating an object but let subclasses decide which class to instantiate. Factory Method lets a class delegate instantiation to the subclasses.

The Factory Pattern applied to Pizza



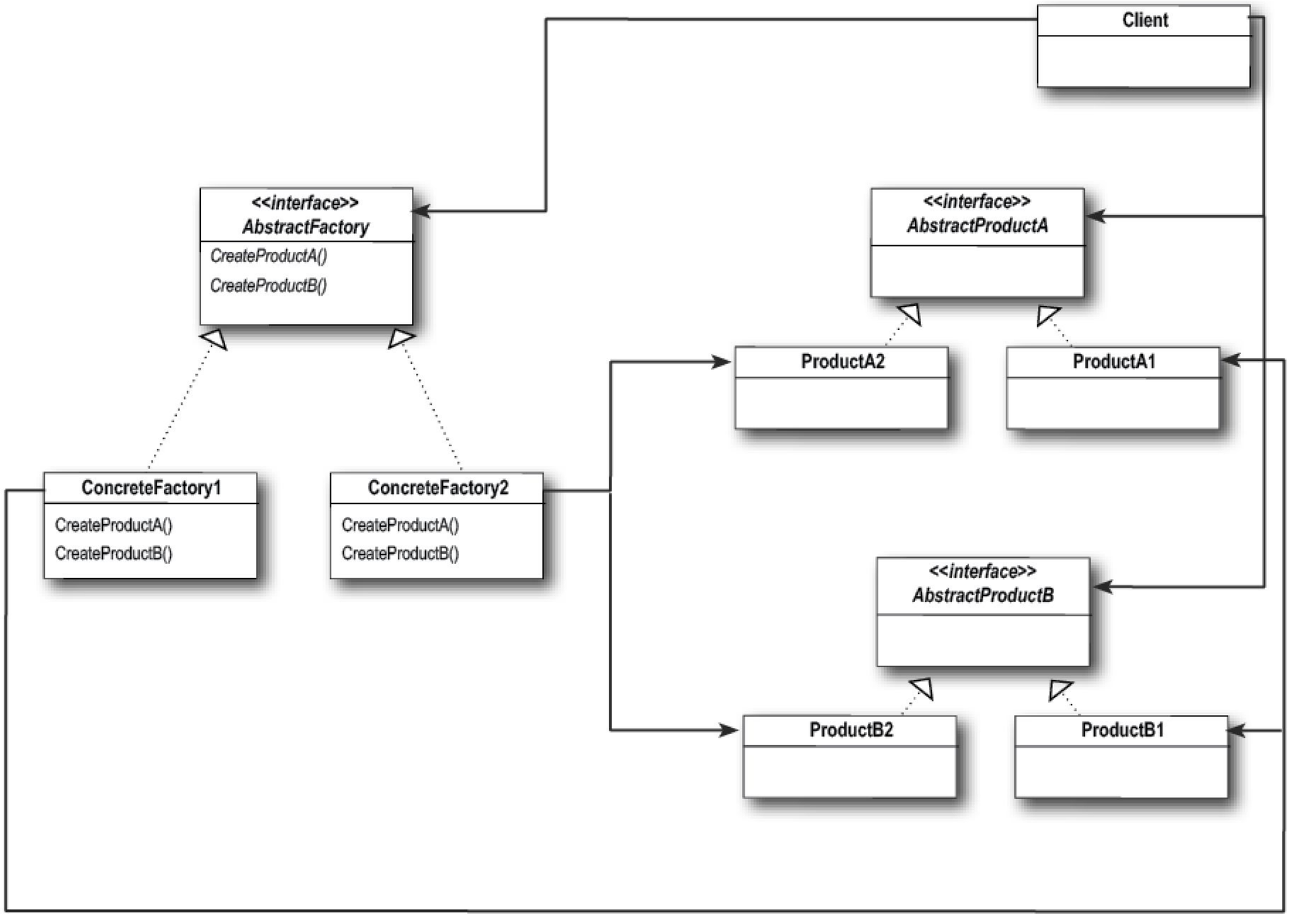
A Framework for Pizza Creation



Abstract Factory Pattern

Abstract Factory Pattern: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

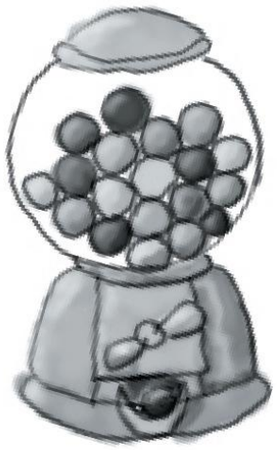
Abstract Factory's Structure

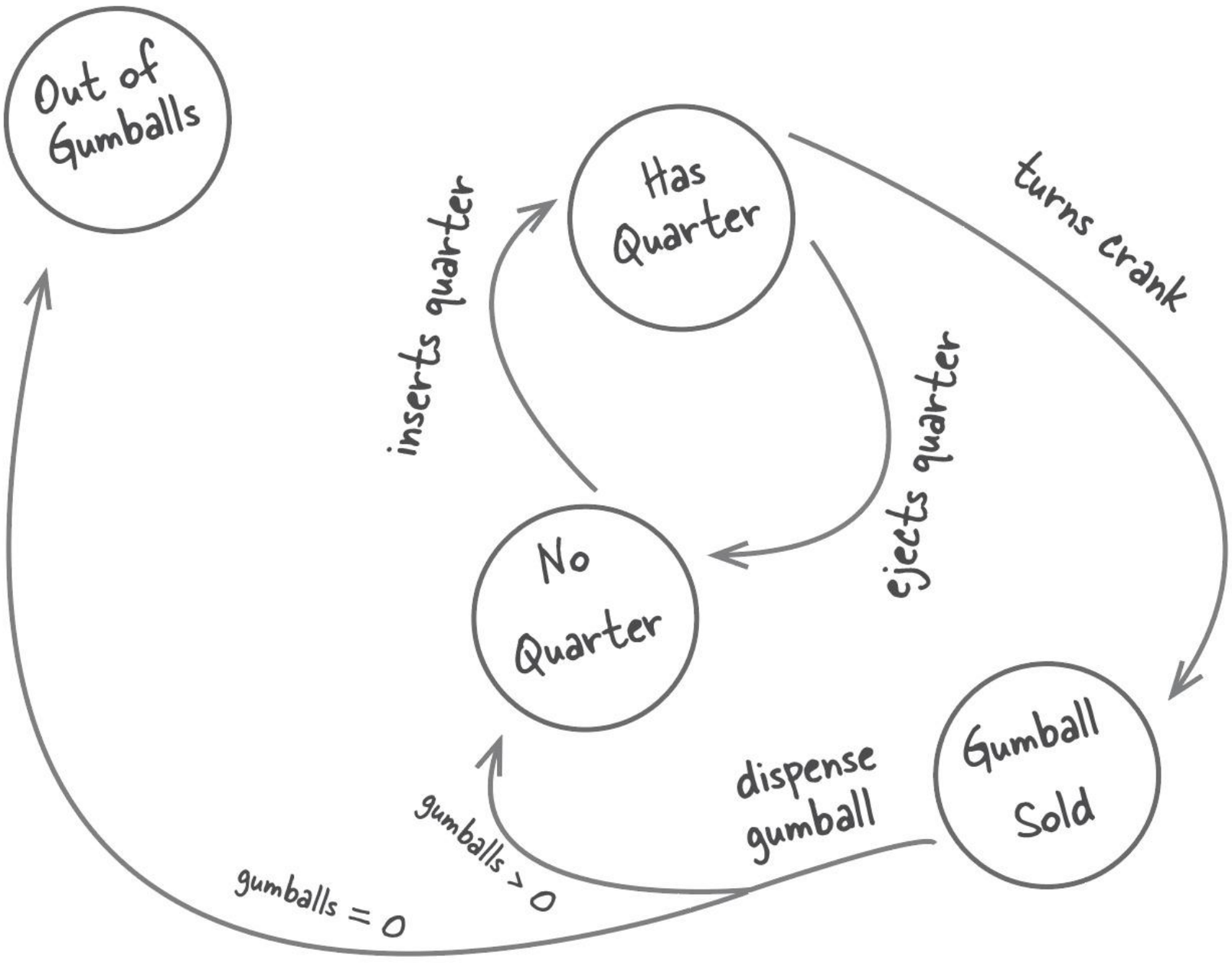


The State of Things

Design Patterns
BOOTCAMP

A State Diagram



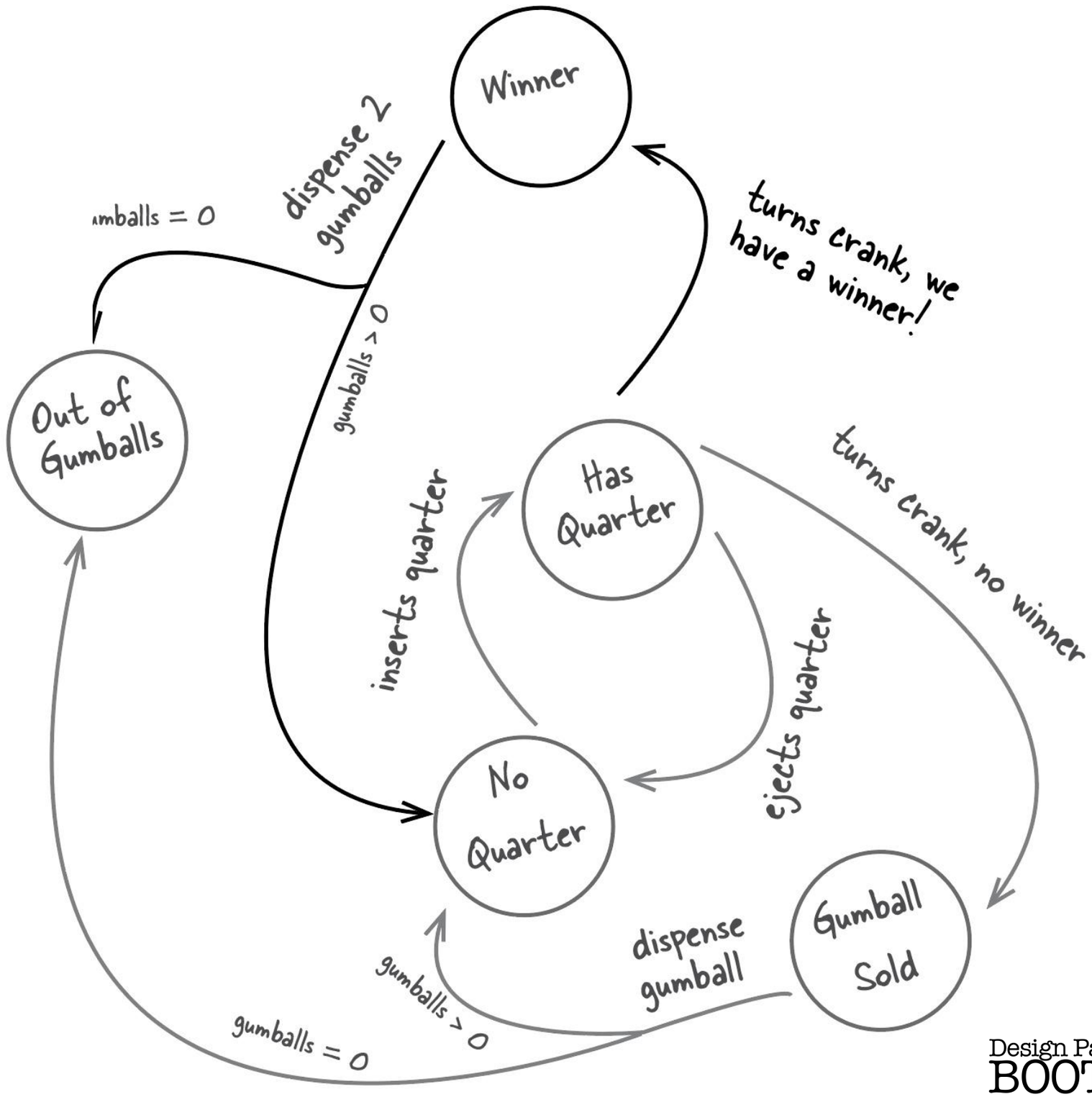


Typical Implementation

```
final static int SOLD_OUT = 0;
final static int NO_QUARTER = 1;
final static int HAS_QUARTER = 2;
final static int SOLD = 3;

int state = SOLD_OUT;

public void insertQuarter() {
    if (state == HAS_QUARTER) {
       System.out.println("You can't insert another quarter");
    } else if (state == NO_QUARTER) {
       state = HAS_QUARTER;
       System.out.println("You inserted a quarter");
    } else if (state == SOLD_OUT) {
       System.out.println("You can't insert a quarter, the machine is sold out");
    } else if (state == SOLD) {
       System.out.println("Please wait, we're already giving you a gumball");
    }
}
```

```
public void ejectQuarter() {
     if (state == HAS_QUARTER) {
          System.out.println("Quarter returned");
          state = NO_QUARTER;
     } else if (state == NO_QUARTER) {
          System.out.println("You haven't inserted a quarter");
     } else if (state == SOLD) {
          System.out.println("Sorry, you already turned the crank");
     } else if (state == SOLD_OUT) {
     System.out.println("You can't eject, you haven't inserted a quarter yet");
     }
}

public void turnCrank() {
     if (state == SOLD) {
          System.out.println("Turning twice doesn't get you another gumball!");
     } else if (state == NO_QUARTER) {
          System.out.println("You turned but there's no quarter");
     } else if (state == SOLD_OUT) {
          System.out.println("You turned, but there are no gumballs");
     } else if (state == HAS_QUARTER) {
          System.out.println("You turned...");
          state = SOLD;
          dispense();
     }
}

private void dispense() {
     if (state == SOLD) {
          System.out.println("A gumball comes rolling out the slot");
          count = count - 1;
          if (count == 0) {
               System.out.println("Oops, out of gumballs!");
               state = SOLD_OUT;
          } else {
               state = NO_QUARTER;
          }
     } else if (state == NO_QUARTER) {
          System.out.println("You need to pay first");
     } else if (state == SOLD_OUT) {
          System.out.println("No gumball dispensed");
     } else if (state == HAS_QUARTER) {
          System.out.println("No gumball dispensed");
     }
}
```

Feature request!



What has to change?

```
final static int SOLD_OUT = 0;
final static int NO_QUARTER = 1;
final static int HAS_QUARTER = 2;
final static int SOLD = 3;

int state = SOLD_OUT;

public void insertQuarter() {
     if (state == HAS_QUARTER) {
          System.out.println("You can't insert another quarter");
     } else if (state == NO_QUARTER) {
          state = HAS_QUARTER;
          System.out.println("You inserted a quarter");
     } else if (state == SOLD_OUT) {
          System.out.println("You can't insert a quarter, the
     machine is sold out");
     } else if (state == SOLD) {
          System.out.println("Please wait, we're already giving you
     a gumball");
     }
}

public void ejectQuarter() {
     if (state == HAS_QUARTER) {
          System.out.println("Quarter returned");
          state = NO_QUARTER;
     } else if (state == NO_QUARTER) {
          System.out.println("You haven't inserted a quarter");
     } else if (state == SOLD) {
          System.out.println("Sorry, you already turned the crank");
     } else if (state == SOLD_OUT) {
          System.out.println("You can't eject, you haven't inserted
a quarter yet");
     }
}
```

```
 public void turnCrank() {
     if (state == SOLD) {
          System.out.println("Turning twice doesn't get you another
gumball!");
     } else if (state == NO_QUARTER) {
          System.out.println("You turned but there's no quarter");
     } else if (state == SOLD_OUT) {
          System.out.println("You turned, but there are no
gumballs");
     } else if (state == HAS_QUARTER) {
          System.out.println("You turned...");
          state = SOLD;
          dispense();
     }
}

private void dispense() {
     if (state == SOLD) {
          System.out.println("A gumball comes rolling out the
slot");
          count = count - 1;
          if (count == 0) {
               System.out.println("Oops, out of gumballs!");
               state = SOLD_OUT;
          } else {
               state = NO_QUARTER;
          }
     } else if (state == NO_QUARTER) {
          System.out.println("You need to pay first");
     } else if (state == SOLD_OUT) {
          System.out.println("No gumball dispensed");
     } else if (state == HAS_QUARTER) {
          System.out.println("No gumball dispensed");
     }
}
```

Observations about the Code

A. This code isn’t adhering to the Open Closed Principle.

B. This code would make a FORTRAN programmer proud.

C. This design isn’t even very object oriented.

D. State transitions aren’t explicit; they are buried in the middle of a bunch of conditional statements.

E. We haven’t encapsulated anything that varies here.

F. Further additions are likely to cause bugs in working code.

STATE

Object Behavioral

Intent

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

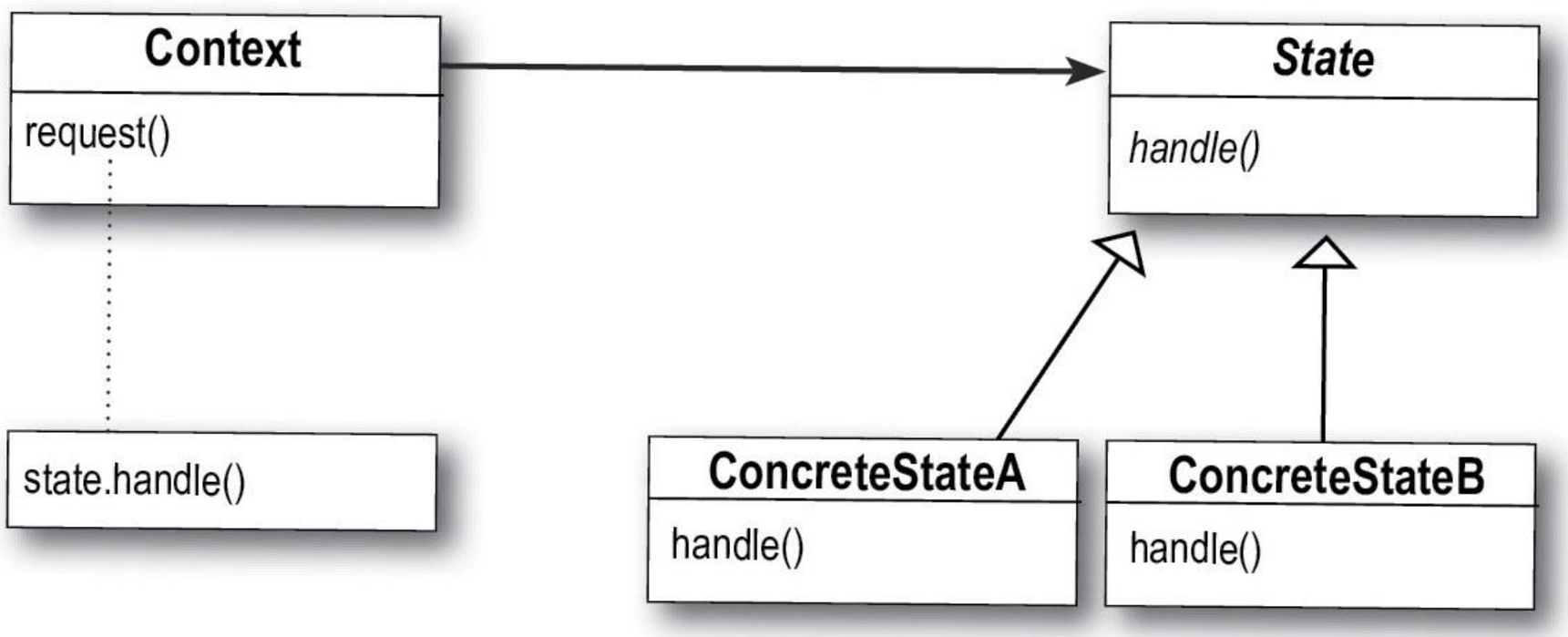
Applicability

Use the State pattern in either of the following cases:

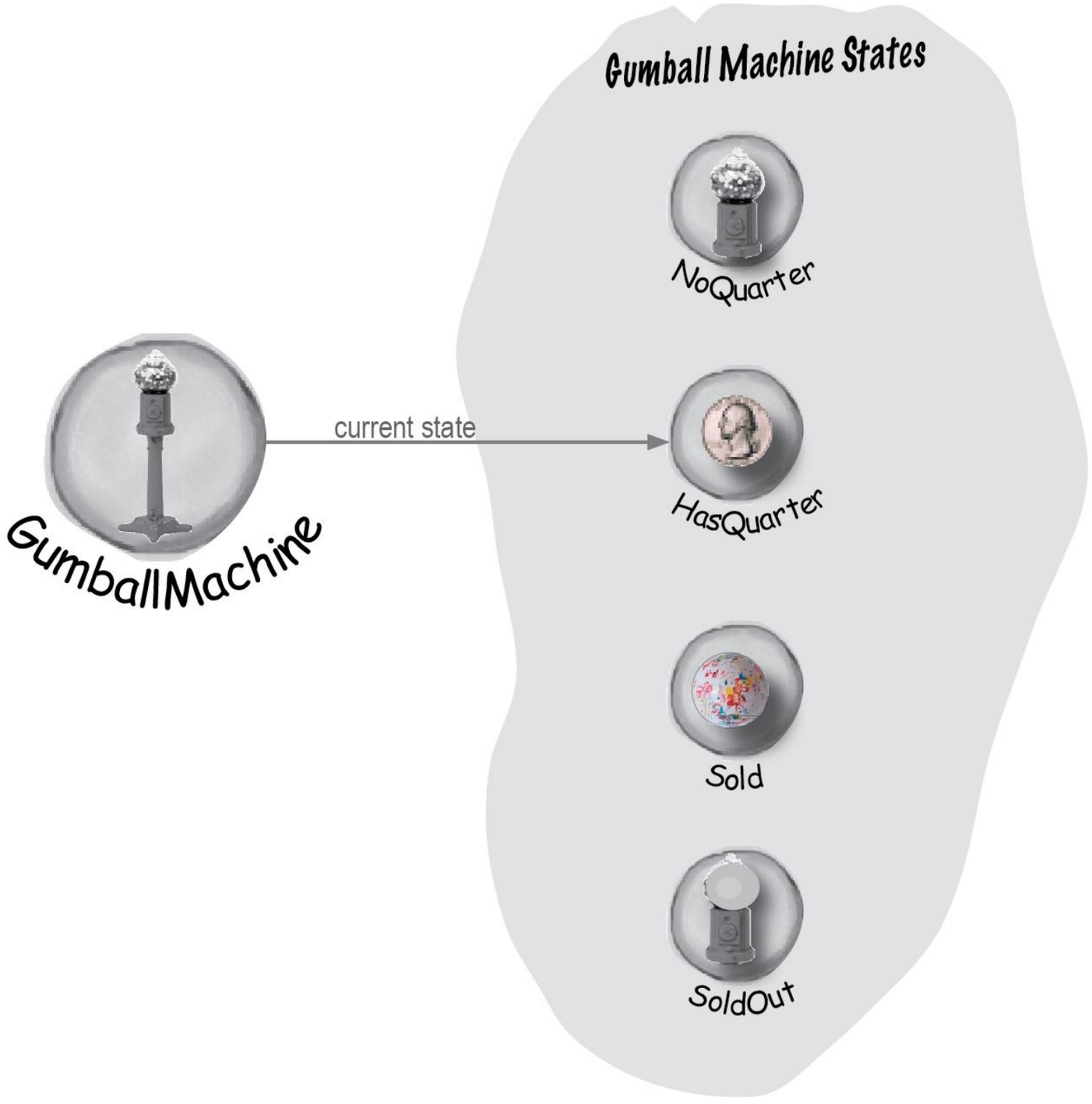
* An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.

* Operations have large, multipart conditional statements that depend on the object's state. This state is usually represented by one or more enumerated constants. Often, several operations will contain this same conditional structure. The State pattern puts each branch of the conditional in a separate class. This lets you treat the object's state as an object in its own right that can vary independently from other objects.

Structure



Big Picture


Gumball Machine States
NoQuarter
current state
HasQuarter
GumballMachine
Sold
SoldOut

```
public class GumballMachine {

    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER = 2;
    final static int SOLD = 3;

    int state = SOLD_OUT;
    int count = 0;
```

Old code

In the GumballMachine, we update the code to use the new classes rather than the static integers. The code is quite similar, except that in one class we have integers and in the other objects...

New code

```
public class GumballMachine {

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;

    State state = soldOutState;
    int count = 0;

}
```

The Gumball Machine Class

```
public class GumballMachine {

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;

    State state;
    int count = 0;

    public GumballMachine(int numberGumballs) {
        soldOutState = new SoldOutState(this);
        noQuarterState = new NoQuarterState(this);
        hasQuarterState = new HasQuarterState(this);
        soldState = new SoldState(this);

        this.count = numberGumballs;
        if (numberGumballs > 0) {
            state = noQuarterState;
        } else {
            state = soldOutState;
        }
    }
...
}
```

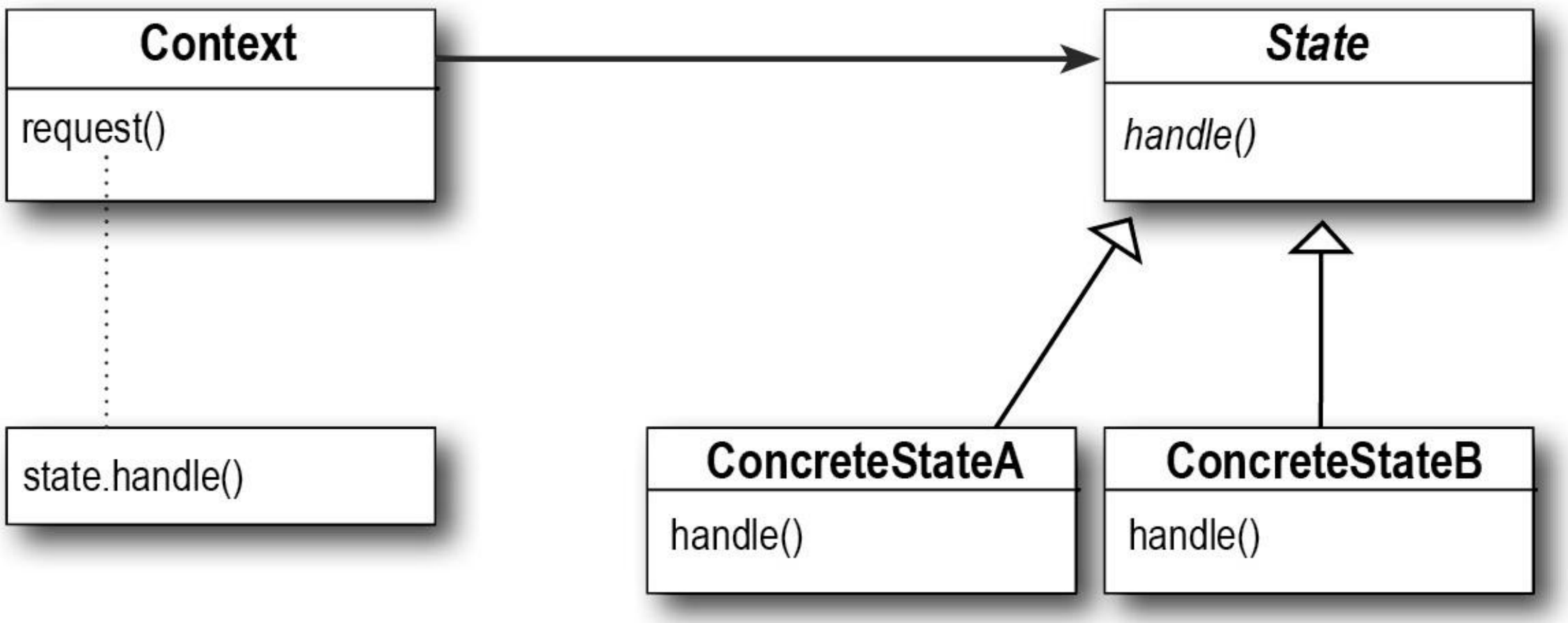
The HasQuarterState Class

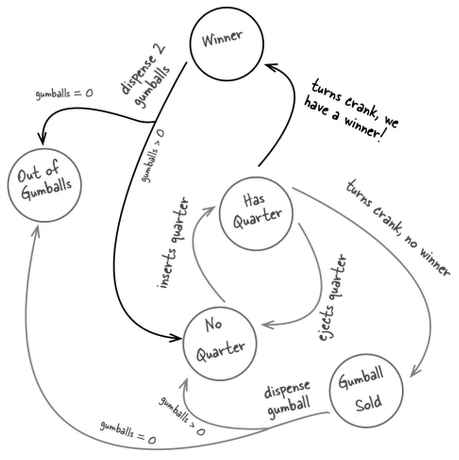
```
public class HasQuarterState implements State {
    GumballMachine gumballMachine;

    public HasQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You can't insert another quarter");
    }

    public void ejectQuarter() {
        System.out.println("Quarter returned");
        gumballMachine.setState(gumballMachine.getNoQuarterState());
    }

    public void turnCrank() {
        System.out.println("You turned...");
        gumballMachine.setState(gumballMachine.getSoldState());
    }

     public void dispense() {
         System.out.println("No gumball dispensed");
     }
}
```

The Gumball Machine Class

```
public void insertQuarter() {
   state.insertQuarter();
}

public void ejectQuarter() {
   state.ejectQuarter();
}

public void turnCrank() {
   state.turnCrank();
   state.dispense();
}

void releaseBall() {
   System.out.println("A gumball comes rolling out the slot...");
   if (count != 0) {
      count = count - 1;
   }
}

…

void setState(State state) {
   this.state = state;
}
```

The State Pattern





Can we easily add the winning state now?

```
public class WinnerState implements State {
   public void dispense() {
      // logic to dispense two gumballs if the
      // customer is a winner
   }
   …
}

public class HasQuarterState implements State {
   public void turnCrank() {
      // logic to randomly pick a winner and
      // transition to Winner state
      // otherwise just go to Sold state
   }
   …
}
```

Did we solve these issues?

A. This code isn't adhering to the Open Closed Principle.

B. This code would make a FORTRAN programmer proud.

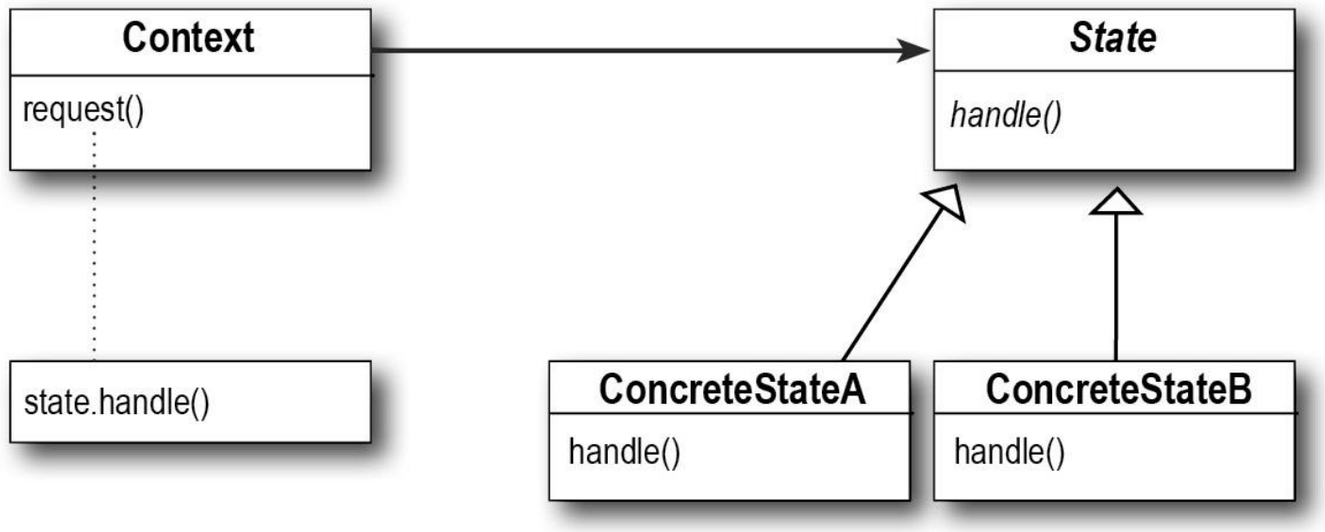
C. This design isn't even very object oriented.

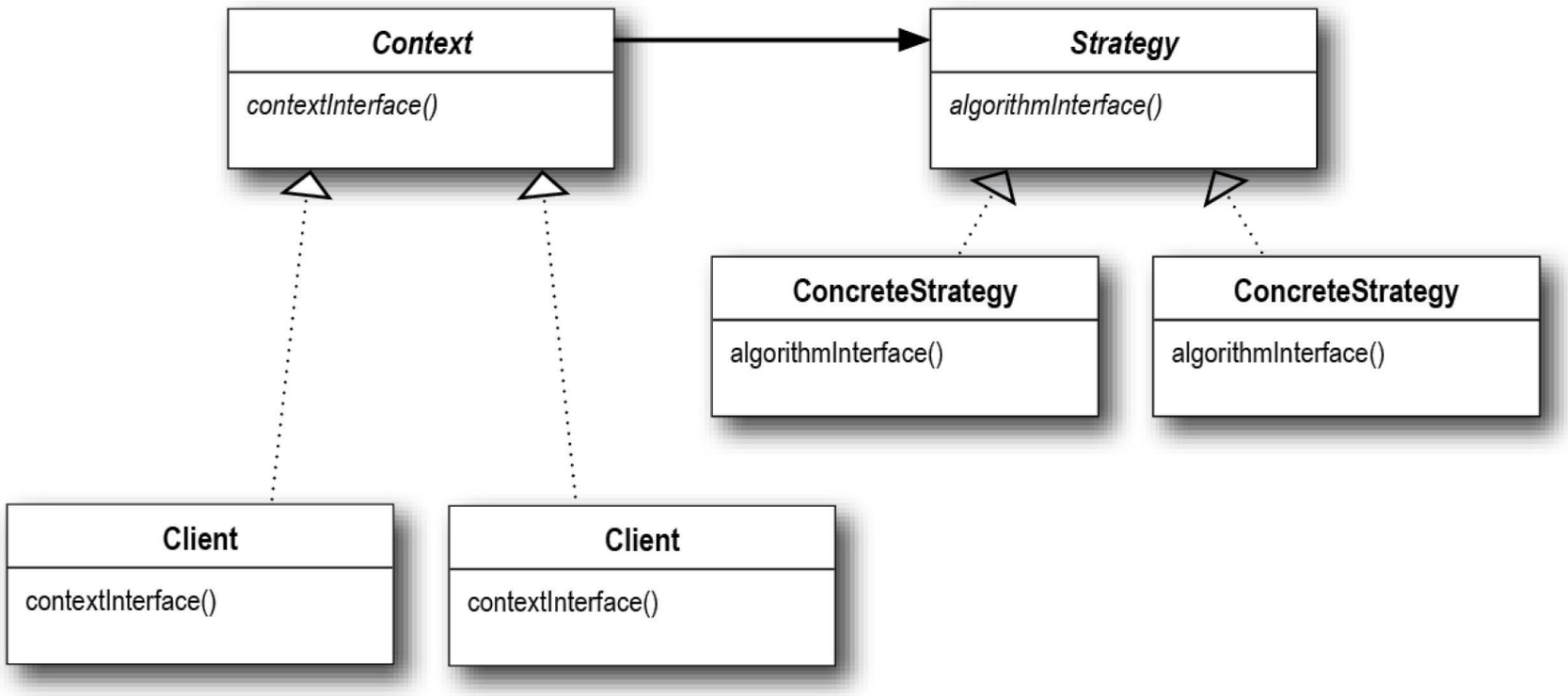
D. State transitions aren't explicit; they are buried in the middle of a bunch of conditional statements.

E. We haven't encapsulated anything that varies here.

F. Further additions are likely to cause bugs in working code.

Wait, does State look familiar?



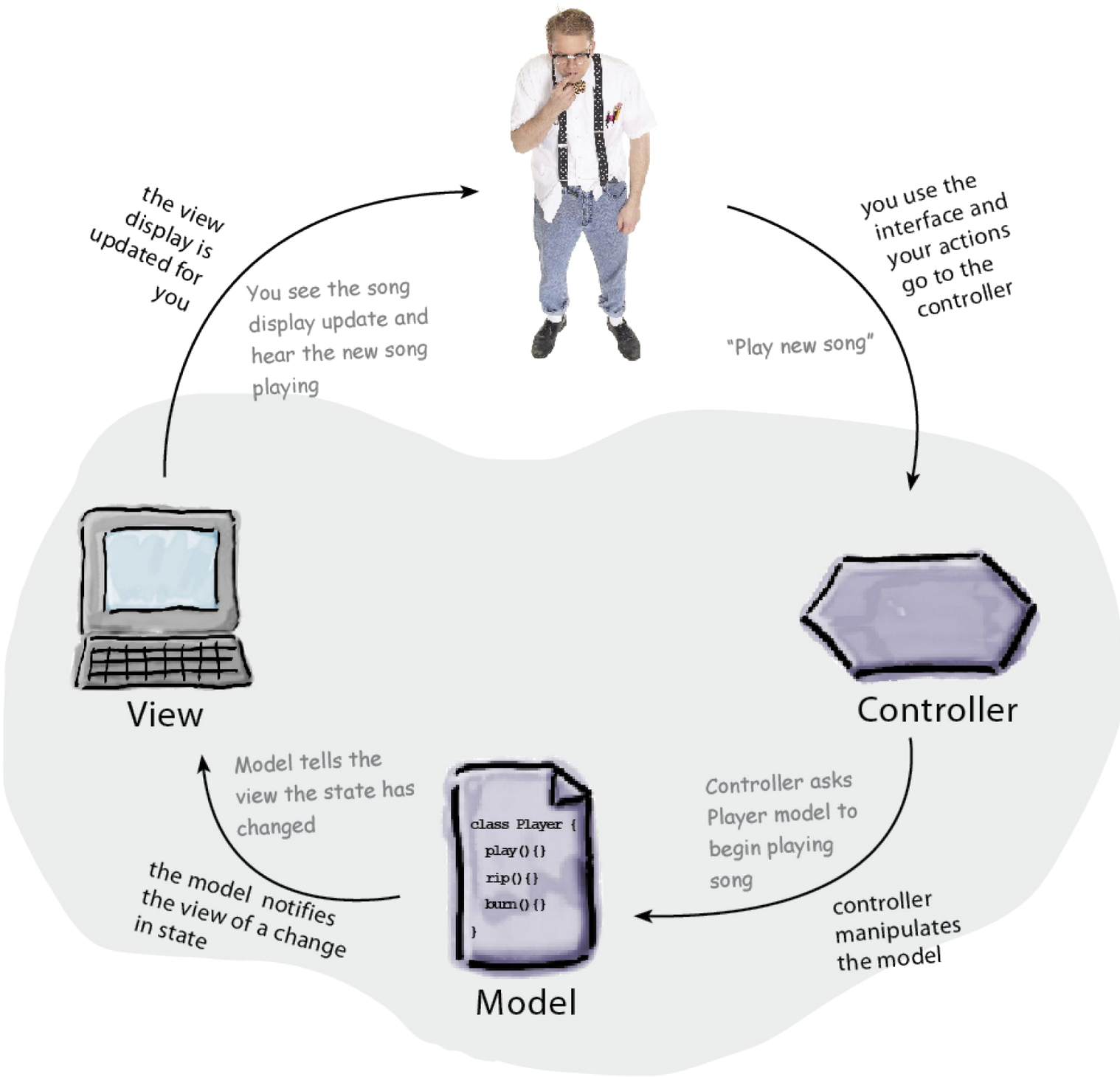


Compound Patterns

Patterns are often used together and combined within the same design solution.

A **compound pattern** combines two or more patterns into a solution that solves a recurring or general problem.

A common problem



Writing a Music Player Controller

CONTROLLER

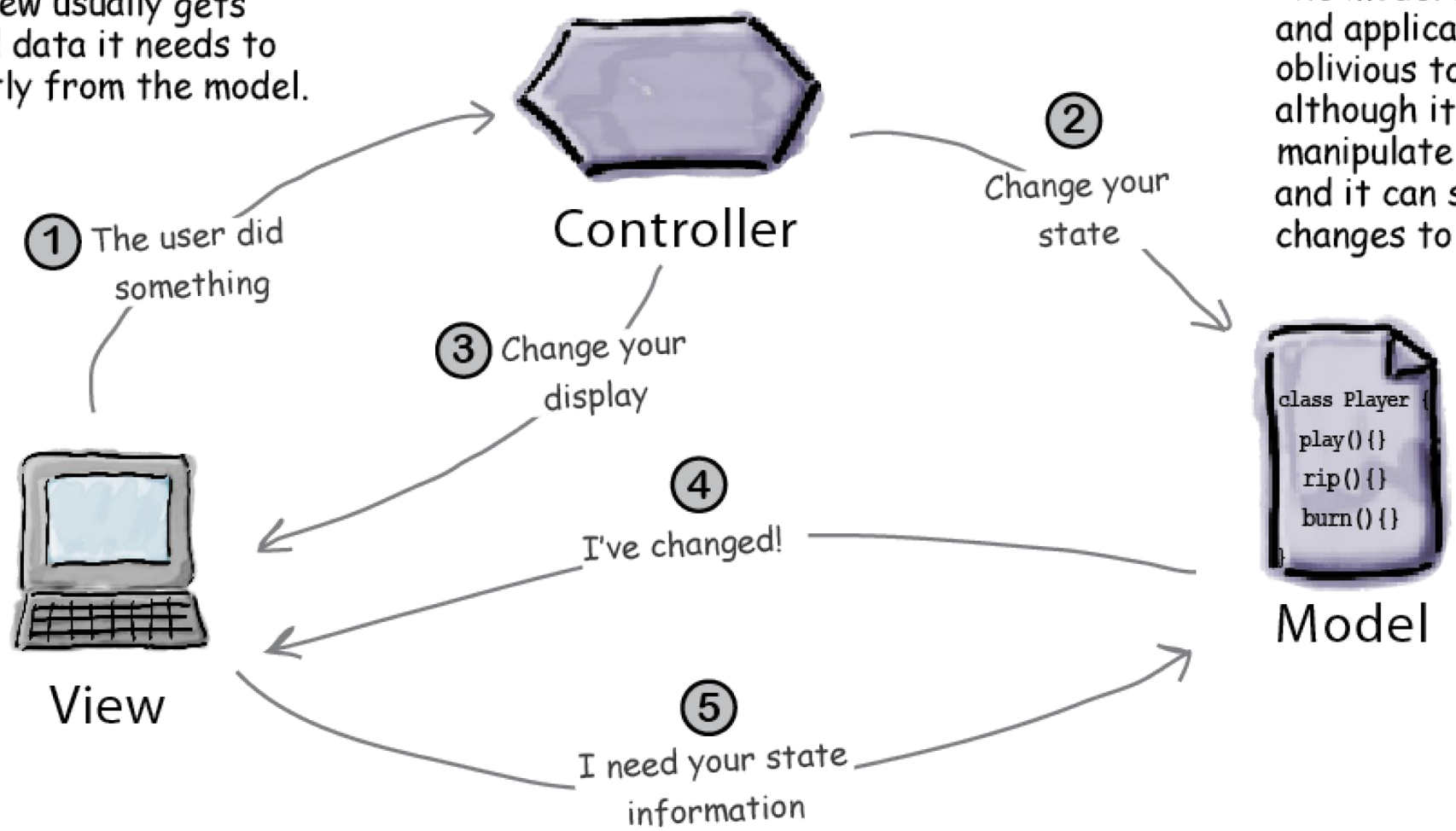
Takes user input and figures out what it means to the model.

VIEW

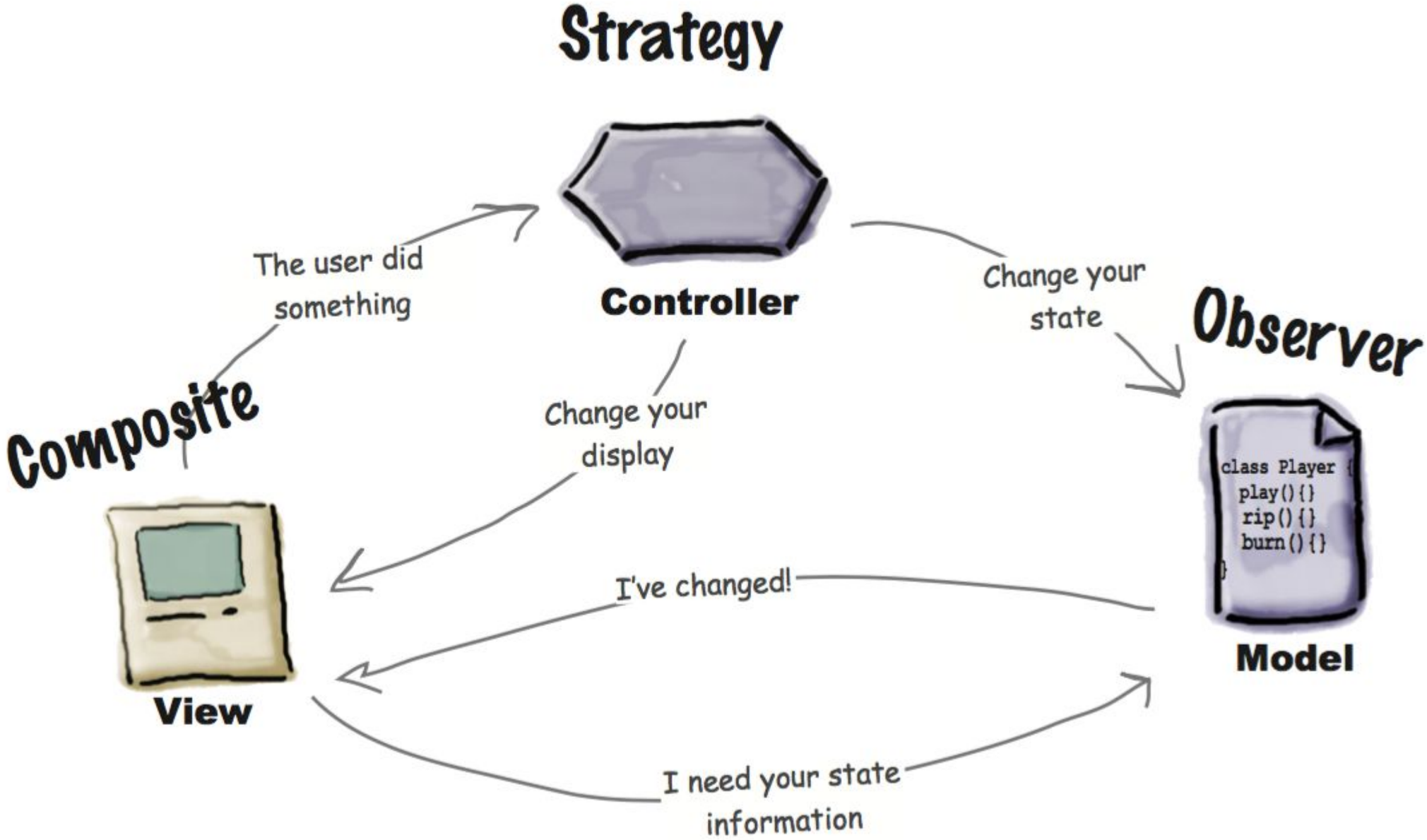
Gives you a presentation of the model. The view usually gets the state and data it needs to display directly from the model.

MODEL

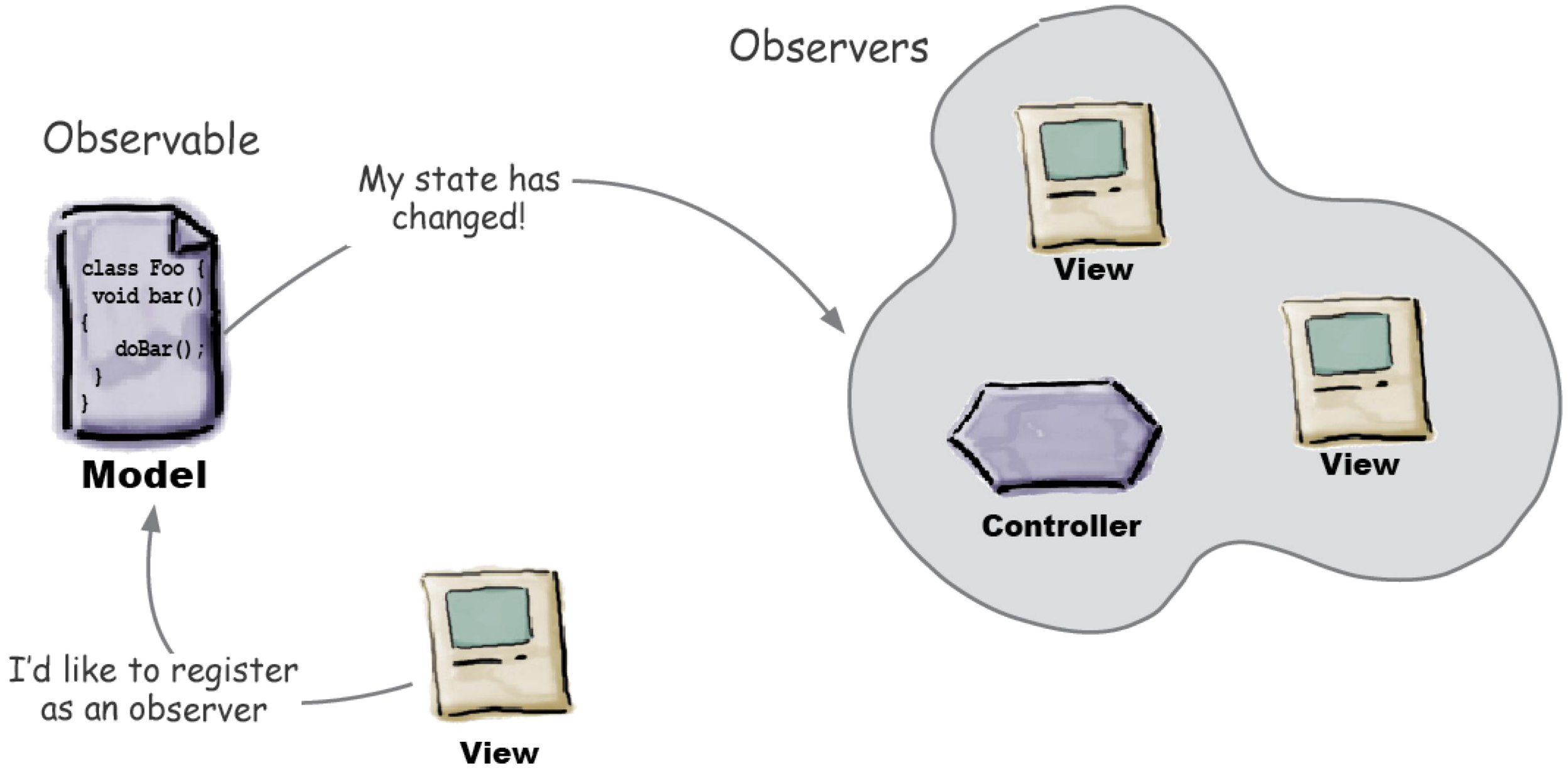
The model holds all the data, state and application logic. The model is oblivious to the view and controller, although it provides an interface to manipulate and retrieve its state and it can send notifications of state changes to observers.



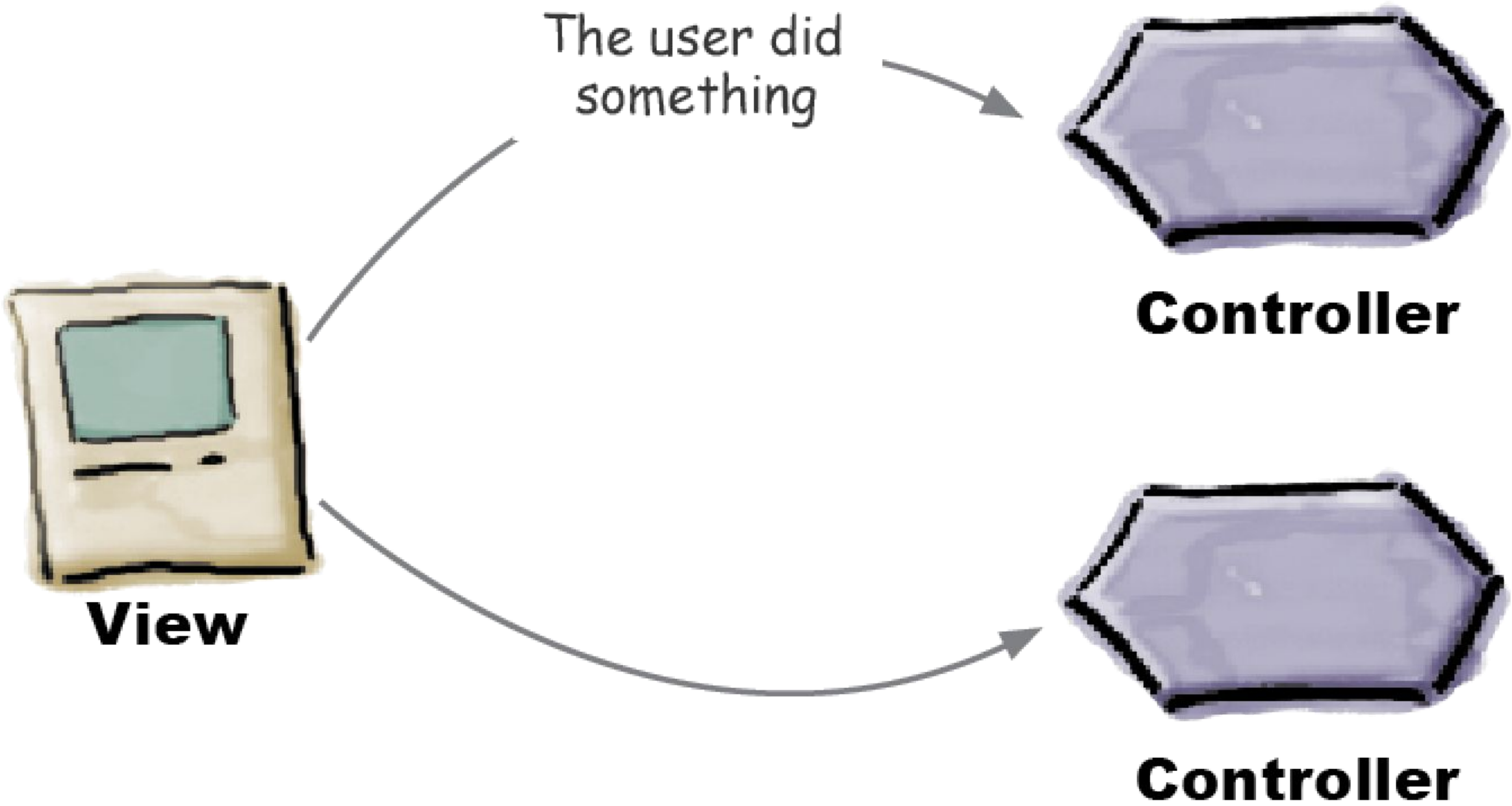
A Pattern made of Patterns



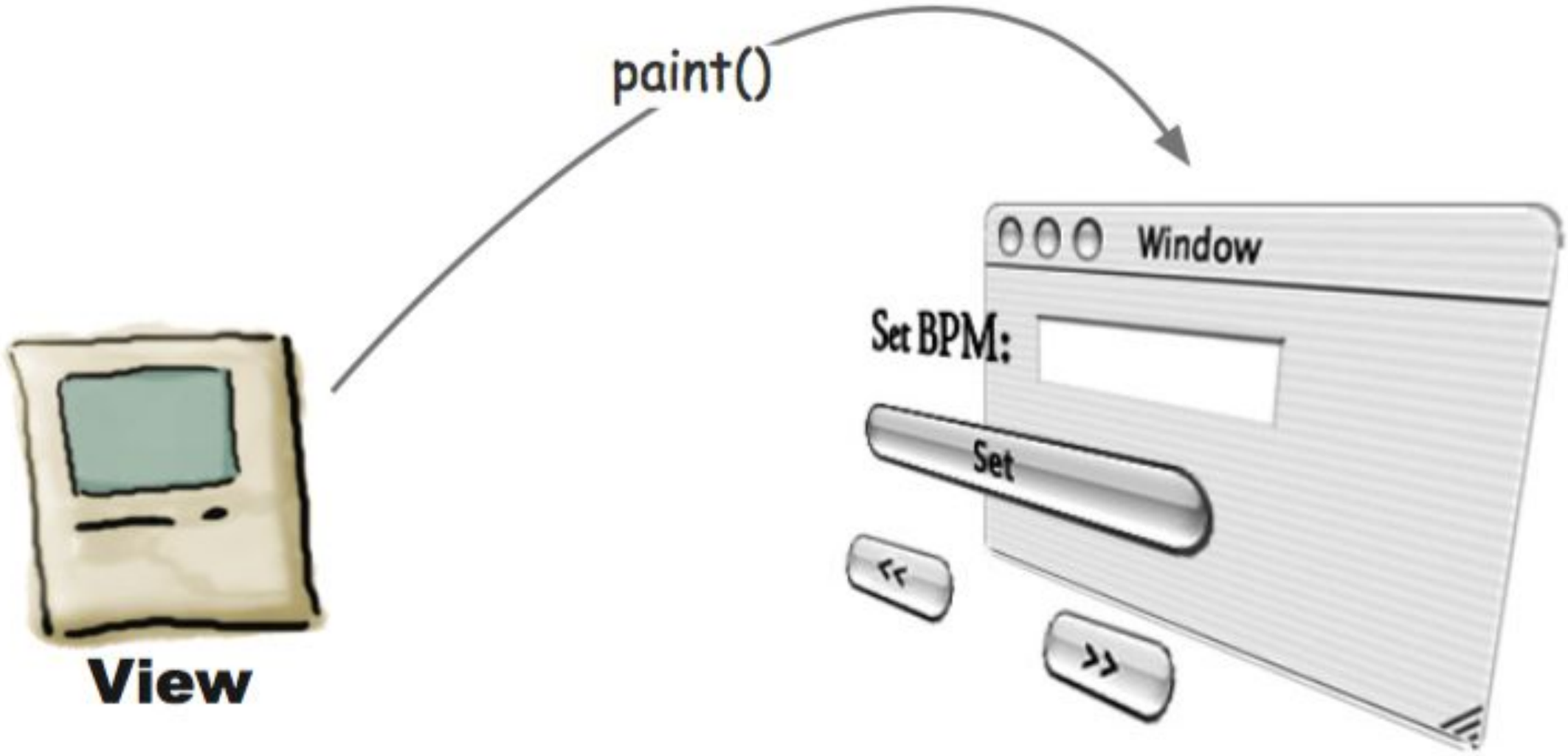
Observer in MVC



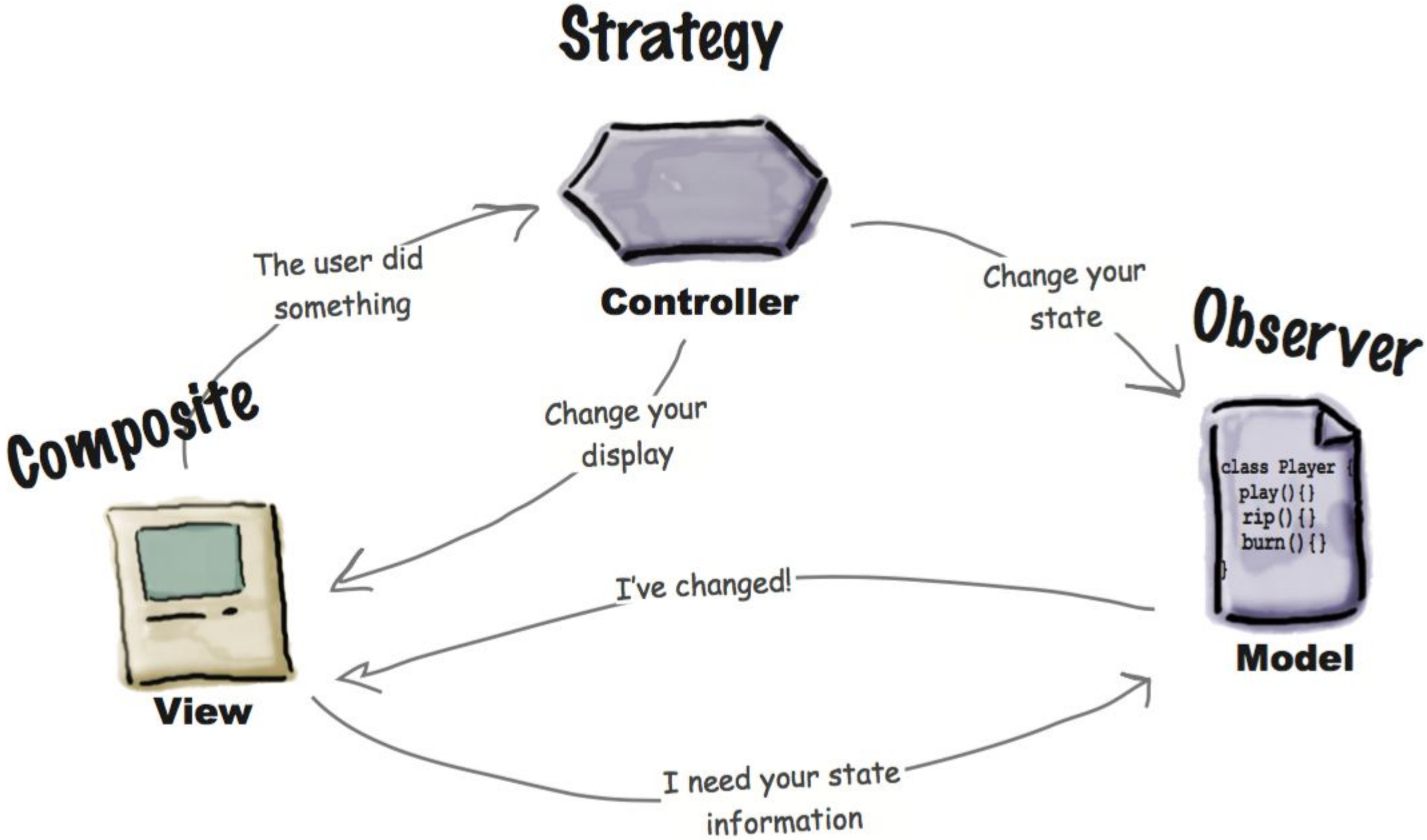
Strategy in MVC



Composite in MVC

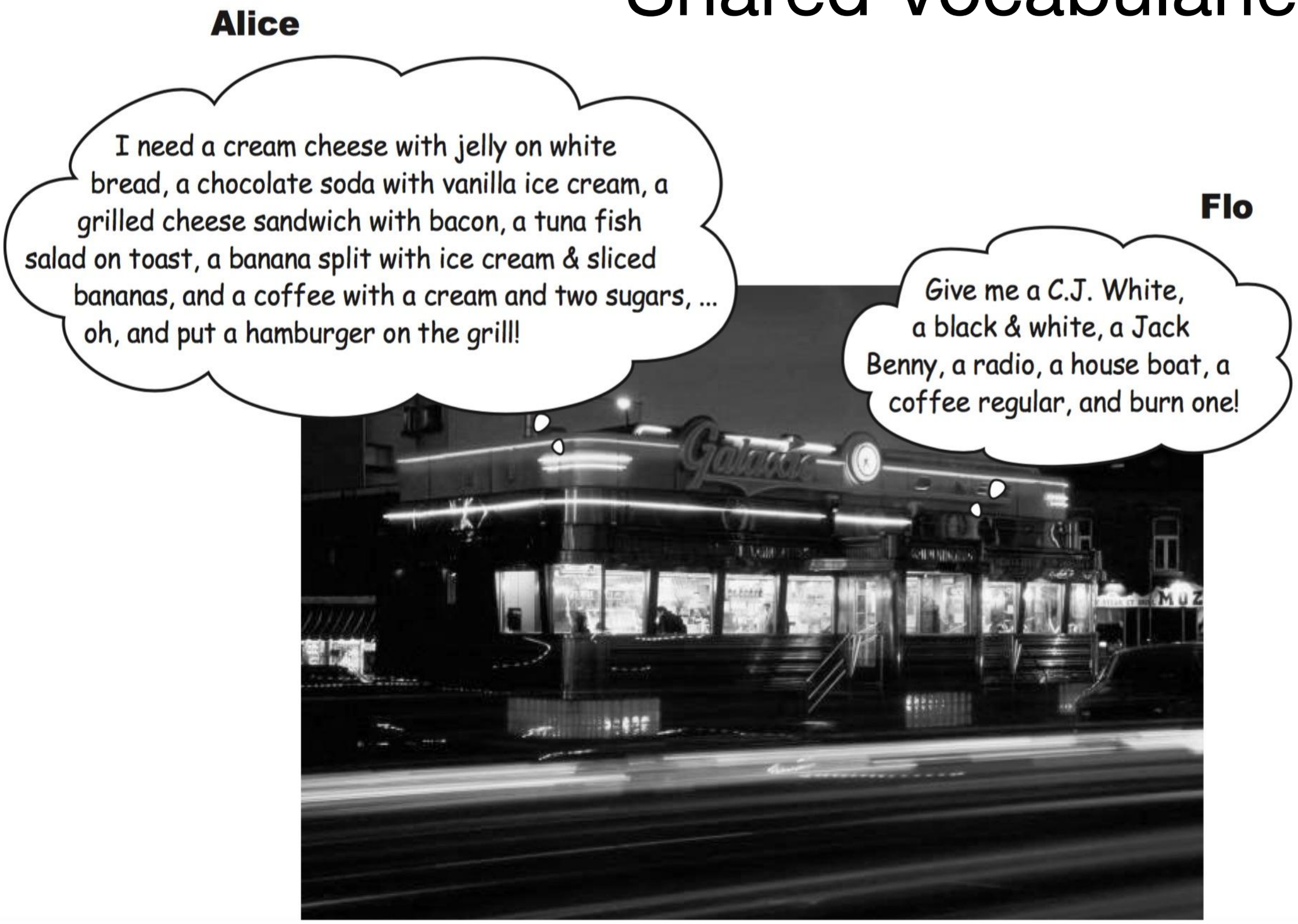


A Pattern made of Patterns



Your journey has just begun...

Shared Vocabularies



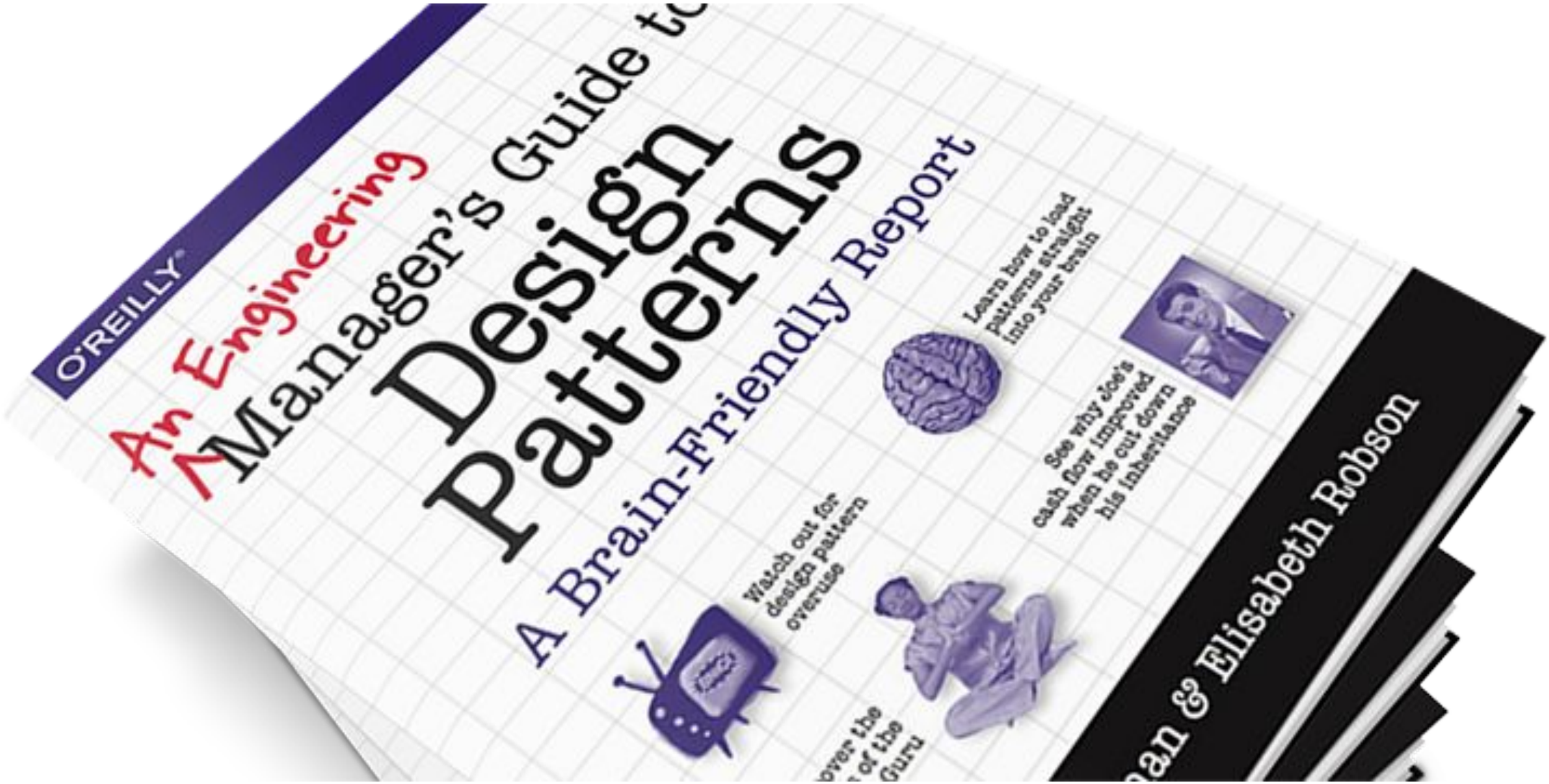
So I created this broadcast class. It keeps track of all the objects listening to it, and anytime a new piece of data comes along it sends a message to each listener. What's cool is that the listeners can join the broadcast at any time or they can even remove themselves. It is really dynamic and loosely coupled!

Rick, why didn't you just say you are using the **Observer Pattern**?

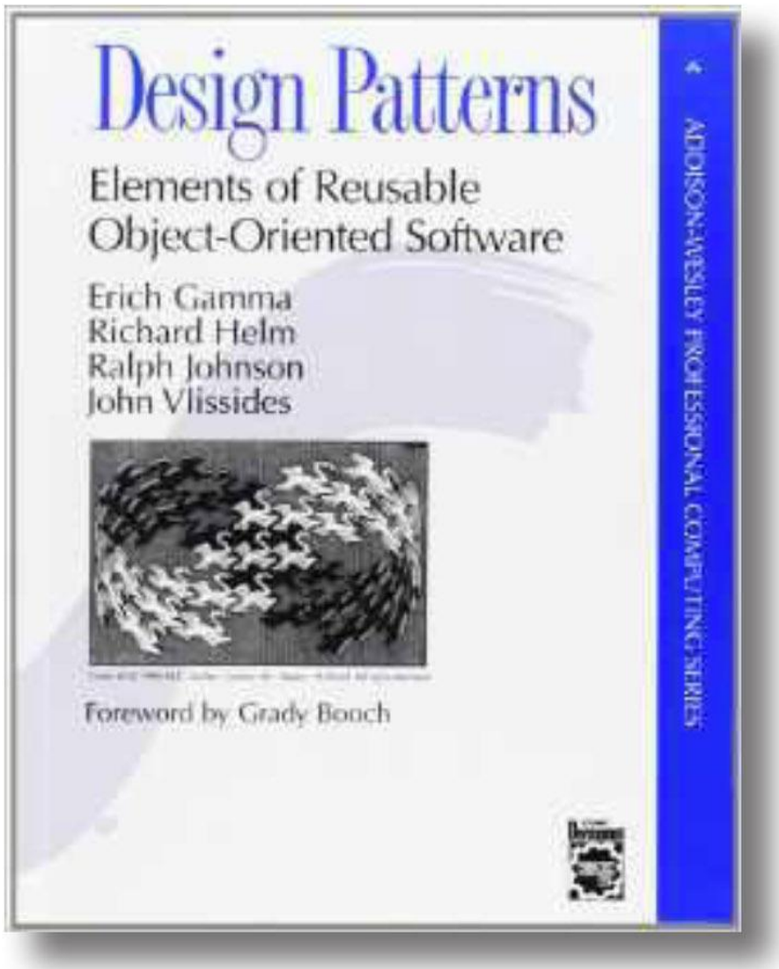
The Power of a Shared Vocabulary

- Shared pattern vocabularies are POWERFUL.
- Patterns allow you to say more with less.
- Talking at the pattern level allows you to stay “in the design” longer.
- Shared vocabularies can turbo-charge your development team.
- Shared vocabularies encourage more junior developers to get up to speed.

Engineering Manager's Guide



Where to go next

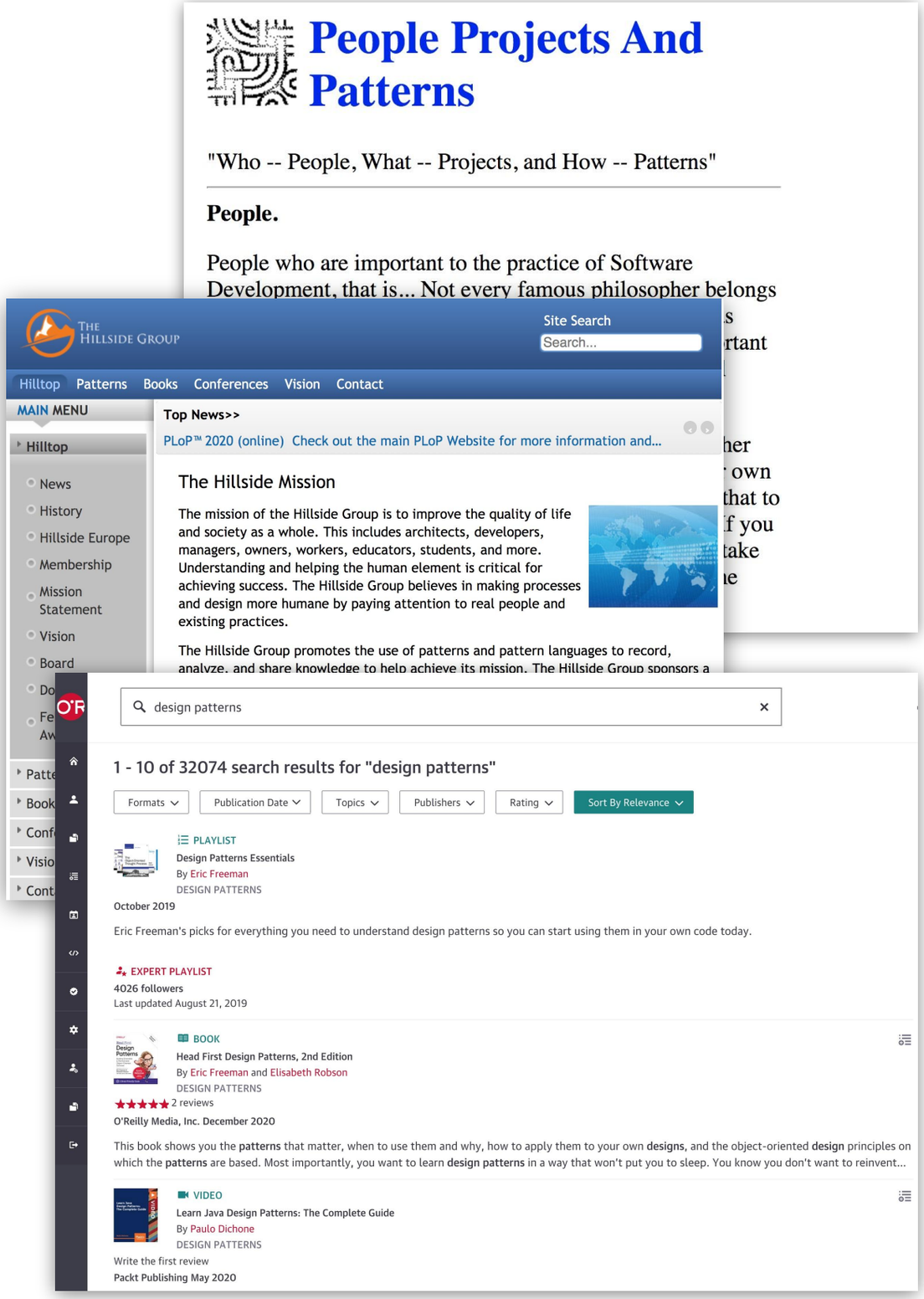


Peruse the definitive Gang of Four book

The authors of Design Patterns are affectionately known as the "Gang of Four," or GoF for short.

Really Learn this Stuff

Community



Websites
The Portland Patterns Repository, is run by Ward Cunningham, is a wiki devoted to all things related to patterns.
http://wiki.c2.com/?PeopleProjectsAndPatterns

The Hillside Group includes information, articles, books, and tools related to software design, including design patterns.
https://hillside.net

O'Reilly Learning has a wide collection of books, courses, and online training about design patterns.
https://learning.oreilly.com

Thinking in Patterns

- Design Patterns are not a magic bullet.
- You know you need a pattern when…
- Don’t be afraid to remove a pattern from your design.
- If you don’t need it now, don’t do it now.

Head First
Design Patterns

WARNING: Overuse of design patterns can lead to code that is downright over-engineered. Always go with the simplest solution that does the job and introduce patterns where the need emerges.

Anti-Patterns, What NOT to Do

Name: Golden Hammer

Problem: You need to choose technologies for your development and you believe that exactly one technology must dominate the architecture.

Context: You need to develop some new system or piece of software that doesn't fit well with the technology that the development team is familiar with.

Forces:

- The development team is committed to the technology they know.
- The development team is not familiar with other technologies.

Anti-Patterns, What NOT to Do

risky.

- It is easy to plan and estimate for development using the familiar technology.

Supposed Solution: Use the familiar technology anyway. The technology is applied obsessively to many problems, including places where it is clearly inappropriate.

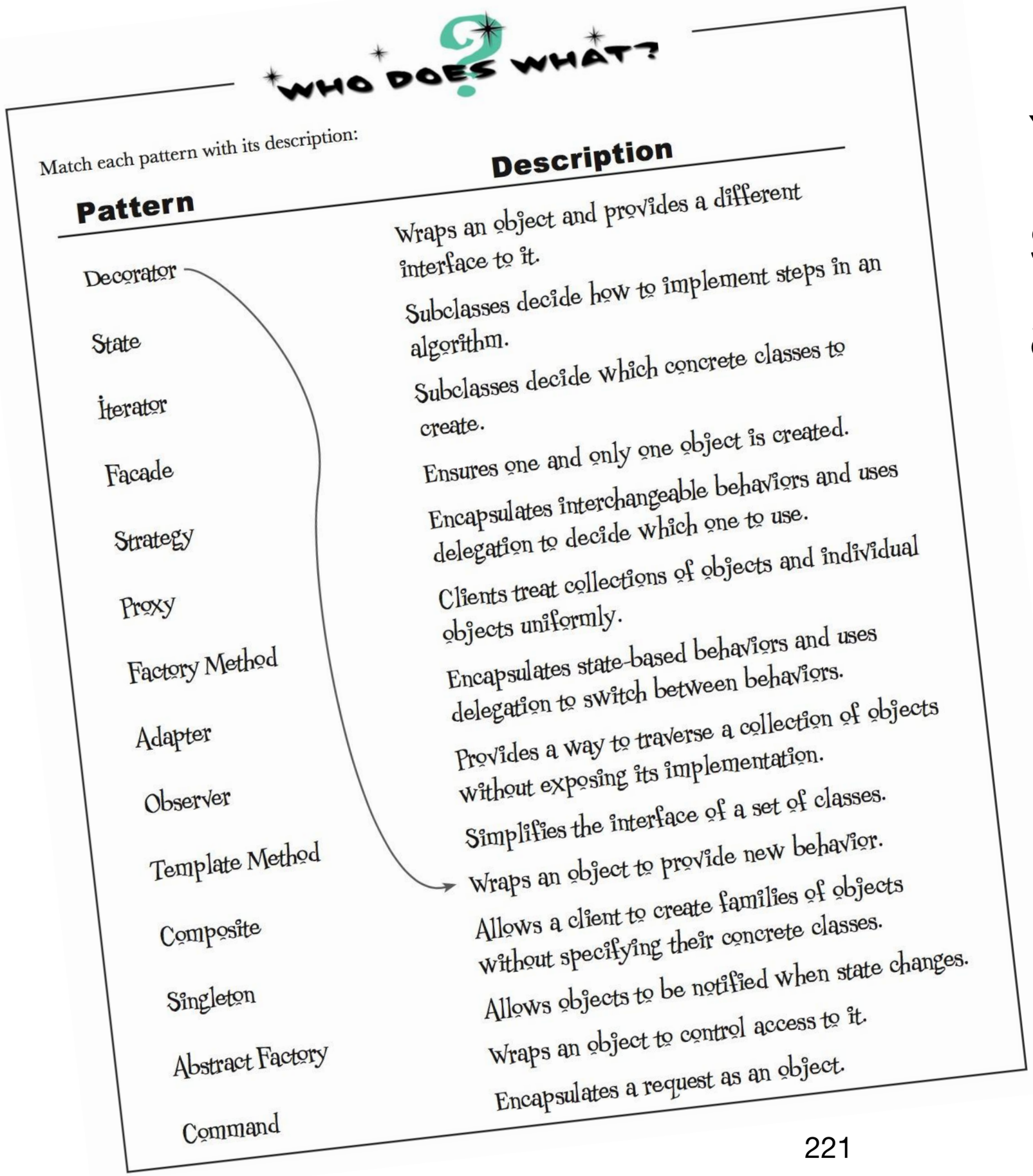
Refactored Solution: Expanding the knowledge of developers through education, training, and book study groups that expose developers to new solutions.

Examples:

Web companies keep using and maintaining their internal homegrown caching systems when open source alternatives are in use.

Can I write my own Patterns?

- Remember patterns are more discovered than created.
- Most people don’t author patterns, they just use them.
- If you do come across what you think is a new pattern, document it.
- Apply the rule of three: it’s not a pattern until it’s been applied in three different contexts.



Your challenge, should you accept it...

Boy, it’s been great having you.

We’re going to miss you, for sure. But don’t worry—you can always find us at **wickedlysmart.com** or @erictfree, @elisabethrobson on Twitter.