10 DESIGN PATTERNS

course “software requirements and architecture”

Oct 2021

JM Fernandes

contents

- patterns
- strategy pattern
- observer pattern

patterns

'someone has already solved your problems'

- Patterns constitute proved solutions to known and common problems.
- Patterns emerged from experience from other developers solving design problems.
- The best way to use patterns is to load your brain with them and then recognise place in your designs where you can apply them.
- Instead of code reuse, patterns allow **experience reuse**.

patterns

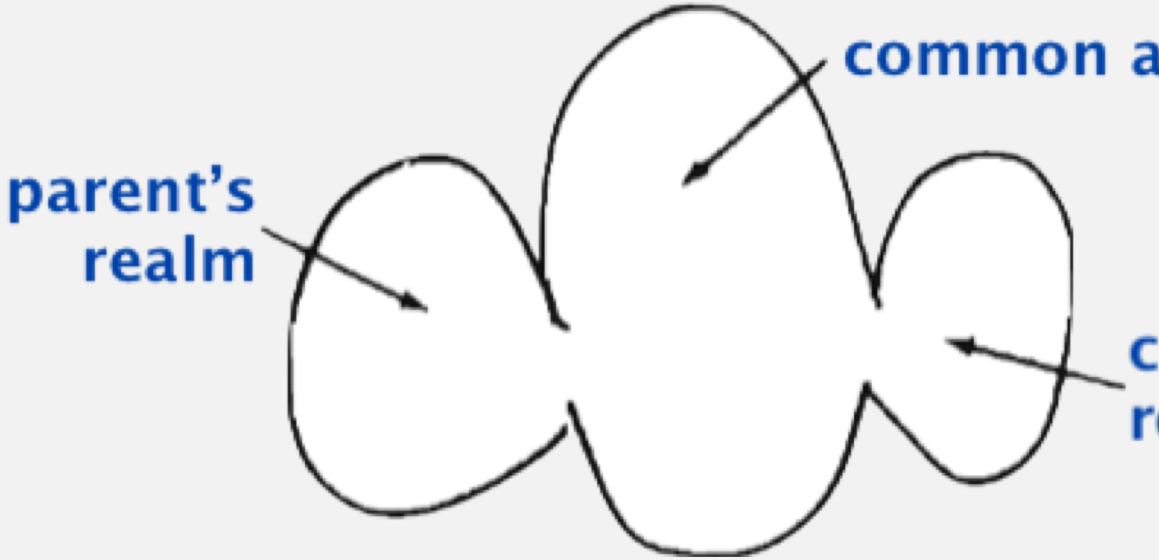
- Patterns have their origin in the work of Alexander.
- He collected generic solutions to solve recurring problems in the architecture domain.

> Each pattern describes a problem that 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
> (Alexander, 1977)

- A pattern is a textual description of a generic solution for a recurring problem in a given context.

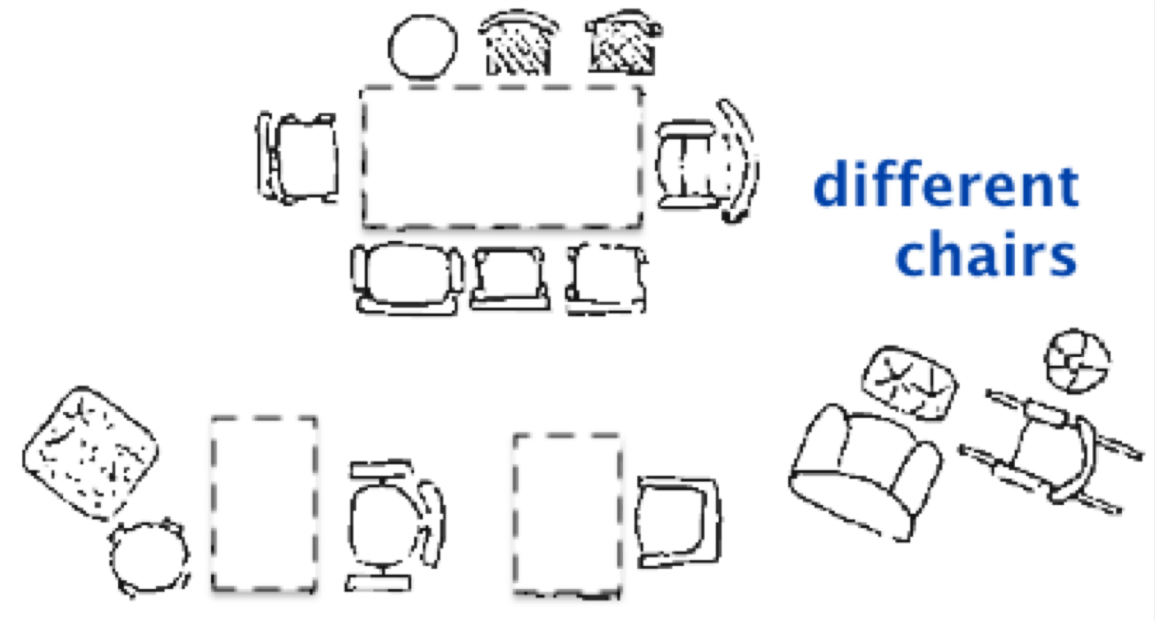
patterns

- *Problem*: “in a house for a small family, it is the relationship between children and adults which is most critical”.
- *Solution*: “give the house three distinct parts: a realm for parents, a realm for children, and a common area; conceive these three realms as roughly similar in size, with the commons the largest”



patterns

- *Problem*: “people are different sizes: they sit in different ways; and yet there is a tendency in modern times to make all chairs alike”
- *Solution*: “never furnish any place with chairs that are identically the same; choose a variety of different chairs, some big, some small, some (…)”



patterns

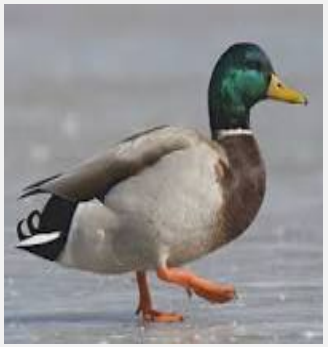
- A pattern is:
  - a reusable solution to a recurring problem
  - a pre-designed chunk, tailored to fit a given situation
  - a package of design decisions that can be reused as a set
- Patterns exist at different levels
  - system patterns (architectural styles)
  - design patterns
  - code patterns
- Patterns are not perfect for every problem.
- Their ability to separate the things that change from those that do not is useful for implementing systems.

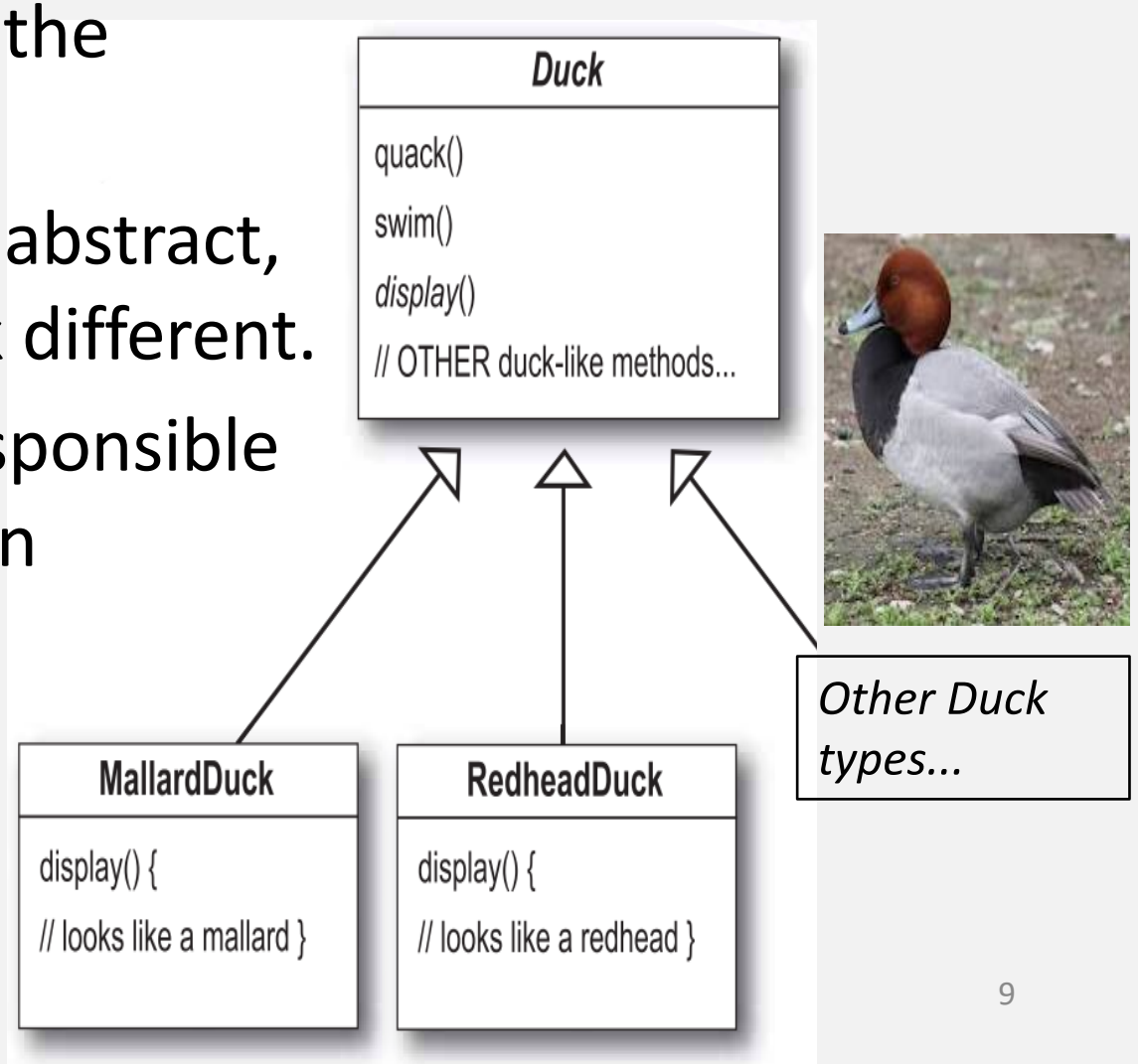
patterns

- There are 3 types of design patterns:
  - creational
  - structural
  - behavioural
- Two patterns are described:
  - strategy (structural)
  - observer (behavioural)

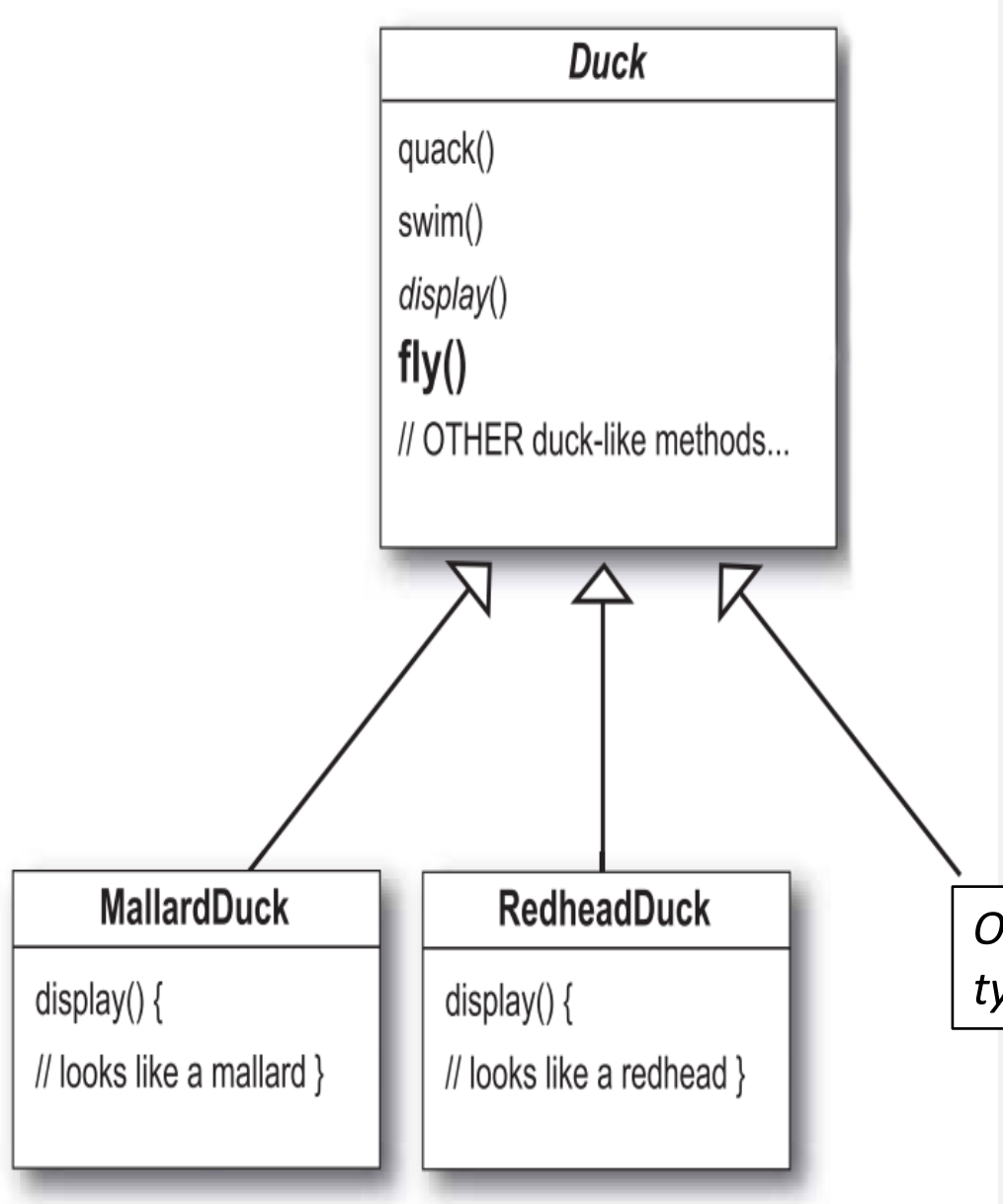
strategy pattern

- All ducks **quack** and **swim**, so the superclass takes care of the implementation code.
- The **display()** method is abstract, since all duck types look different.
- Each duck subtype is responsible for implementing its own **display()** behaviour.





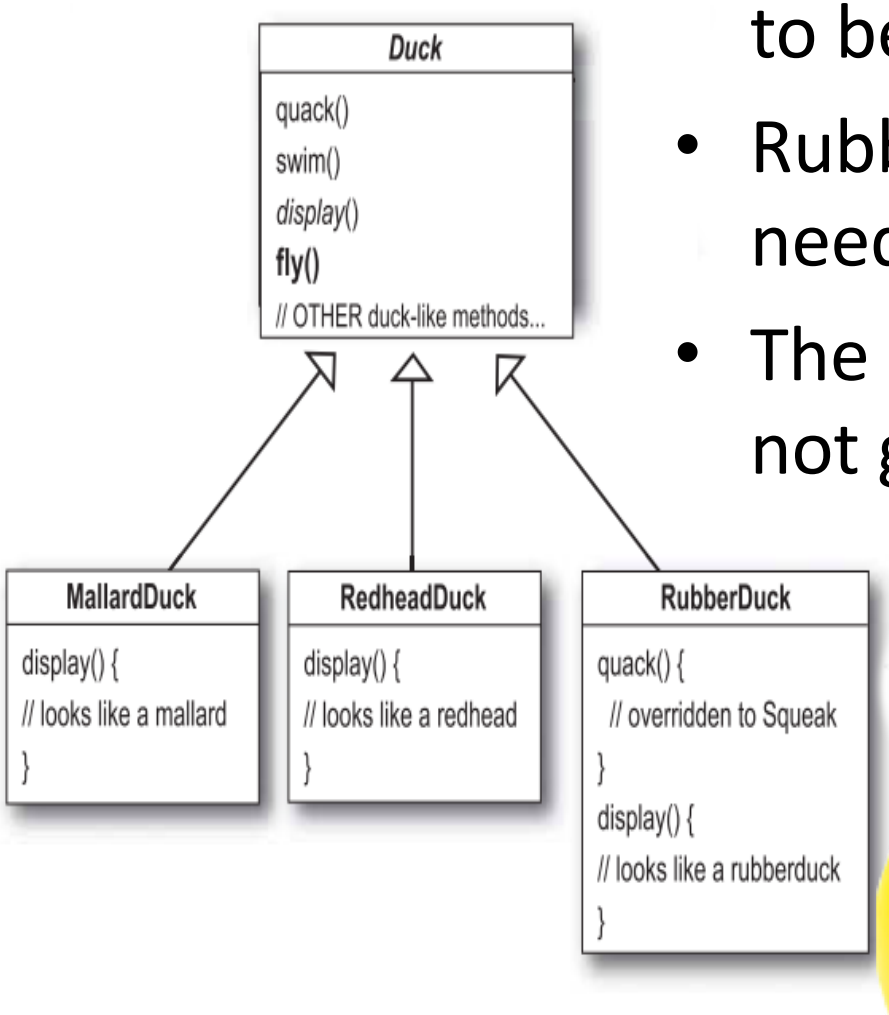
strategy pattern

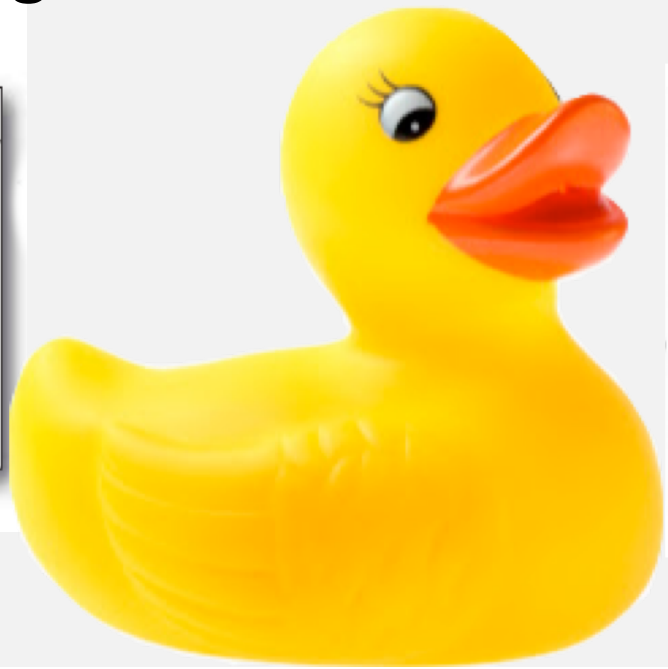


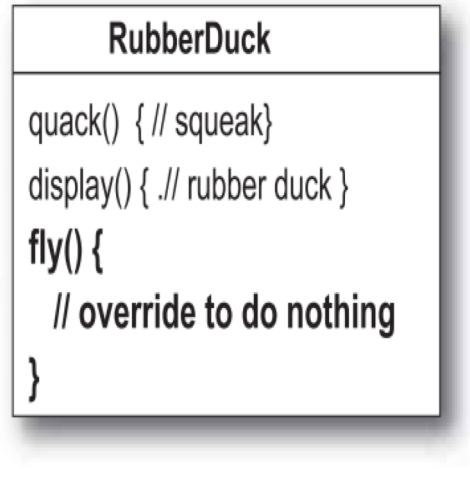
- Let us add a new feature to ducks.
- All ducks need to fly, so a **fly()** method is added to the top class.
- All subclasses inherit **fly()** by inheritance.

strategy pattern

- Rubber ducks don’t fly, so fly() needs to be overridden to do nothing
- Rubber ducks don’t quack, so quack() needs to be overridden to “squeak”
- The use of inheritance for reusing was not good for maintenance





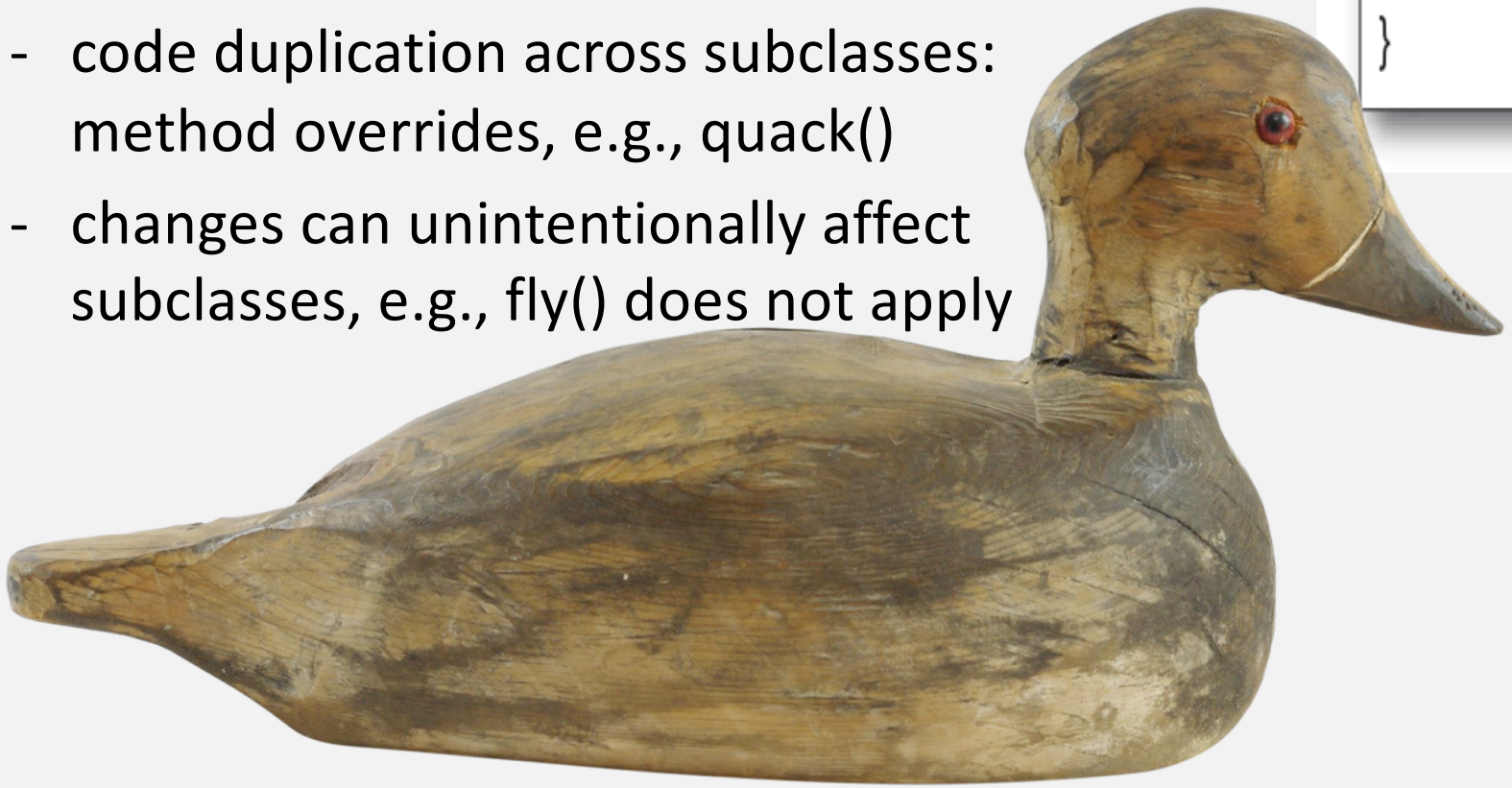


strategy pattern

- Wooden decoy ducks don't fly and don't quack, so fly() and quack() are overridden to do nothing
- Inheritance is not a good mechanism when updates are necessary.
  - code duplication across subclasses: method overrides, e.g., quack()
  - changes can unintentionally affect subclasses, e.g., fly() does not apply

DecoyDuck

```
quack() {
  // override to do nothing
}

display() { // decoy duck}

fly() {
  // override to do nothing
}
```

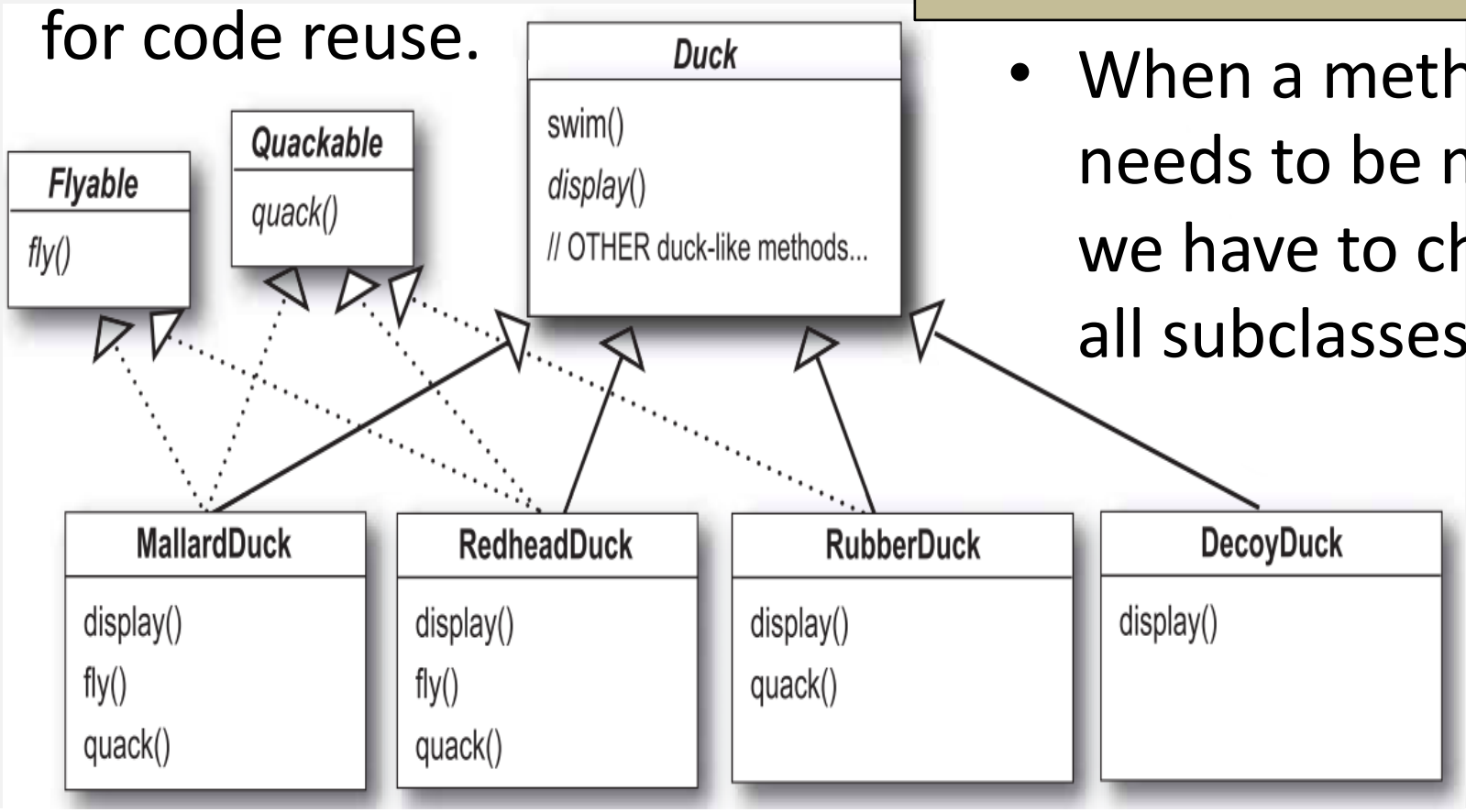


strategy pattern

- Methods fly() and quack() **vary** across ducks.
- They need to be separated.
- The use of interfaces is bad for code reuse.

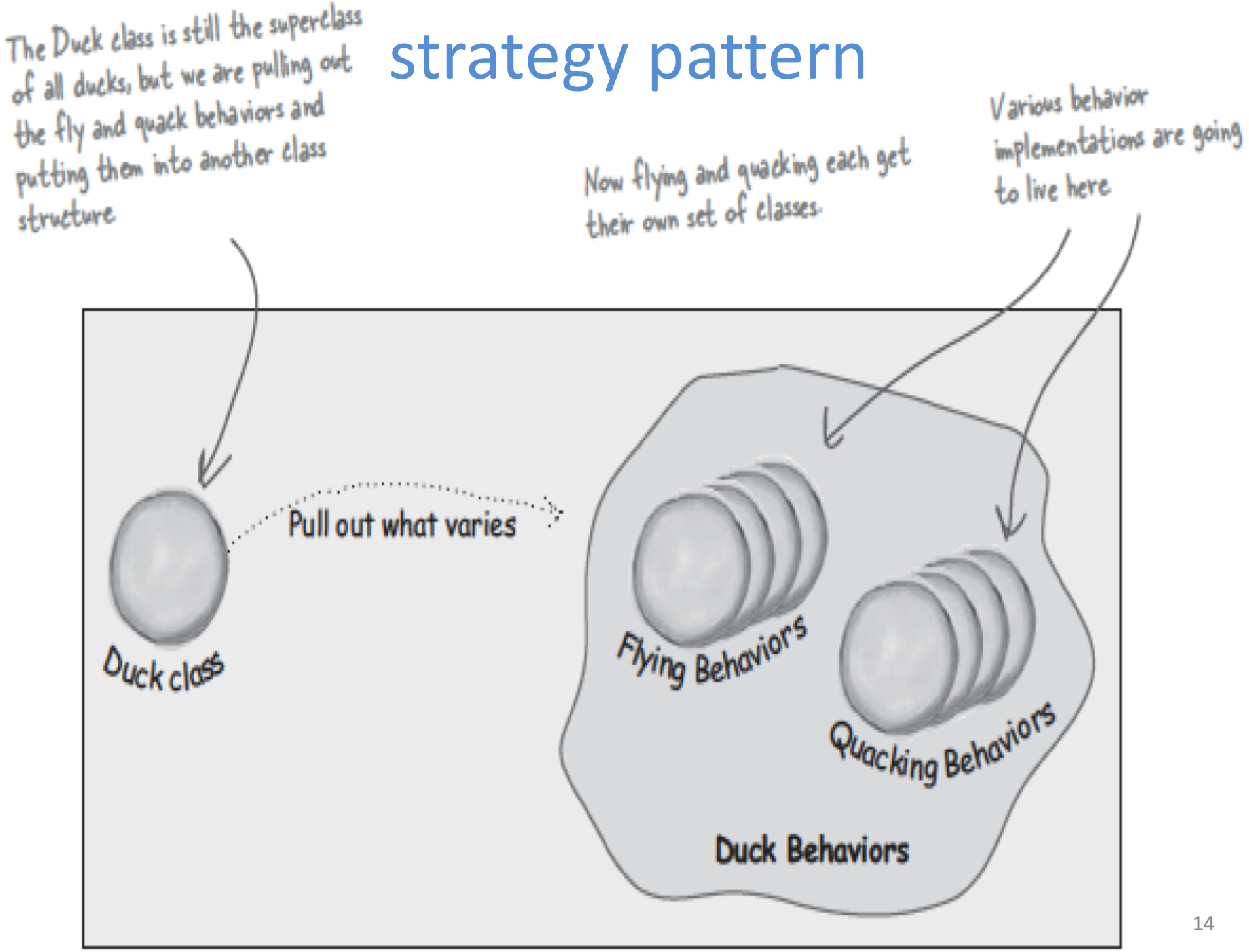
design principle

identify the aspects of the software application that vary and separate them from what remains the same



- When a method needs to be modified, we have to check it in all subclasses.

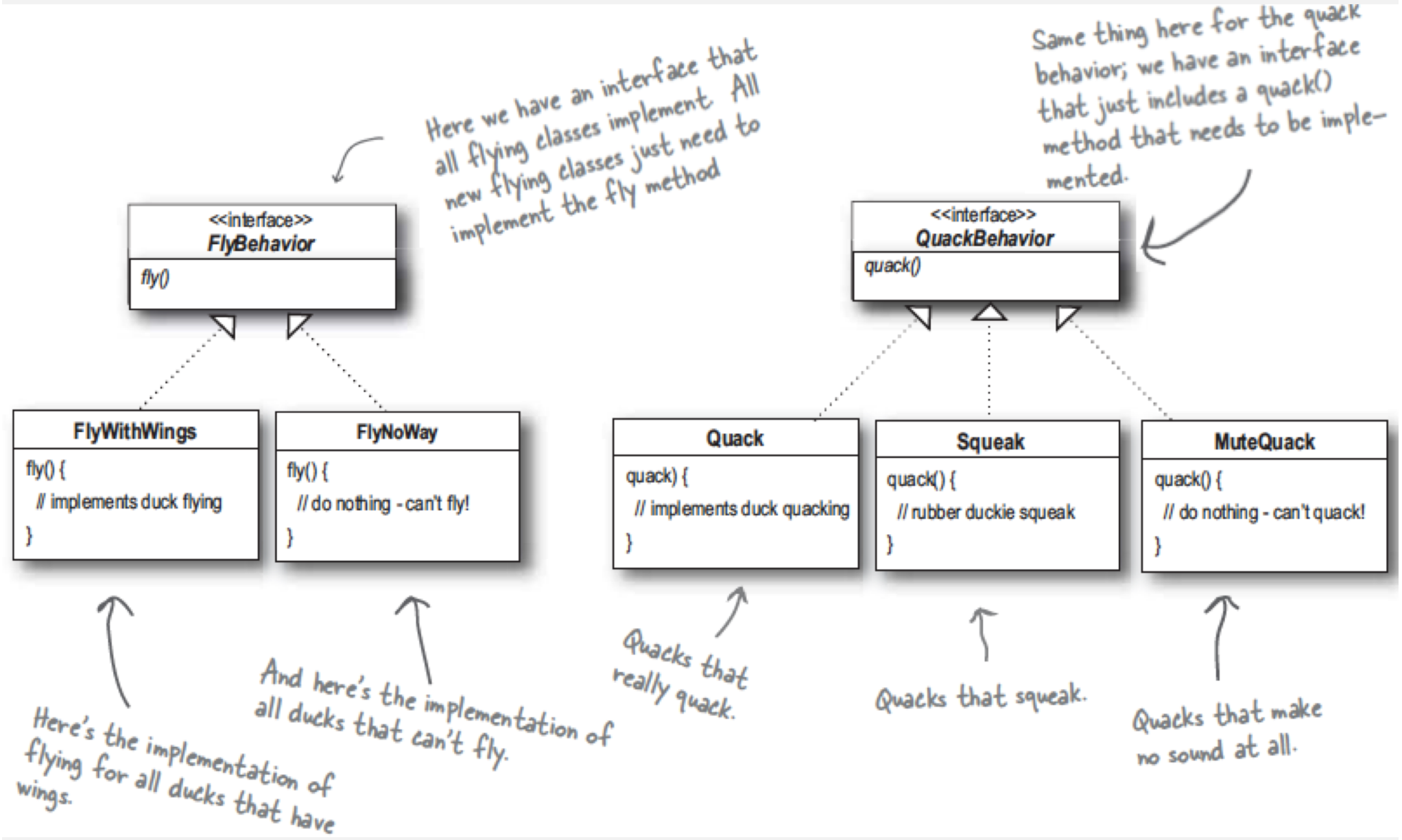
strategy pattern



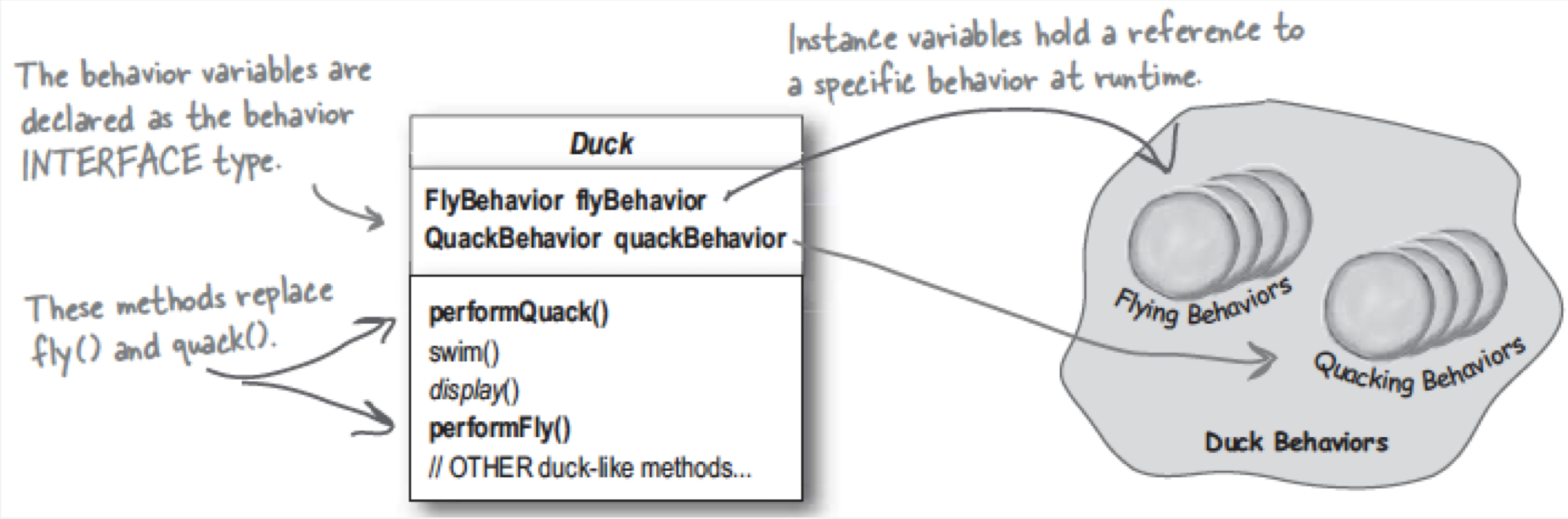
strategy pattern

design principle

program to an interface, not the implementation

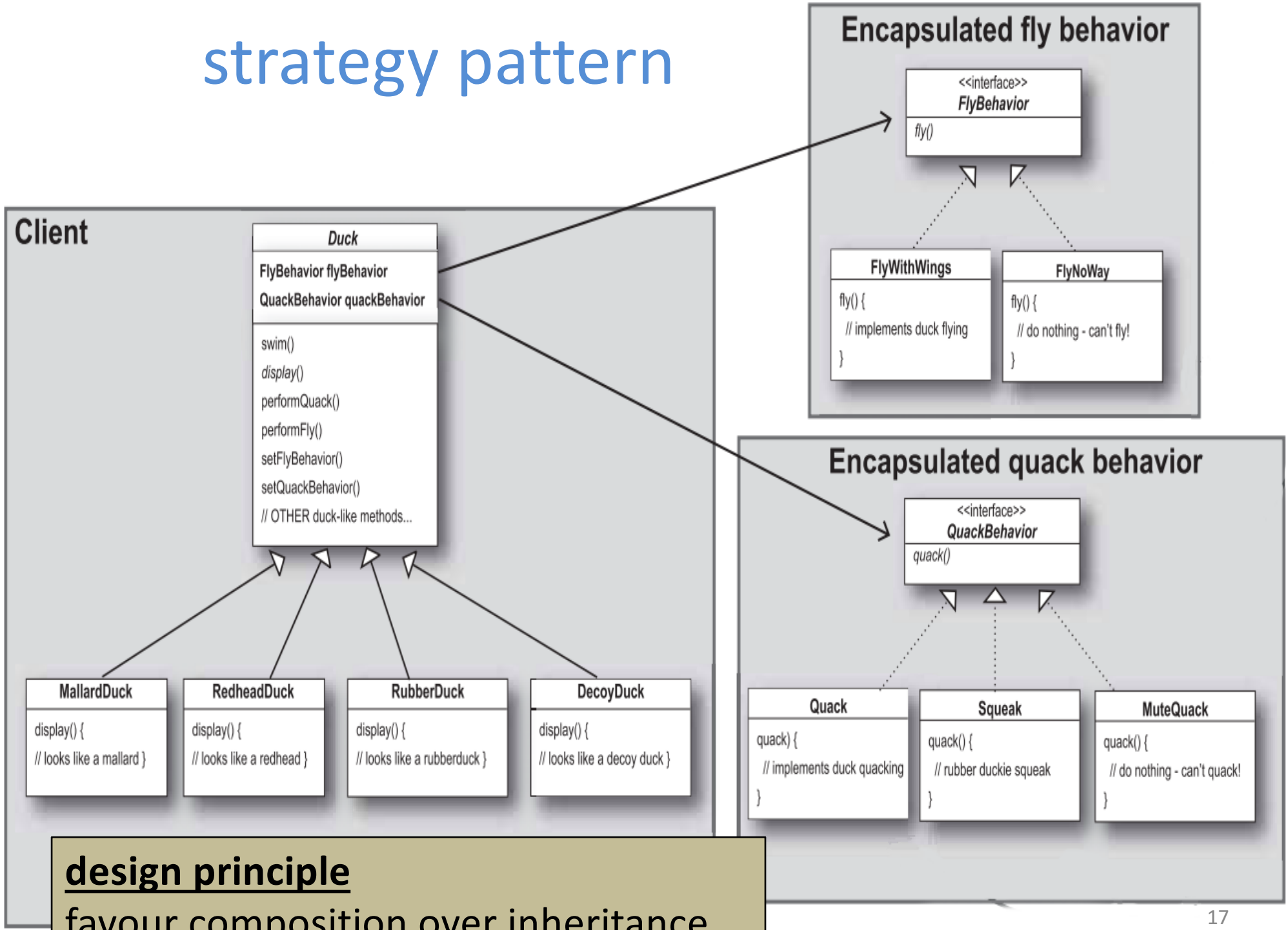


strategy pattern



- The idea is that a duck **delegates** its quacking and flying behaviours

strategy pattern



design principle

favour composition over inheritance

strategy pattern

Now we implement performQuack():

```
public class Duck {
    QuackBehavior quackBehavior;
    // more

    public void performQuack() {
      quackBehavior.quack();
    }
}
```

Each Duck has a reference to something that implements the QuackBehavior interface.

Rather than handling the quack behavior itself, the Duck object delegates that behavior to the object referenced by quackBehavior.

```
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");
    }
}
```

A MallardDuck uses the Quack class to handle its quack, so when performQuack is called, the responsibility for the quack is delegated to the Quack object and we get a real quack.

And it uses FlyWithWings as its FlyBehavior type.

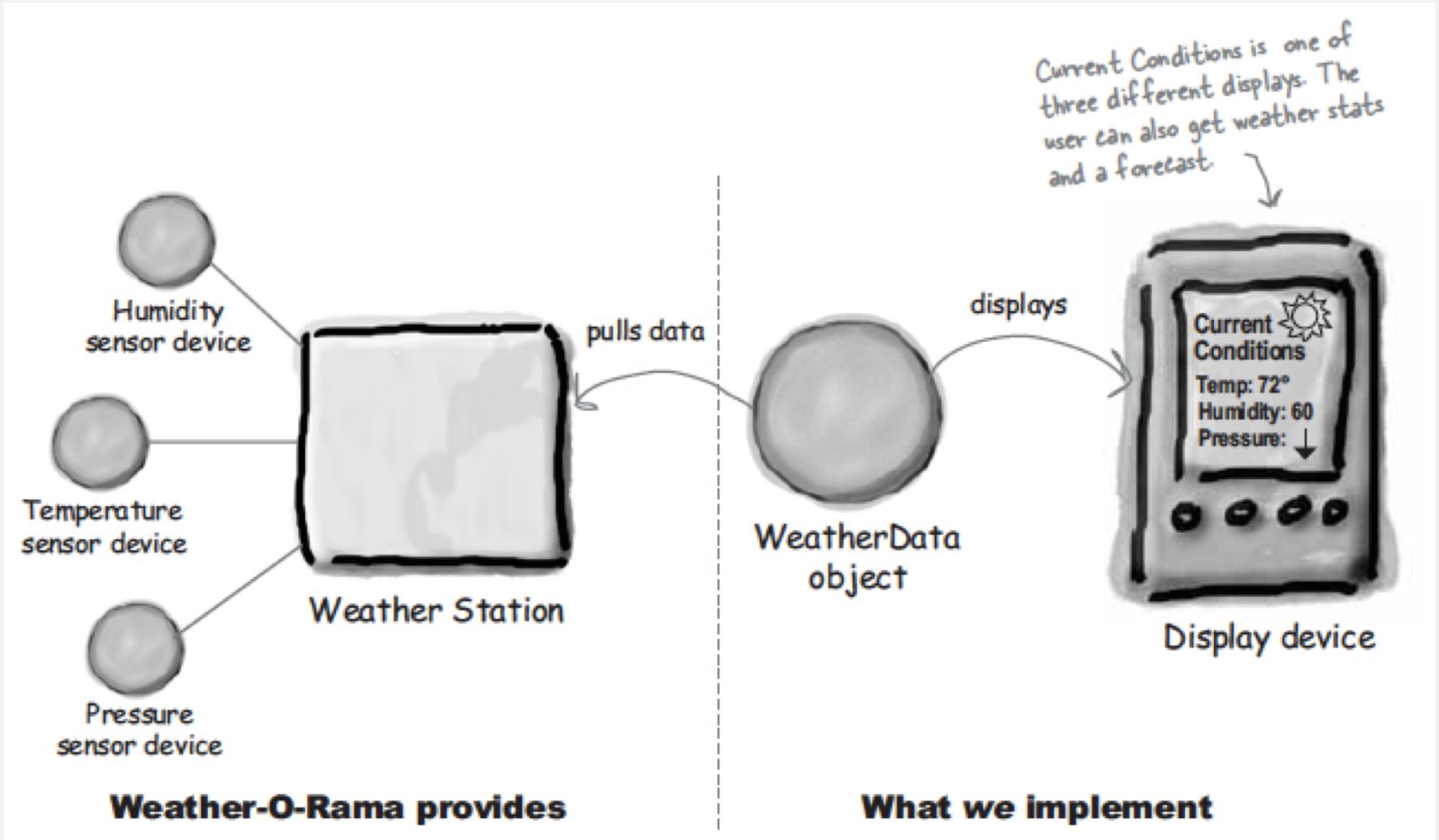
Remember, MallardDuck inherits the quackBehavior and flyBehavior instance variables from class Duck.

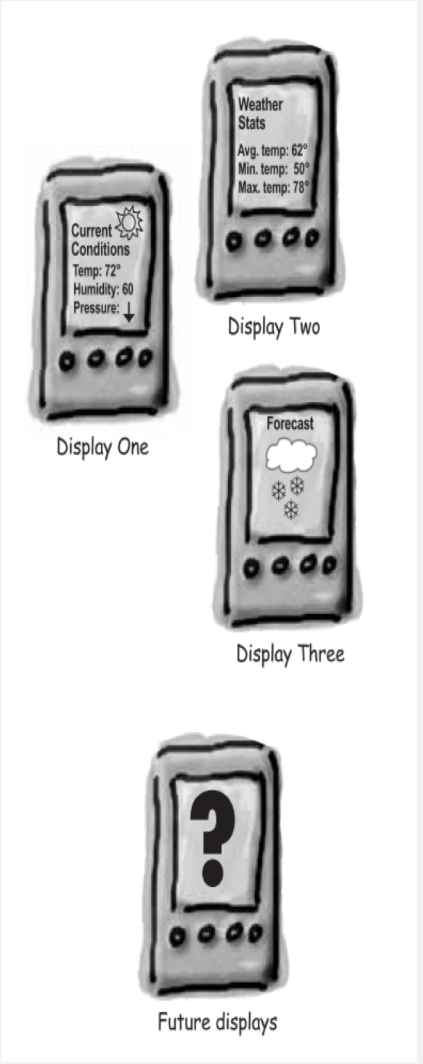
strategy pattern

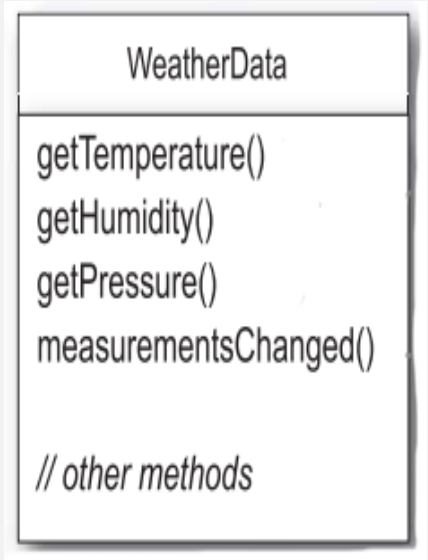
- The strategy pattern was used to rework the ducks application.
- The **strategy pattern** defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- It lets the algorithm vary independently of the client that uses it.

observer pattern

weather monitoring app



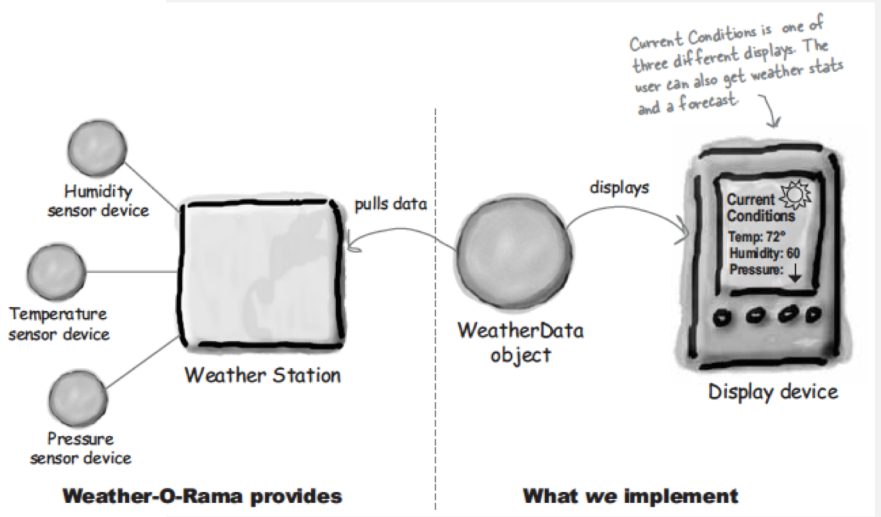




observer pattern

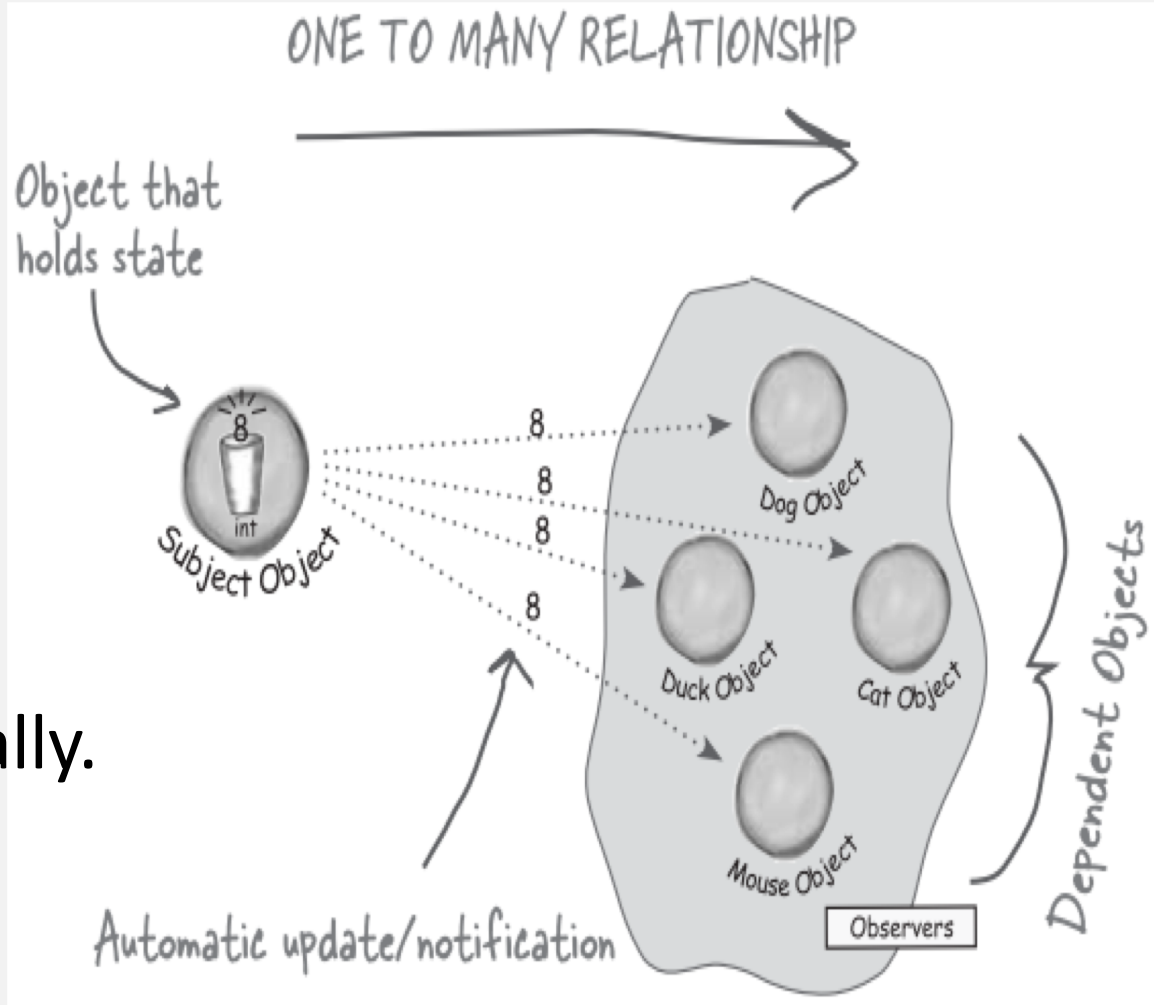
- *Problem*: if new Displays are added we need to update the Weather Data
- *Opportunity*: the information/interface needed by the displays is similar

```
public class WeatherData {

    // instance variable declarations

    public void measurementsChanged() {

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

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

    // other WeatherData methods here
}
```

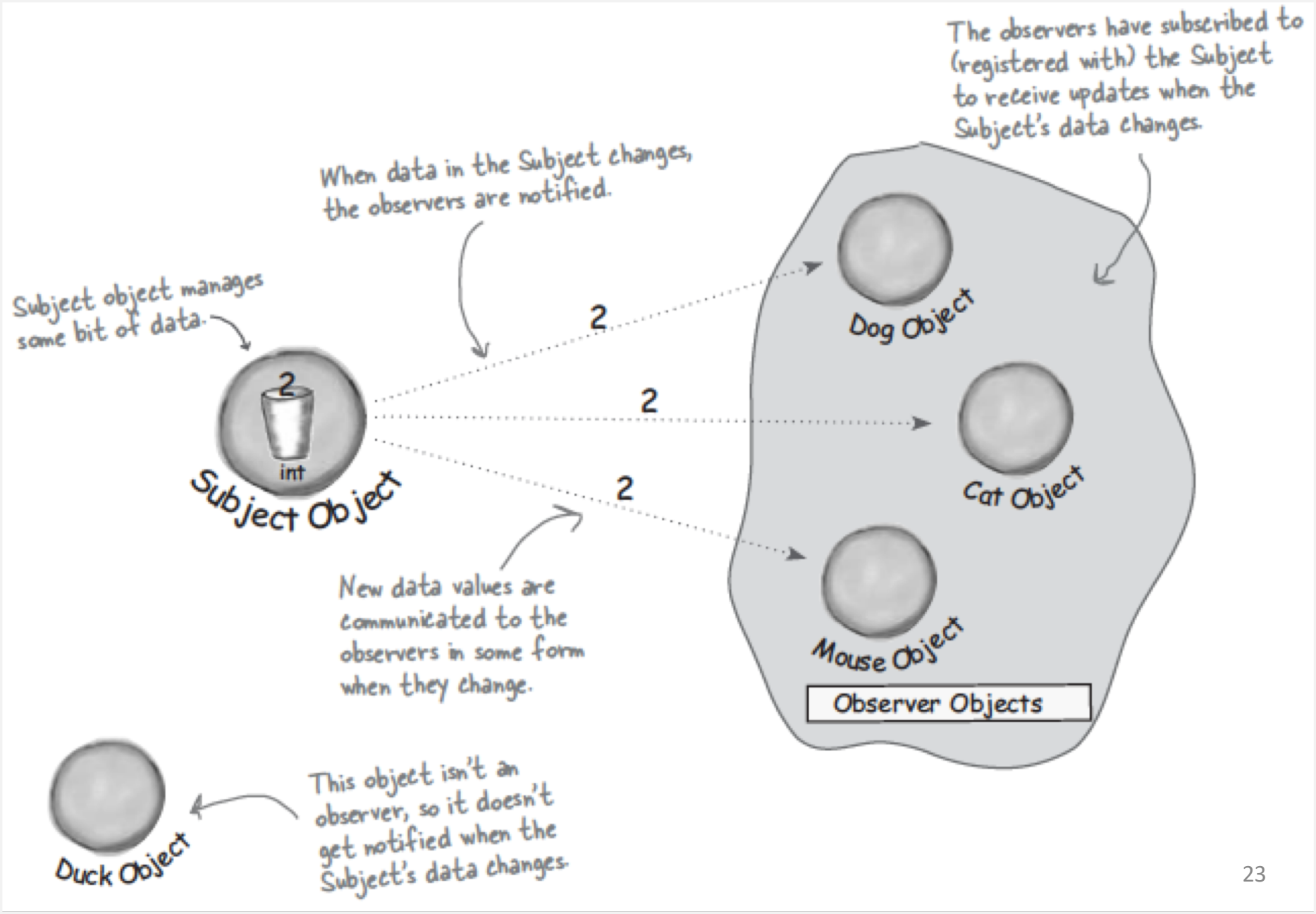


observer pattern

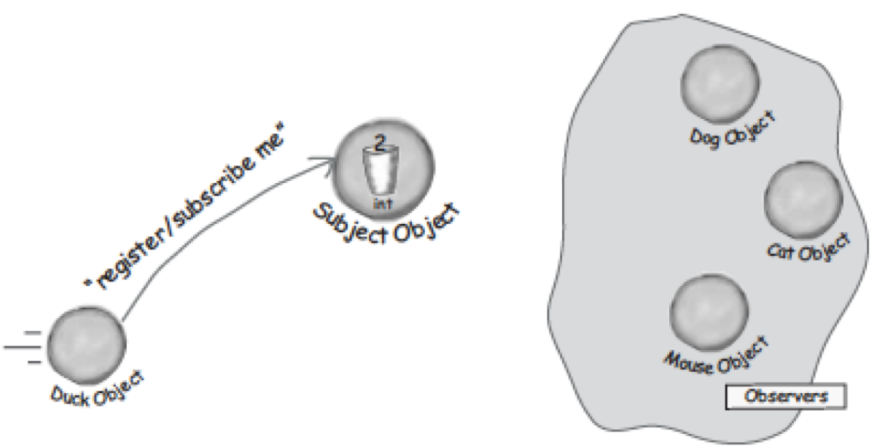
- The **observer pattern** defines a one-to-many dependency between objects.
- Whenever one object changes state, all its dependents are notified and updated automatically.

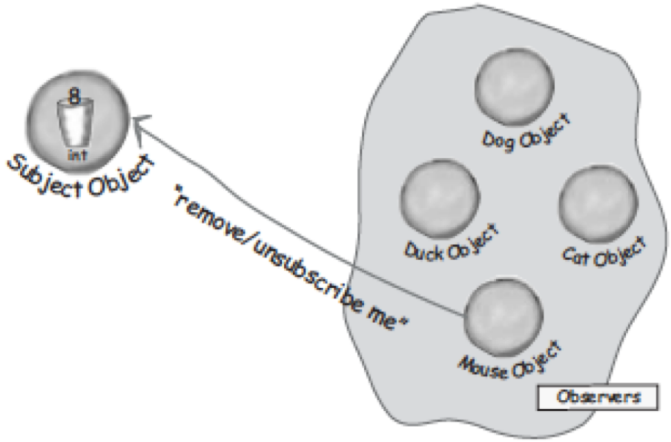


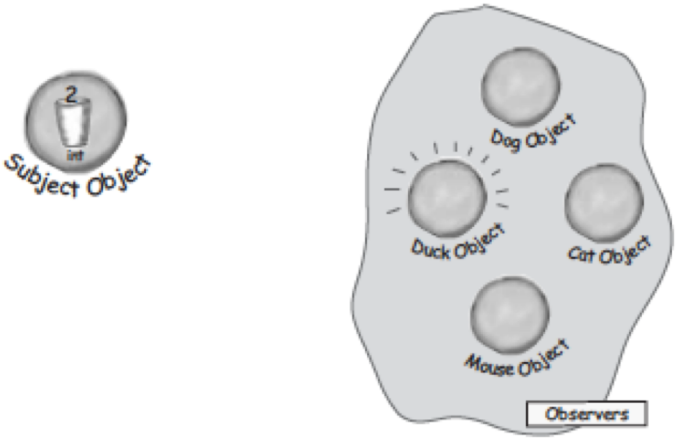
observer pattern

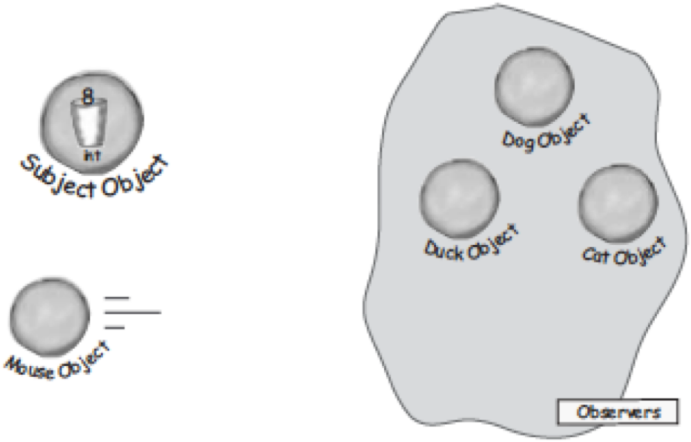


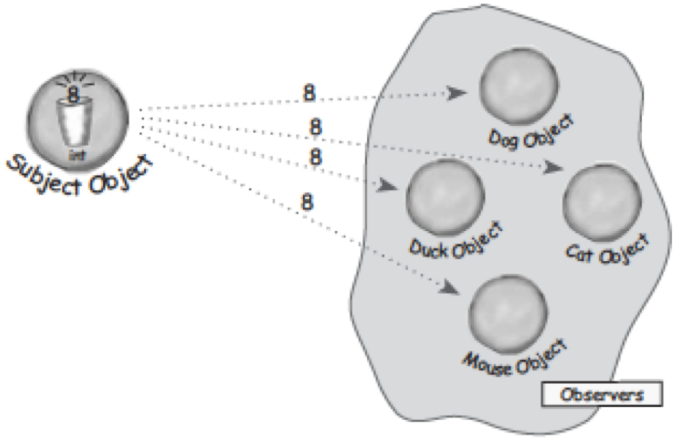
observer pattern

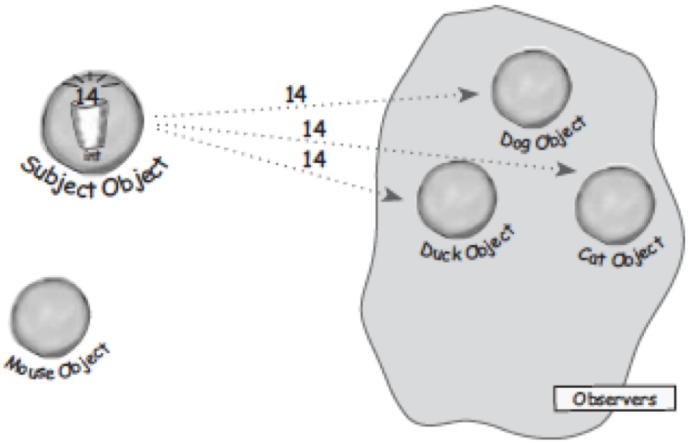




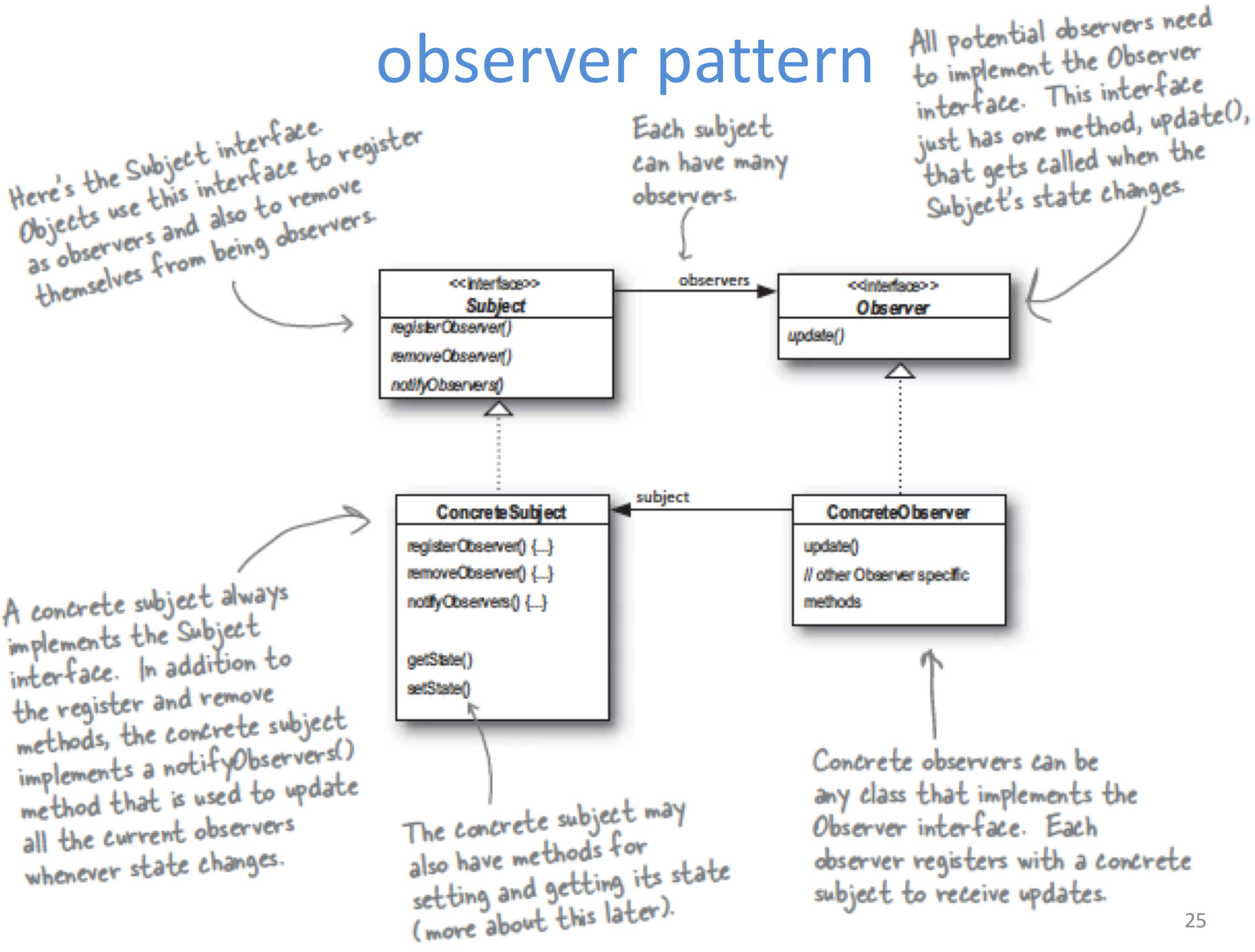


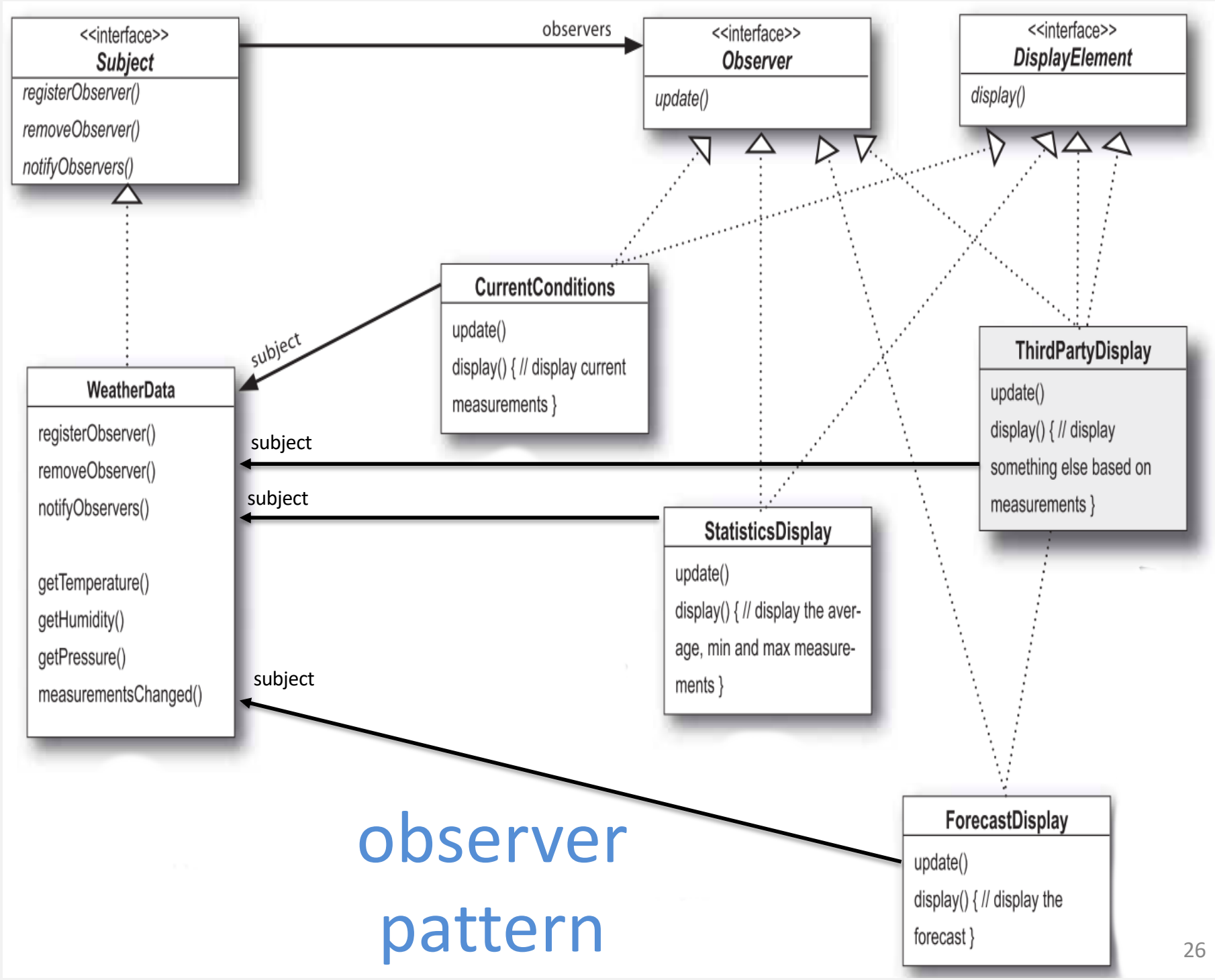






observer pattern

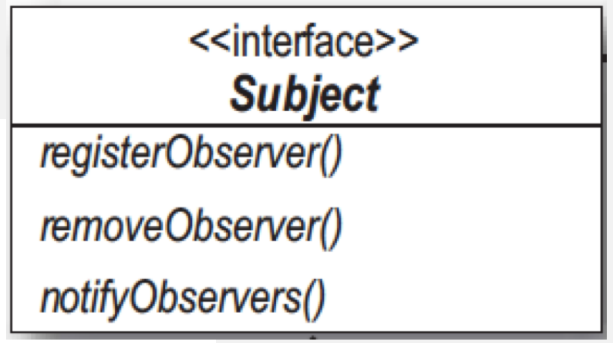




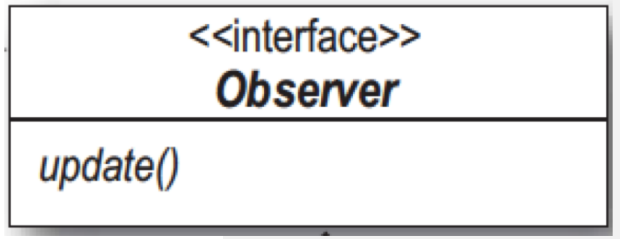
observer pattern

observer pattern

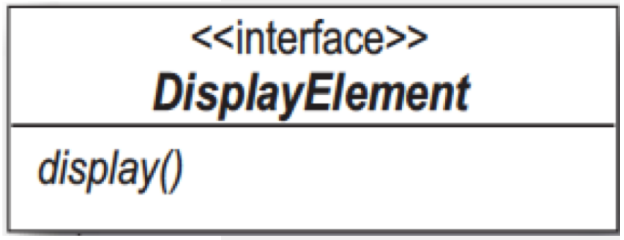
```
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
```




```
public interface Observer {
    public void update(float temp, float humidity, float pressure);
}
```

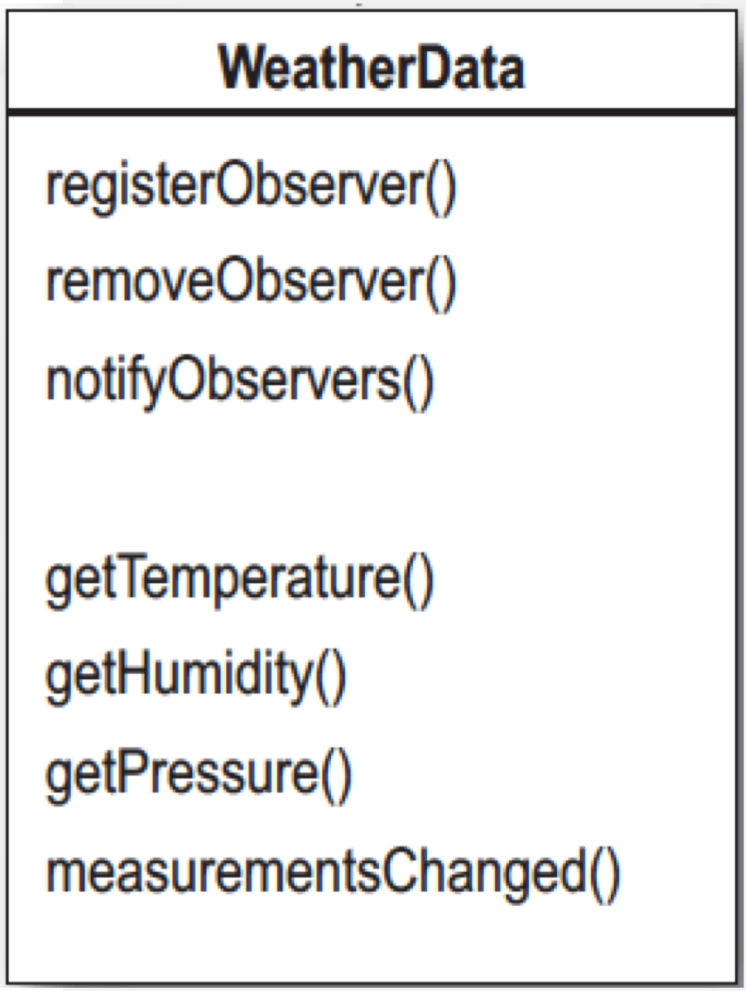



```
public interface DisplayElement {
    public void display();
}
```



observer pattern

```
public class WeatherData implements Subject {
    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList();
    }

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

    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }

    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer)observers.get(i);
            observer.update(temperature, humidity, pressure);
        }
    }

    public void measurementsChanged() {
        notifyObservers();
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }
```

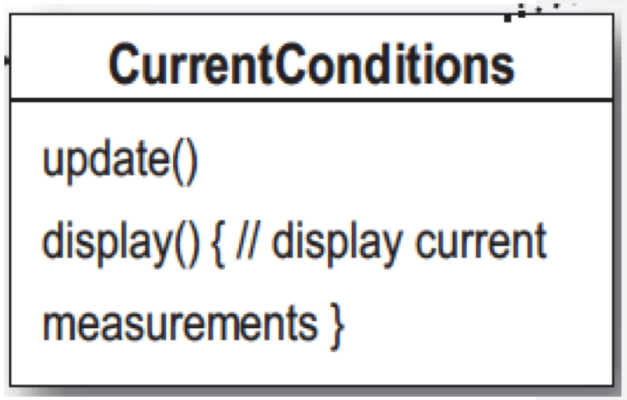


observer pattern

```
public class WeatherStation {
    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();

        CurrentConditionsDisplay currentDisplay =
            new CurrentConditionsDisplay(weatherData);
        StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
        ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

        weatherData.setMeasurements(80, 65, 30.4f);
        weatherData.setMeasurements(82, 70, 29.2f);
        weatherData.setMeasurements(78, 90, 29.2f);
    }
}
```

observer pattern

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private Subject weatherData;

    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
    }
}
```

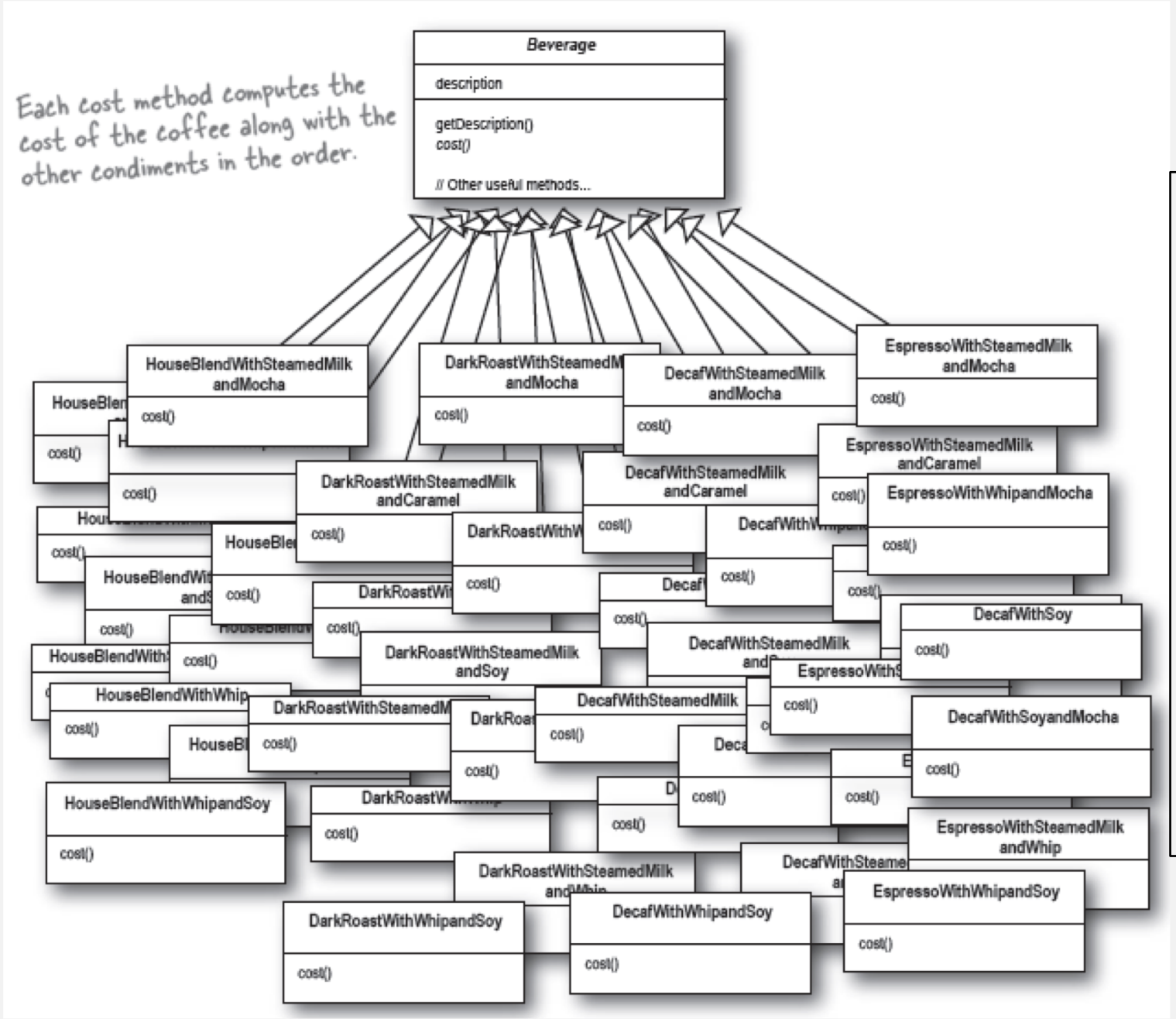


bibliography

- Freeman E, Robson E, Bates B, and Sierra K; *Head-first design patterns*, O'Reilly Media, 2004.

decorator pattern

decorator pattern



- condiments
  - *steamed milk*
  - *mocha*
  - *whip*
  - *Soy*
- toppings
- size
- …

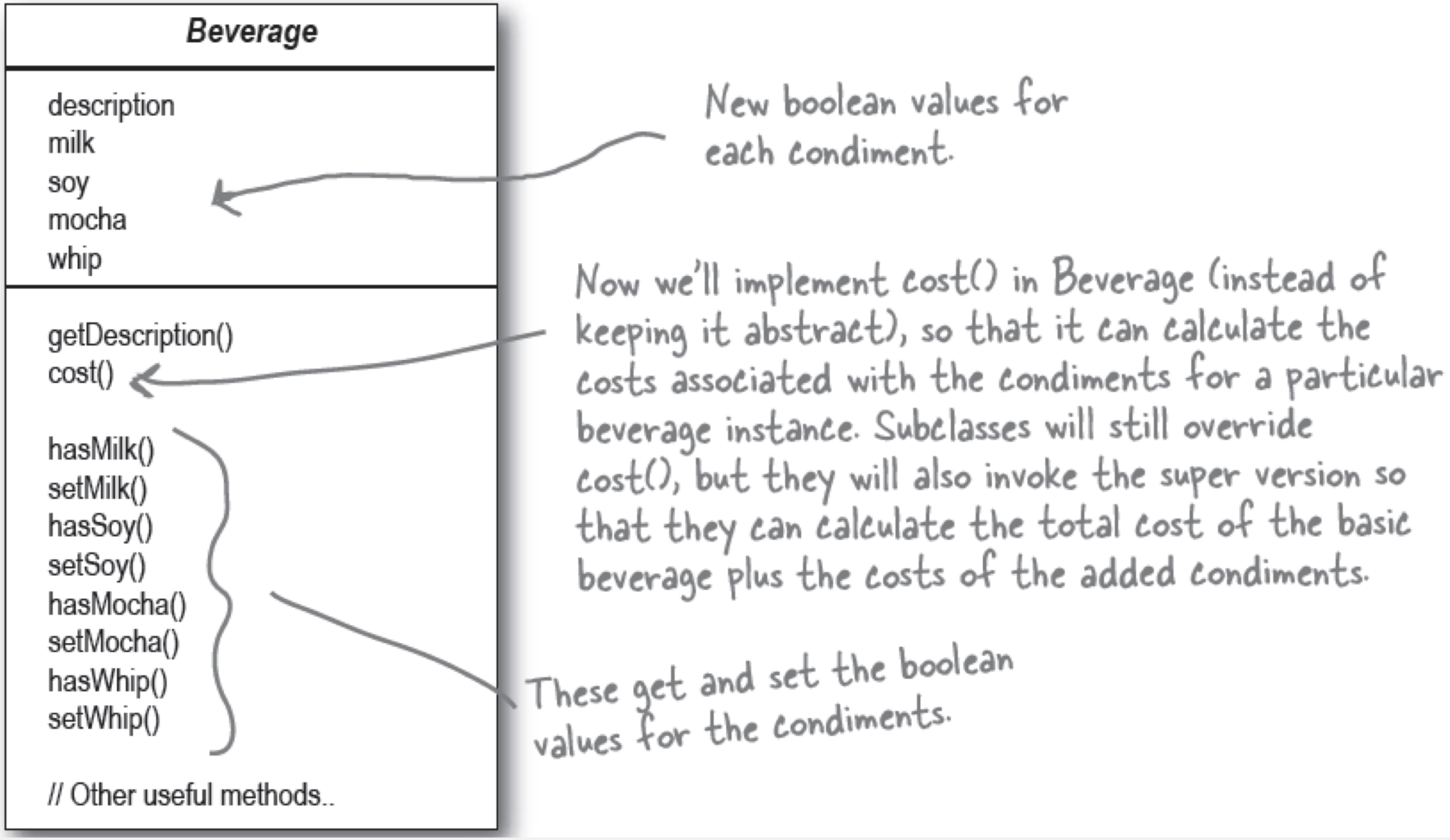
decorator pattern

- Inheritance explodes the number of subclasses and creates maintenance problems, how and why?
- What principles are we violating?

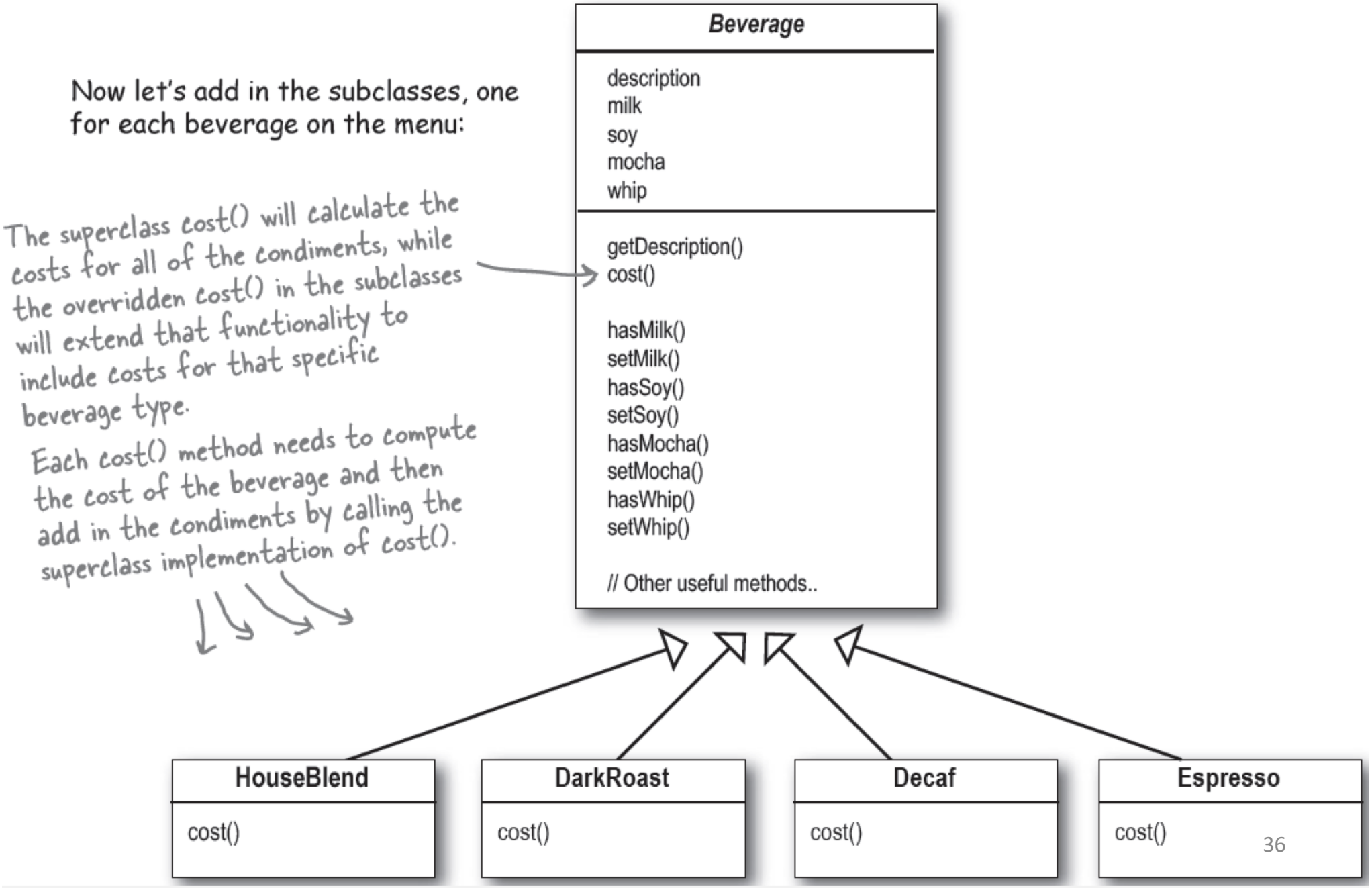
design principle
identify the aspects of your application that vary and separate them from what stays the same

design principle
program to an interface, not the implementation

decorator pattern



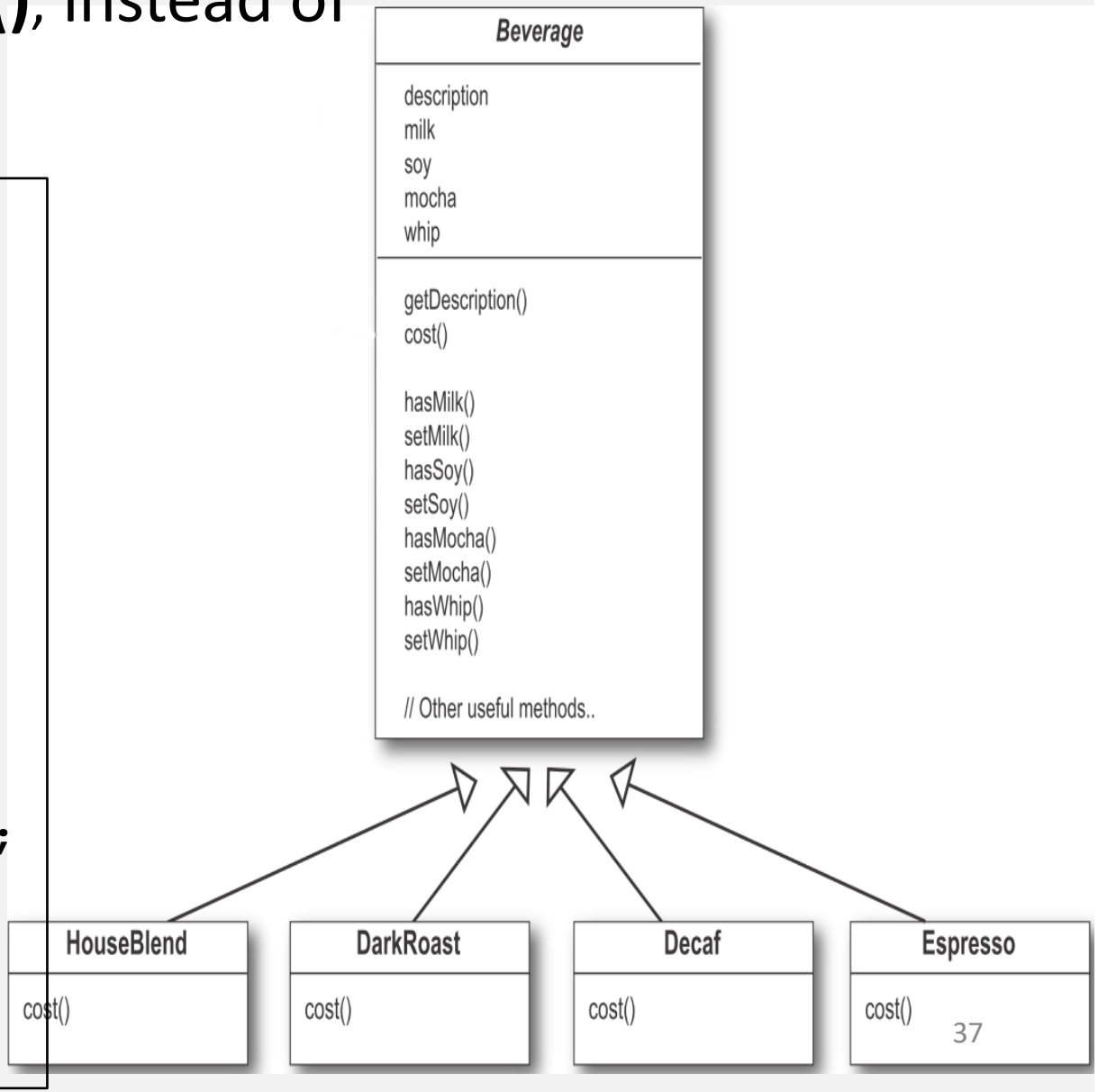
decorator pattern



decorator pattern

Move the condiments to the abstract class implementing **cost()**, instead of keeping it abstract.

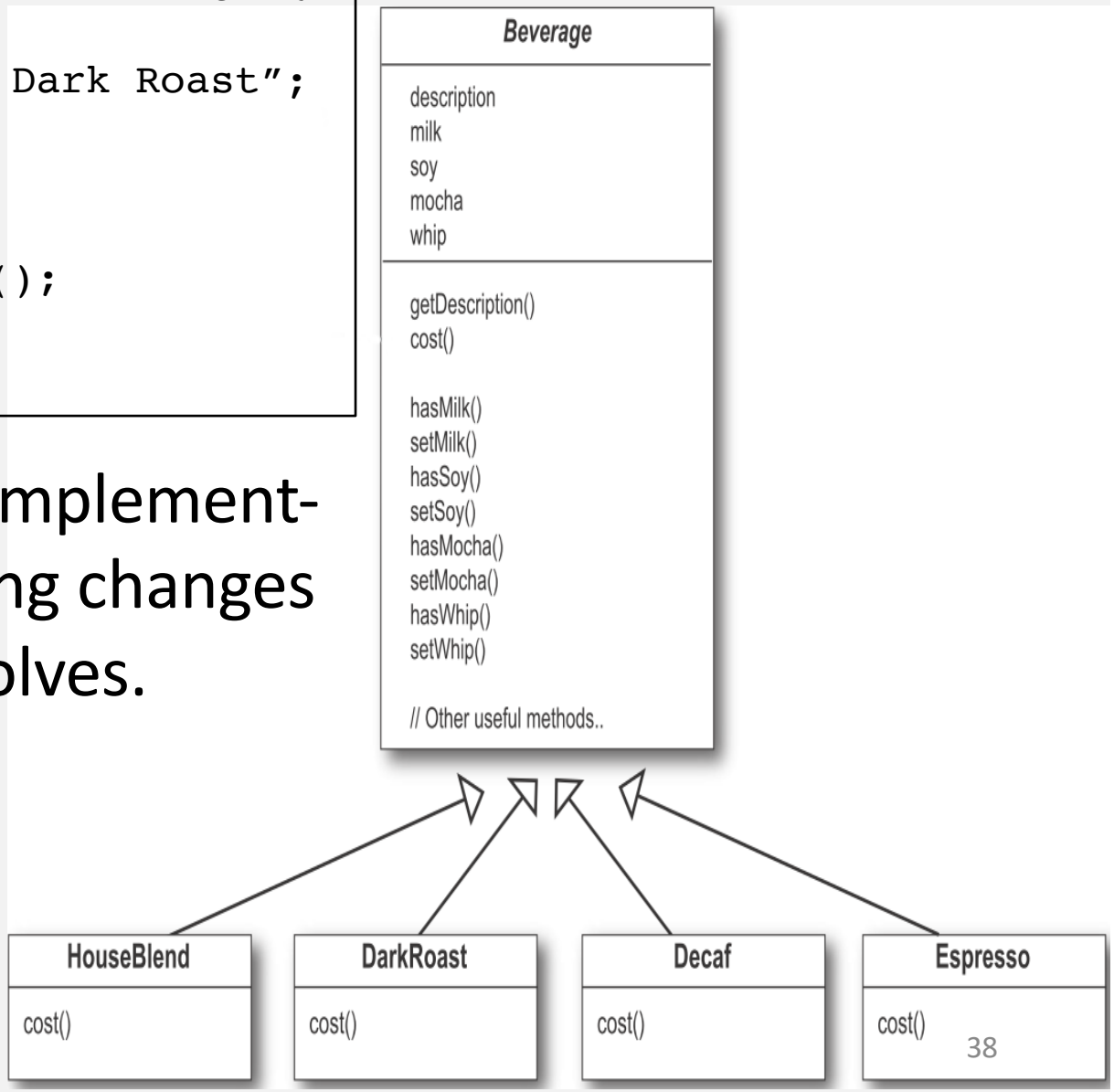
```
public class Beverage {
soyCost;
whipCost;
mochaCost;
milkCost;
public double cost() {
condimentsCost = 0
if(hasMilk())
 condimentCost =+ milkCost;
if(hasSoy())
 condimentCost =+ soyCost;
if(hasMocha())
 condimentCost =+ mochaCost;
if(hasWhip())
 condimentCost =+ whipCost
return condimentCost;
}
```



decorator pattern

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

  public double cost() {
    return 1.99 + super.cost();
  }
}
```

Identify issues with this implement-tation w.r.t. the minimizing changes when the application evolves.



decorator pattern

Identify issues with this implementation with respect to the minimizing changes when application evolves

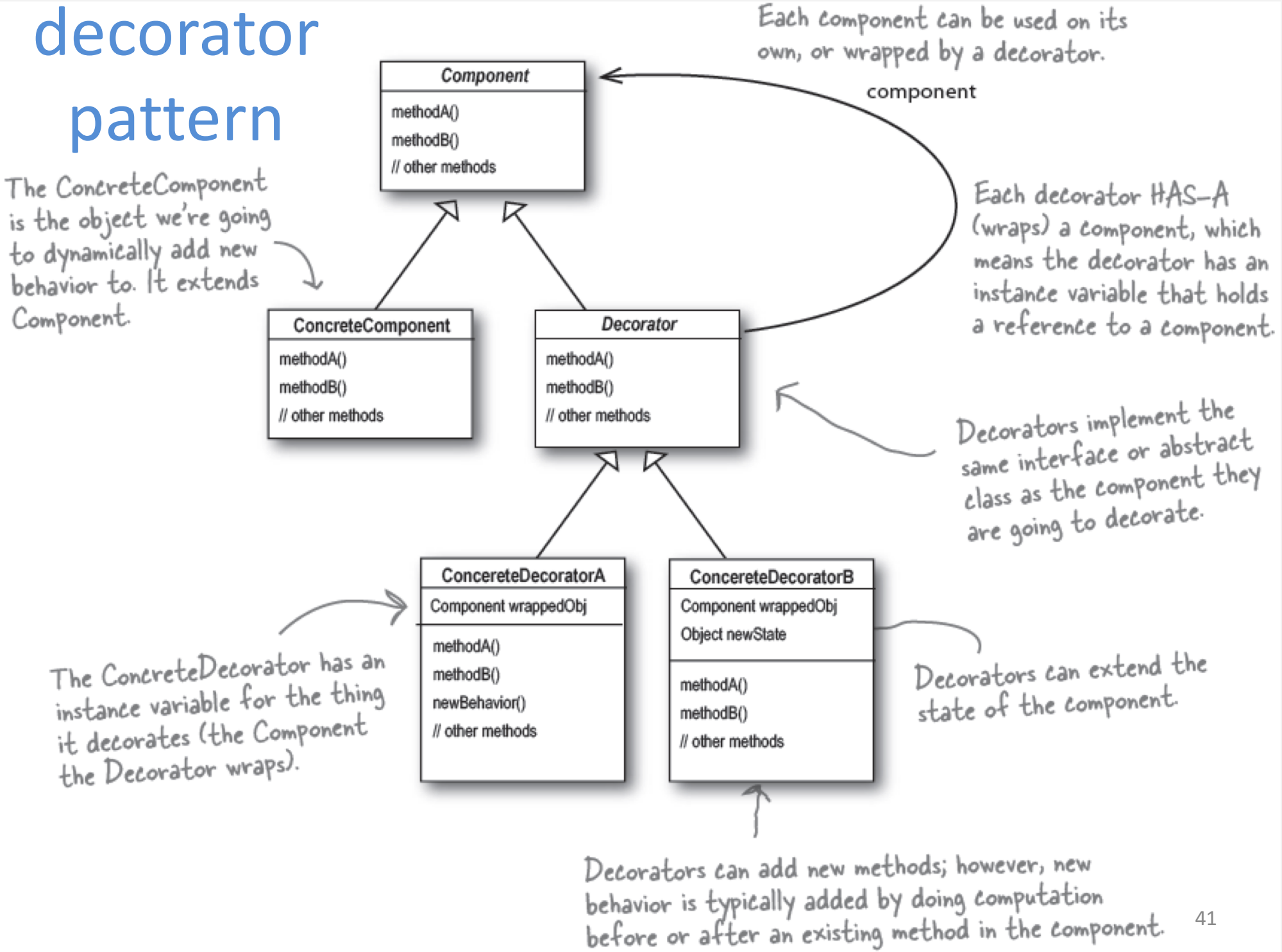
1. Price changes for condiments require changes to the existing code
2. New condiments force to add new methods class and update the cost method
3. If new beverages types are added where condiments do not apply (e.g., ice tea), the IceTea subclass will still inherit methods like hasWhip()
4. What if a customer wants a double mocha?

decorator pattern

open-closed principle: a class/module should be open for extension, but closed for modification

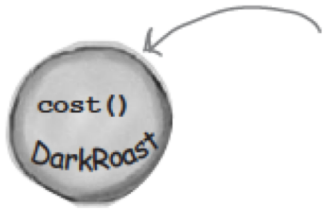
- This is the most important principle of OO design.
- Modules should be written so that they can be extended, without requiring them to be modified.
- We want to be able to change what the modules do, without changing their code.
- The **decorator pattern** attaches additional responsibilities to an object dynamically.
- It provides a flexible alternative to subclassing for extending functionality.

decorator pattern



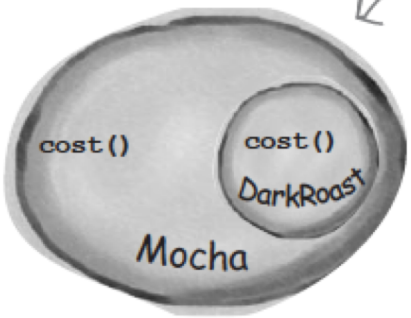
decorator pattern

1 We start with our DarkRoast object.



Remember that DarkRoast inherits from Beverage and has a cost() method that computes the cost of the drink.

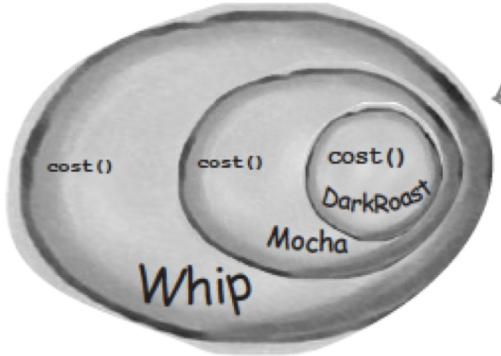
2 The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.



The Mocha object is a decorator. Its type mirrors the object it is decorating, in this case, a Beverage. (By "mirror", we mean it is the same type..)

So, Mocha has a cost() method too, and through polymorphism we can treat any Beverage wrapped in Mocha as a Beverage, too (because Mocha is a subtype of Beverage).

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

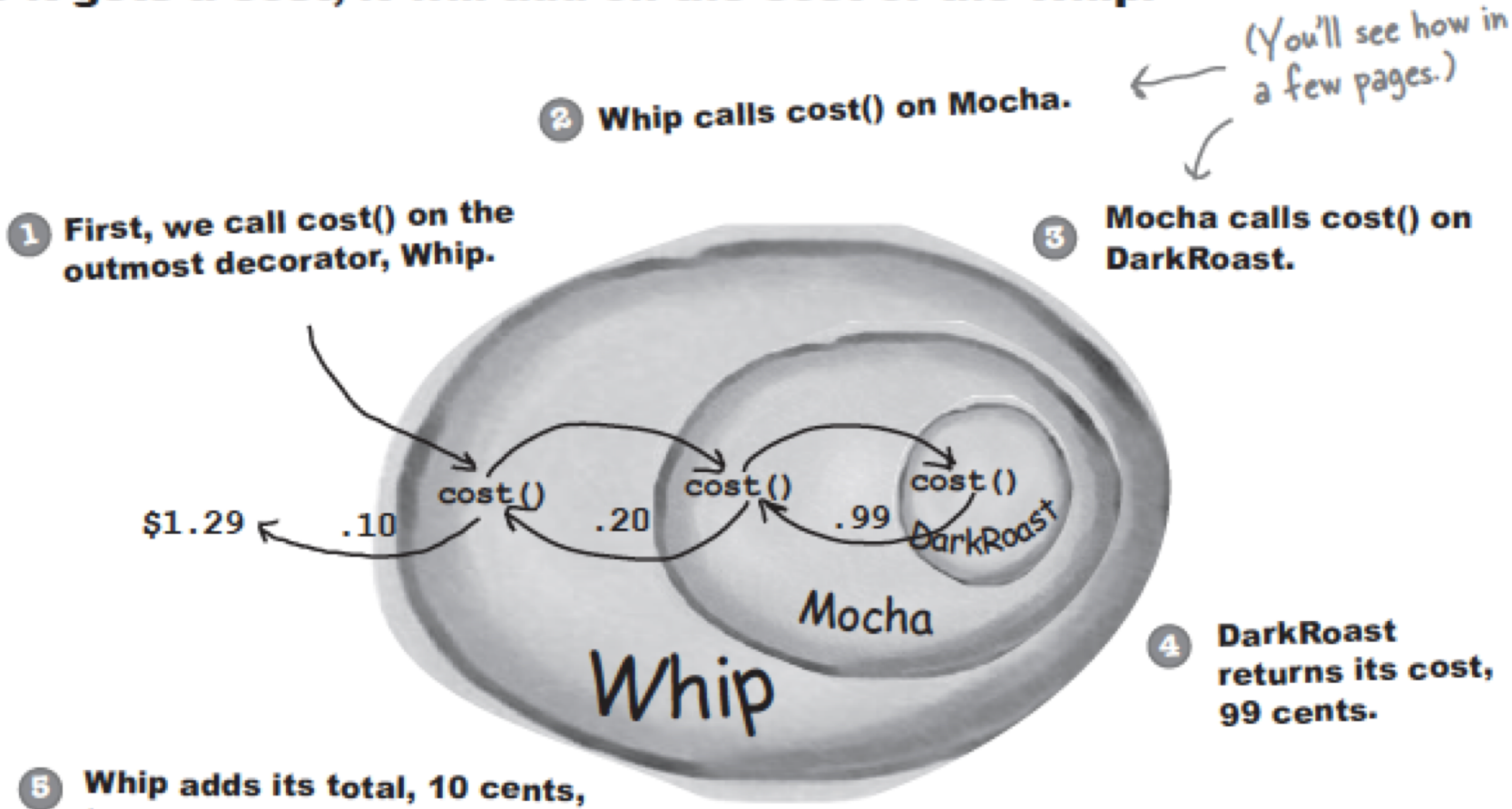


Whip is a decorator, so it also mirrors DarkRoast's type and includes a cost() method.

So, a DarkRoast wrapped in Mocha and Whip is still a Beverage and we can do anything with it we can do with a DarkRoast, including call its cost() method.

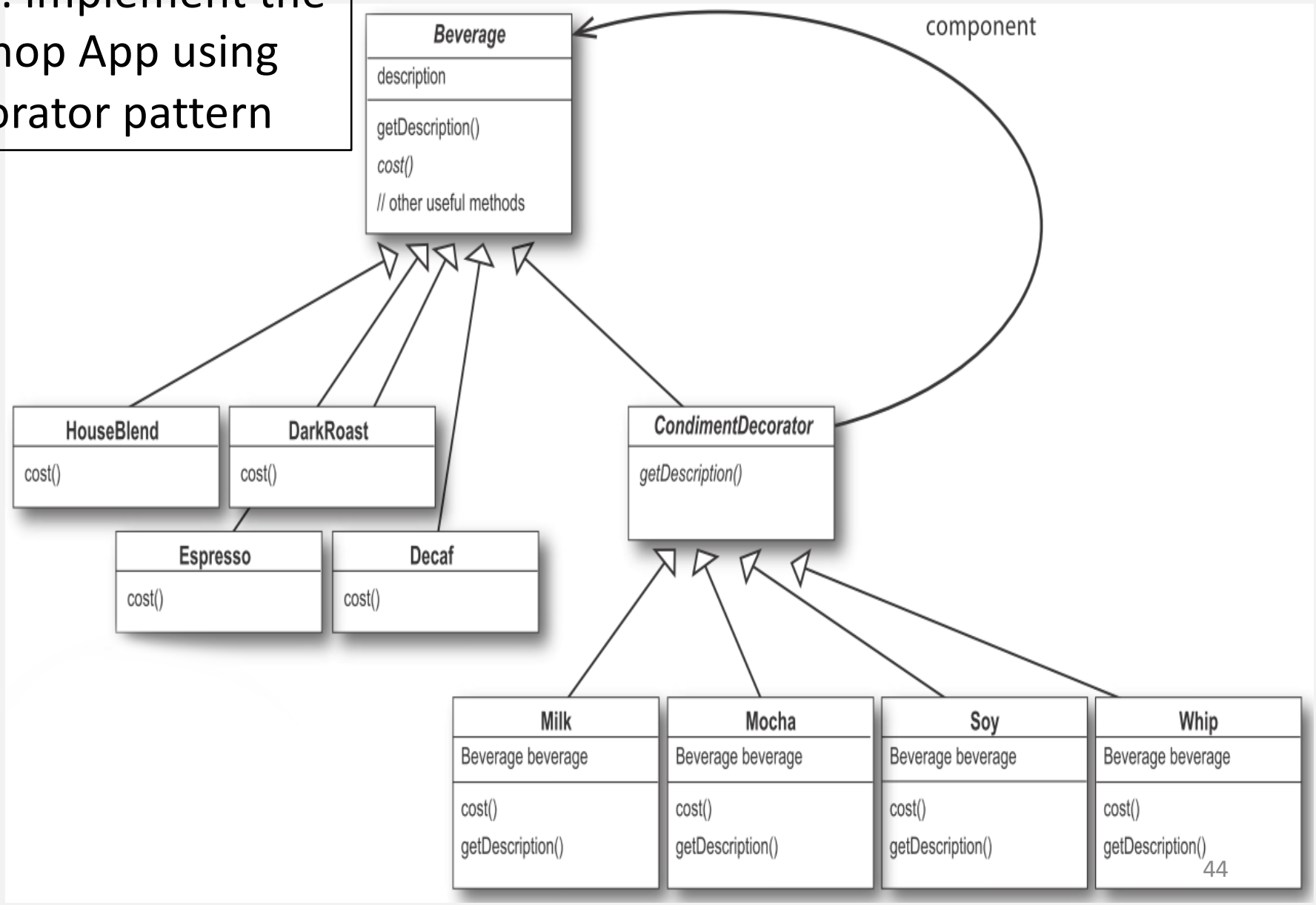
decorator pattern

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.**



decorator pattern

Exercise: implement the CoffeeShop App using the decorator pattern



decorator pattern

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

    public String getDescription() {
        return description;
    }

    public abstract double cost();
}
```

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

```
public class Espresso extends Beverage {

    public Espresso() {
        description = "Espresso";
    }

    public double cost() {
        return 1.99;
    }
}
```

decorator pattern

```
public class HouseBlend extends Beverage {
    public HouseBlend() {
        description = "House Blend Coffee";
    }

    public double cost() {
        return .89;
    }
}
```

```
public class Mocha extends CondimentDecorator {
    Beverage beverage;

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

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

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

decorator pattern

```
public static void main(String args[]) {
    Beverage beverage = new Espresso();
    System.out.println(beverage.getDescription()
            + " $" + beverage.cost());

    Beverage beverage2 = new DarkRoast();
    beverage2 = new Mocha(beverage2);
    beverage2 = new Mocha(beverage2);
    beverage2 = new Whip(beverage2);
    System.out.println(beverage2.getDescription()
            + " $" + beverage2.cost());

    Beverage beverage3 = new HouseBlend();
    beverage3 = new Soy(beverage3);
    beverage3 = new Mocha(beverage3);
    beverage3 = new Whip(beverage3);
    System.out.println(beverage3.getDescription()
            + " $" + beverage3.cost());
}
```