DESIGN PATTERNS

Eng. Hany Safwat

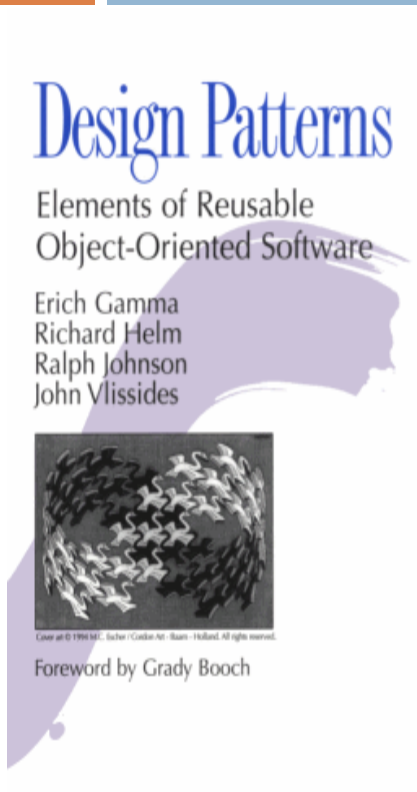
Design Patterns

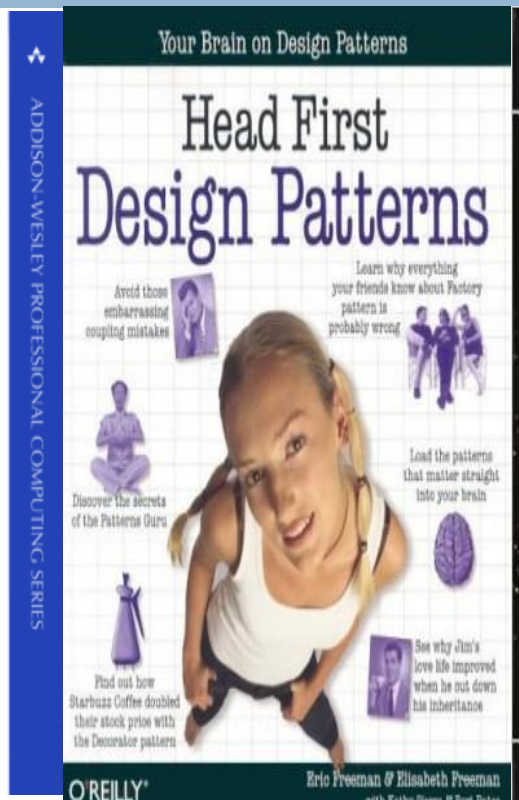
- General repeatable **solution** to a commonly occurring problem in software design. A design pattern is **not a finished design** that can be transformed directly into code.
- It is a **description** or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show **relationships and interactions** between classes or objects, without specifying the final application classes or objects that are involved
- GOF : ***Design Patterns: Elements of Reusable Object-Oriented Software***

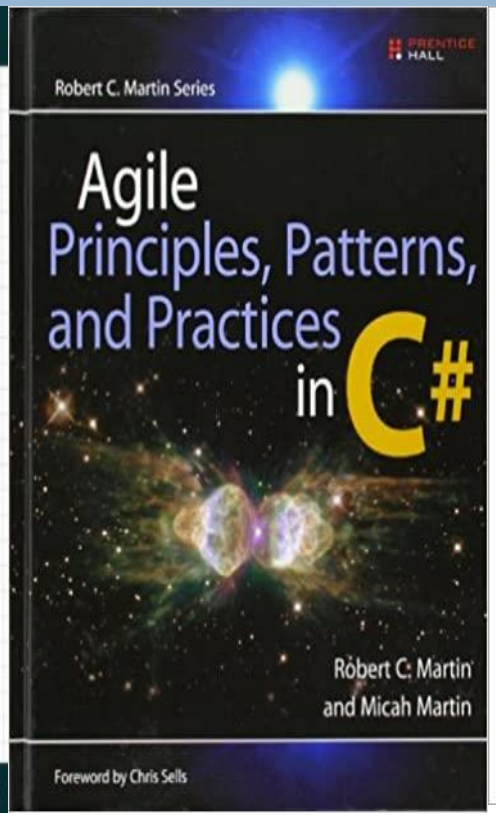
Design Patterns

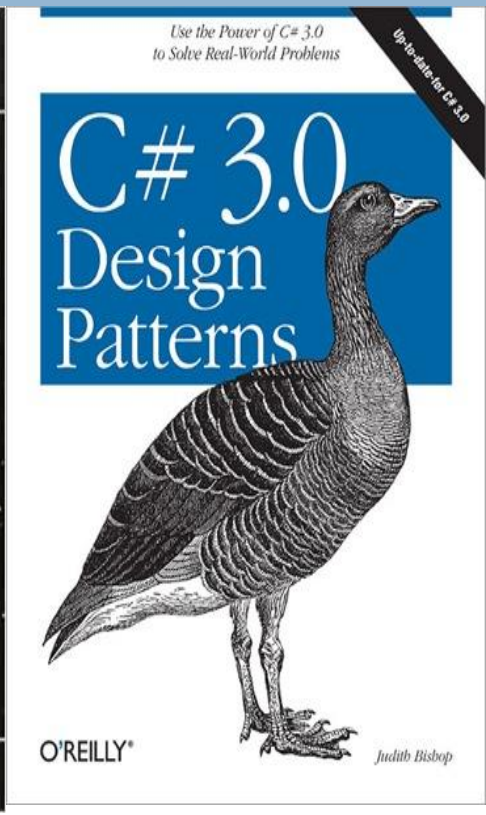
- A Design pattern consists of various co-operating objects (classes, relationships etc)".
- They provide solutions for common design problems. More than anything else, they offer a consistent idiom for designers and programmers to speak about their design.
- A Design Pattern is not code. It is in fact an approach or a model that can be used to solve a problem.
- Design Patterns are about design and interaction of objects and they provide reusable solutions for solving common design problems.

A Must Have Books









Creational Patterns (5)

Object Creation mechanisms

- **Abstract Factory** : Creates an instance of several families of classes
- **Factory Method :** Define an interface for creating an object, but let subclasses decide which class to instantiate.
- **Builder** : Separates object construction from its representation
- **Prototype:** A fully initialized instance to be copied or cloned
- **Singleton:** A class of which only a single instance can exist

Structural Patterns (7)

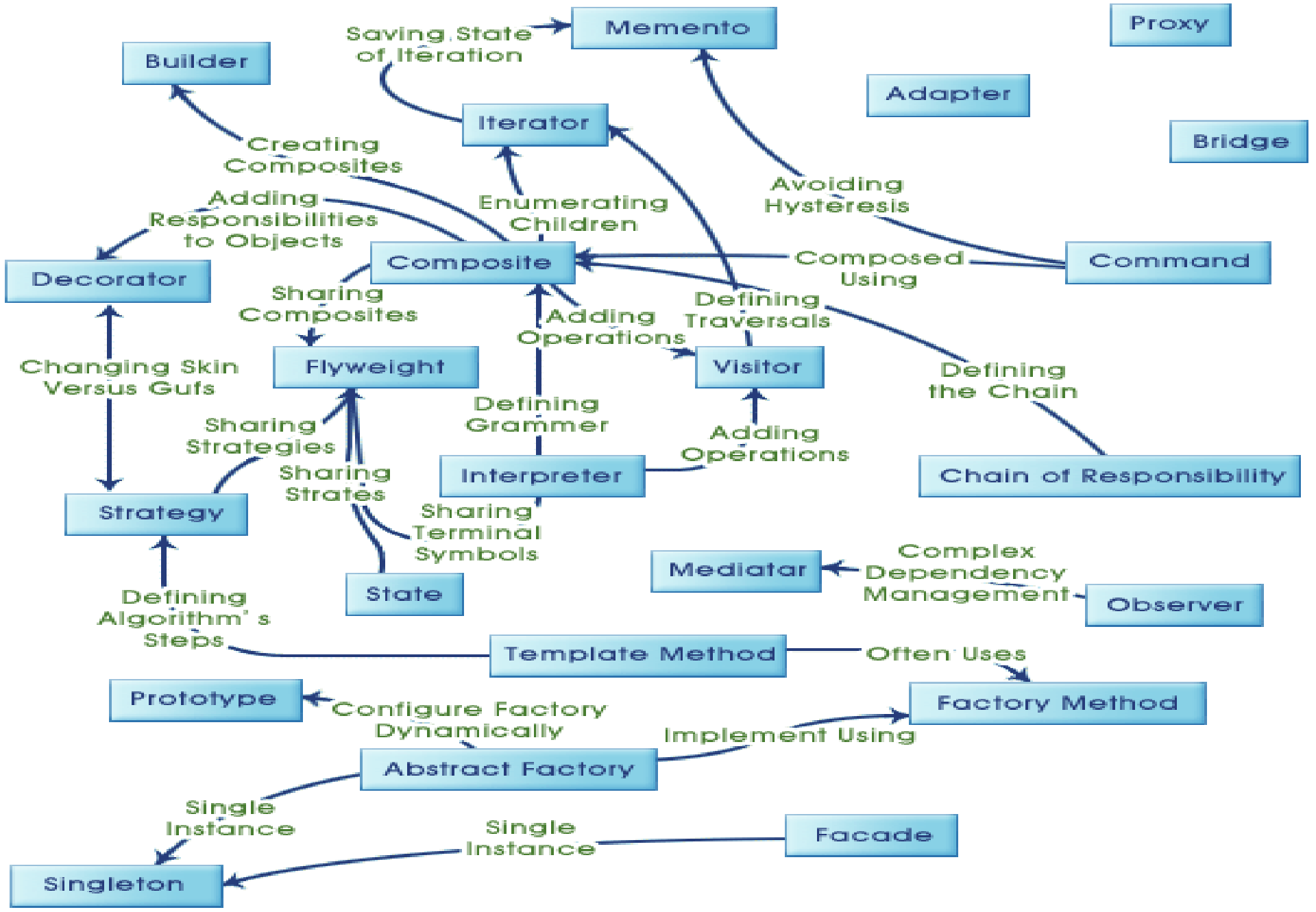
Relationships between Entities

- **Adapter** : Match interfaces of different classes
- **Bridge:** Separates an object's interface from its implementation
- **Composite:** A tree structure of simple and composite objects
- **Decorator:** Add responsibilities to objects dynamically
- **Façade:** A single class that represents an entire subsystem
- **Flyweight:** A fine-grained instance used for efficient sharing
- **Proxy:** An object representing another object

Behavioral Patterns (11)

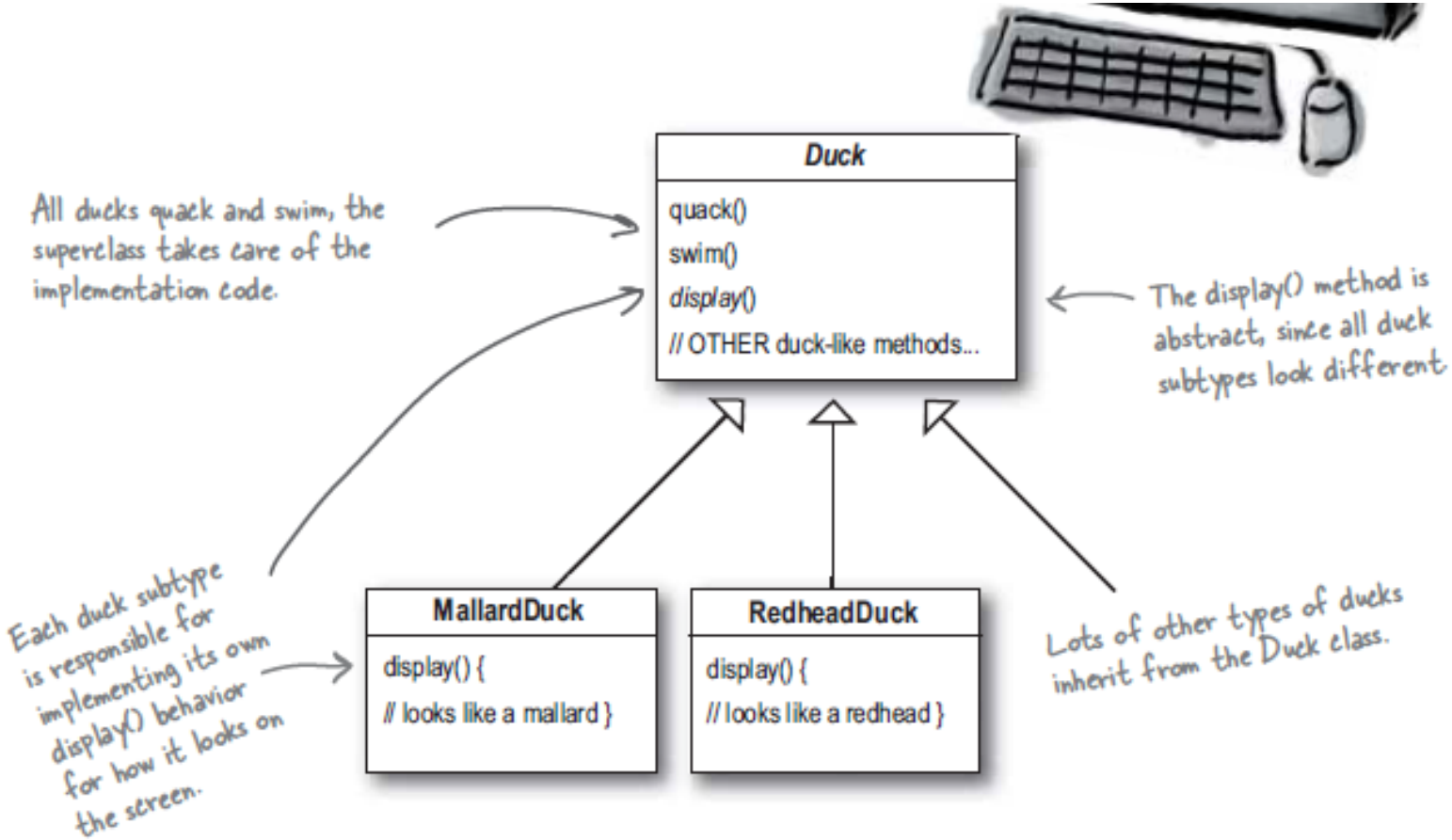
Communication Patterns

- **Chain of Responsibility:** A way of passing a request between a chain of objects
- **Command** : Encapsulate a command request as an object
- **Interpreter:** A way to include language elements in a program
- **Iterator:** Sequentially access the elements of a collection
- **Mediator** : Defines simplified communication between classes
- **Memento:** Capture and restore an object's internal state
- **Observer:** A way of notifying change to a number of classes
- **State:** Alter an object's behavior when its state changes
- **Strategy:** Encapsulates an algorithm inside a class
- **Template Method:** Defer the exact steps of an algorithm to a subclass
- **Visitor:** Defines a new operation to a class without change

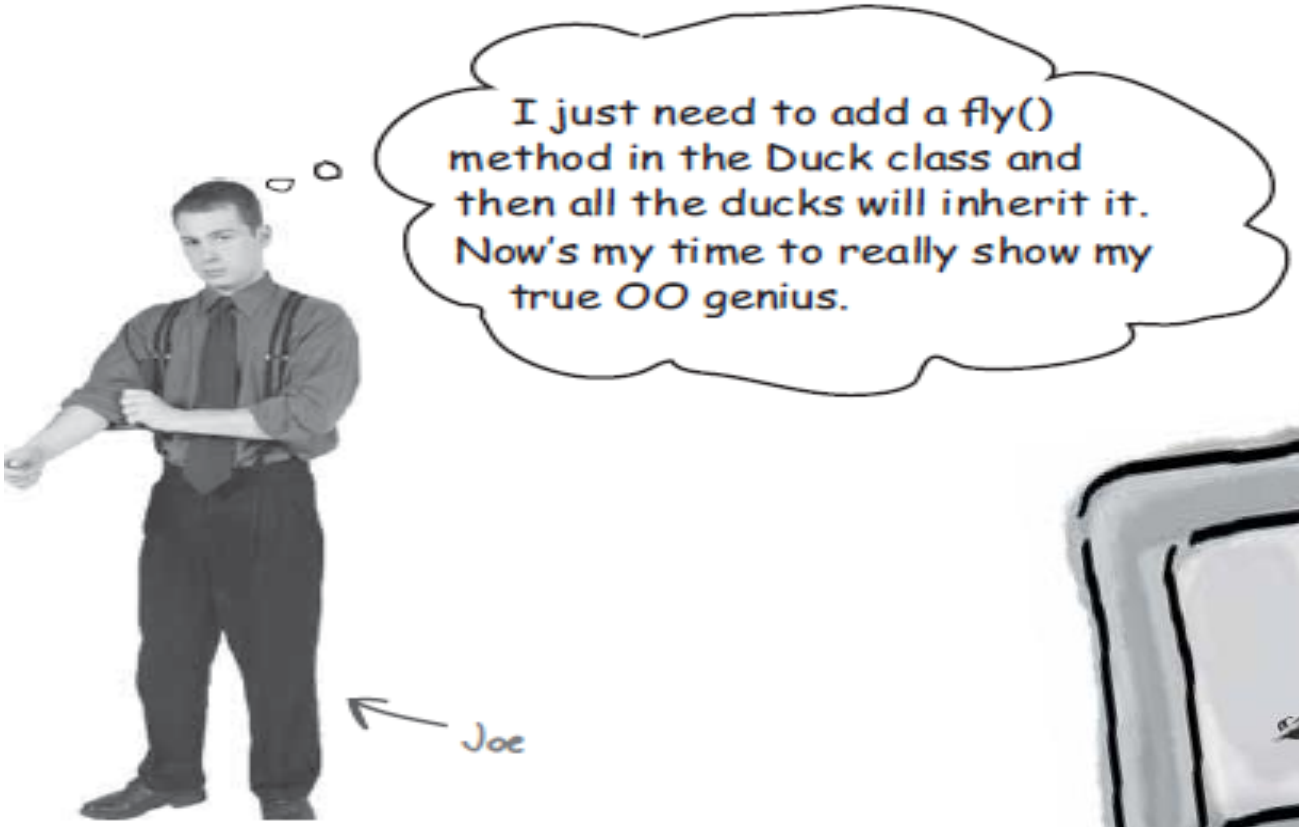

Memento
Proxy
Saving State of Iteration
Builder
Adapter
Iterator
Bridge
Creating Composites
Avoiding Hysteresis
Adding Responsibilities to Objects
Enumerating Children
Composite
Composed Using
Command
Decorator
Sharing Composites
Adding Operations
Defining Traversals
Flyweight
Visitor
Changing Skin Versus Guts
Defining the Chain
Defining Grammer
Sharing Strategies
Adding Operations
Sharing Strates
Interpreter
Chain of Responsibility
Strategy
Sharing Terminal Symbols
Mediatar
Complex Dependency Management
State
Observer
Defining Algorithm's Steps
Template Method
Often Uses
Prototype
Configure Factory Dynamically
Factory Method
Implement Using
Abstract Factory
Single Instance
Single Instance
Facade
Singleton

Let The Patterns Start

It all Started with Ducks

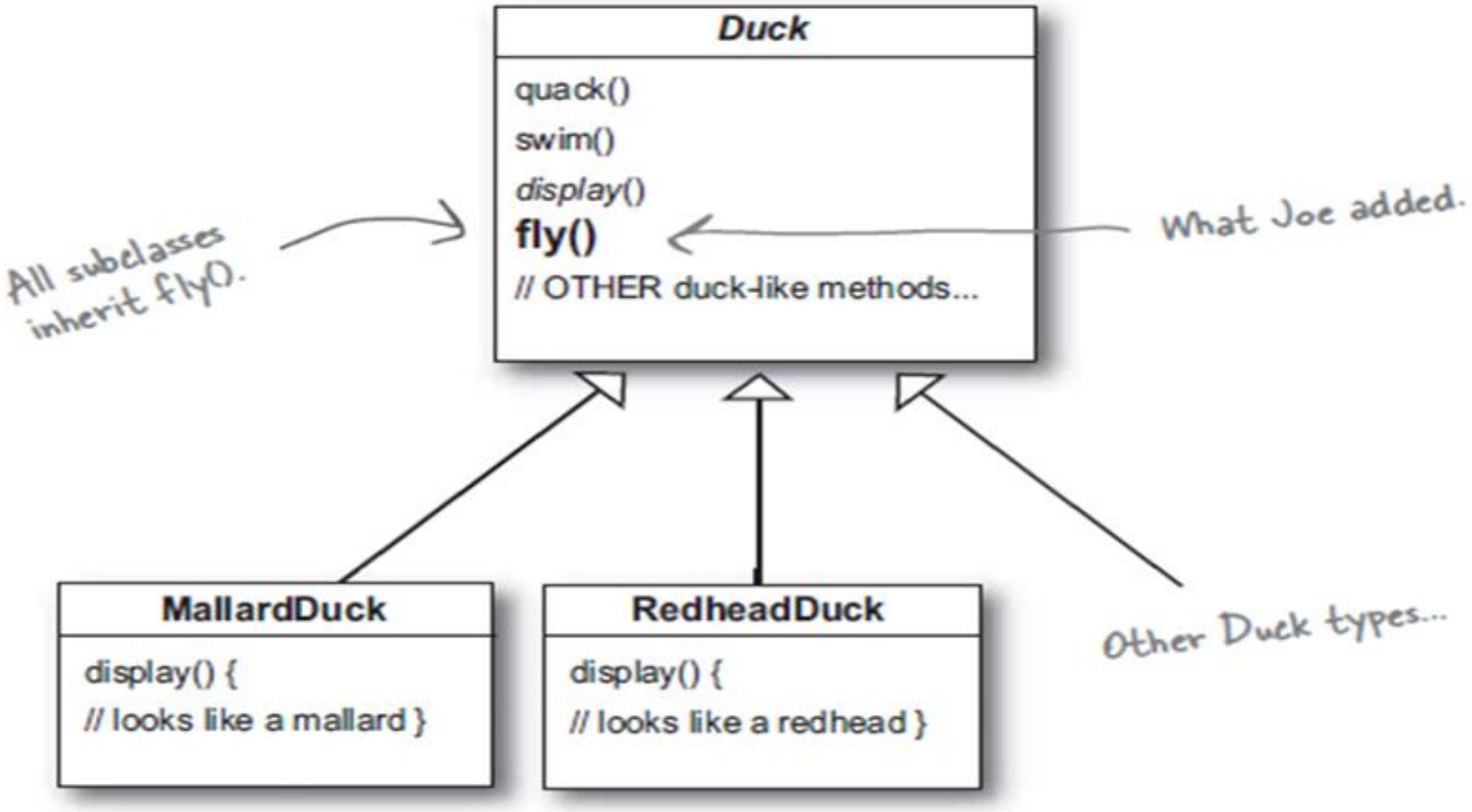


But We Need The Ducks To Fly!!!

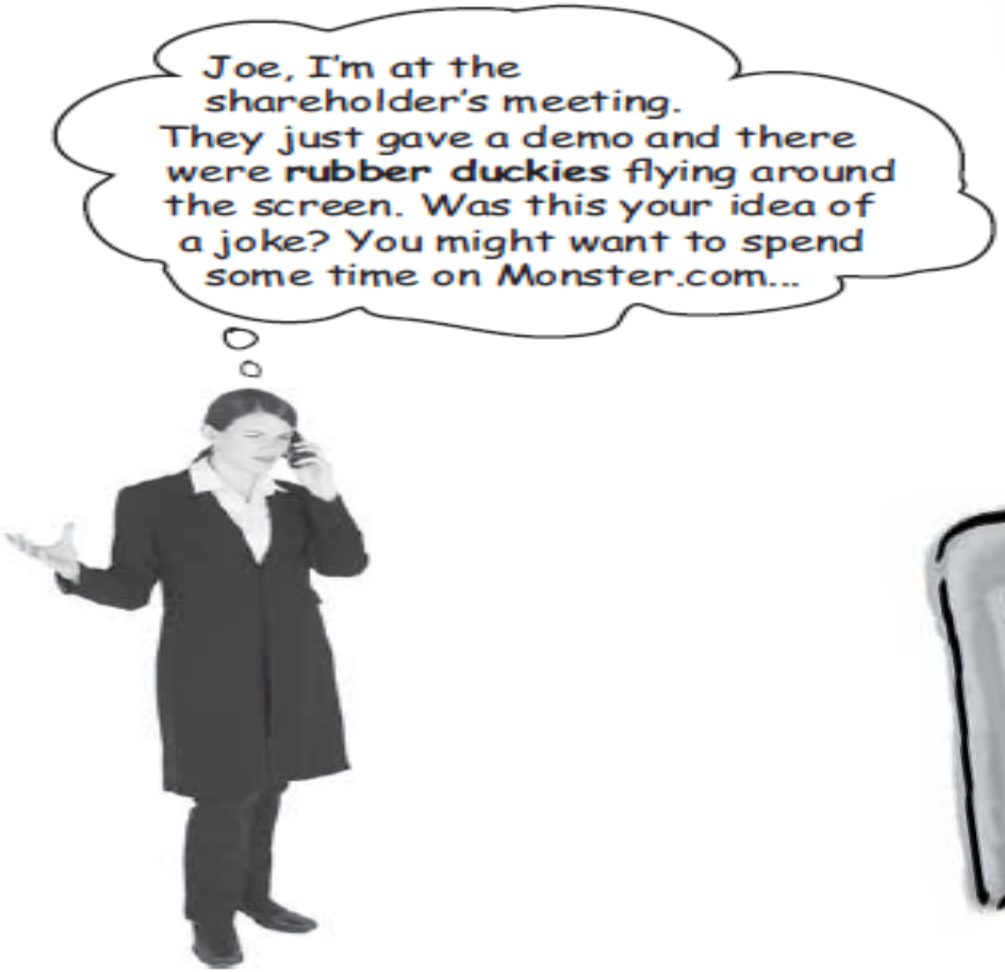


"he's an OO programmer... *how hard can it be?*"

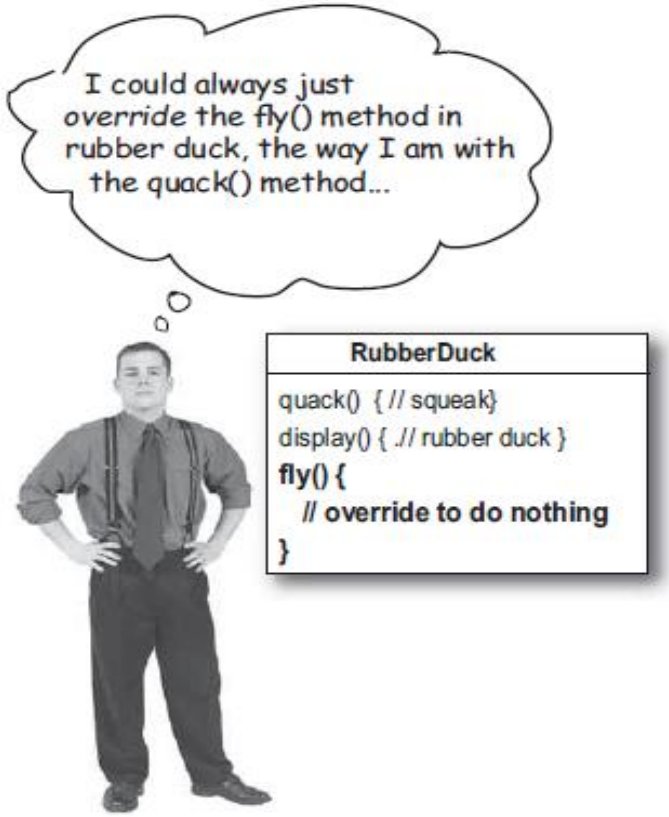
It Really Does Fly!!!!



What Does Really Happened



Maintenance Maintenance Maintenance …

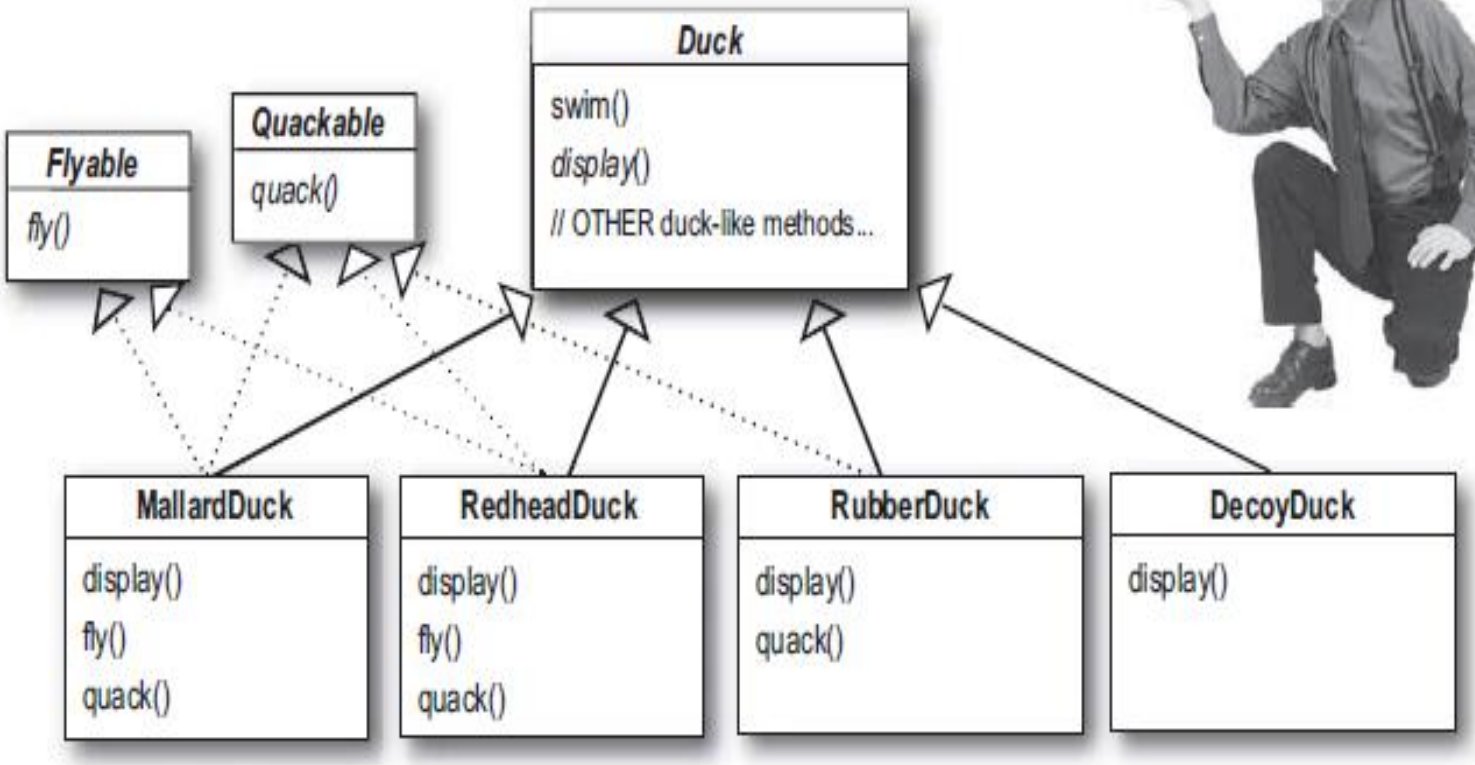


DecoyDuck

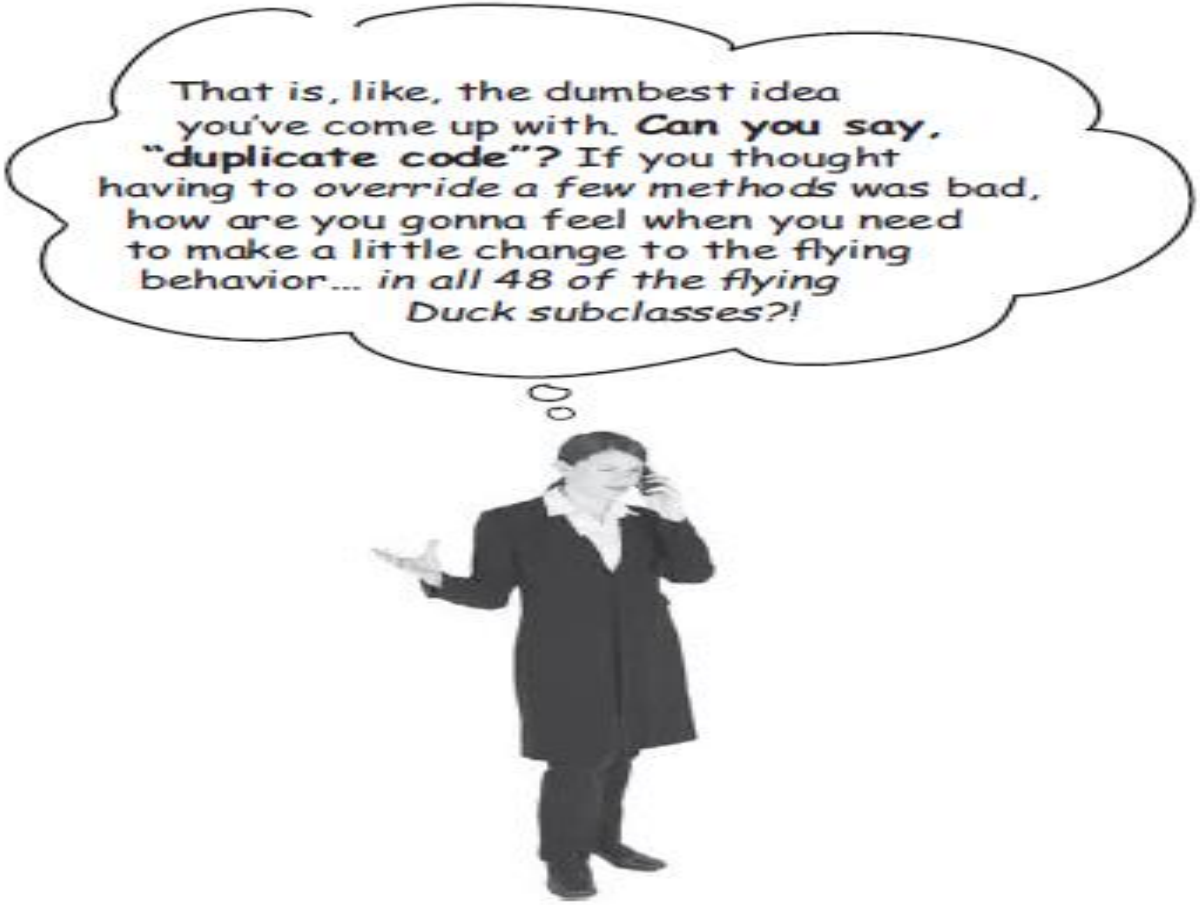
```
quack() {
  // override to do nothing
}

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.

How About An Interface



Not Again !!!!


That is, like, the dumbest idea you've come up with. Can you say, "duplicate code"? If you thought having to override a few methods was bad, how are you gonna feel when you need to make a little change to the flying behavior... in all 48 of the flying Duck subclasses?!

I Have a Dream

The One Constant in Software Development is

Change

The Solution

Design Principle

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

Take what varies and "encapsulate" it so it won't affect the rest of your code.

Here's another way to think about this principle: ***take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don't.***

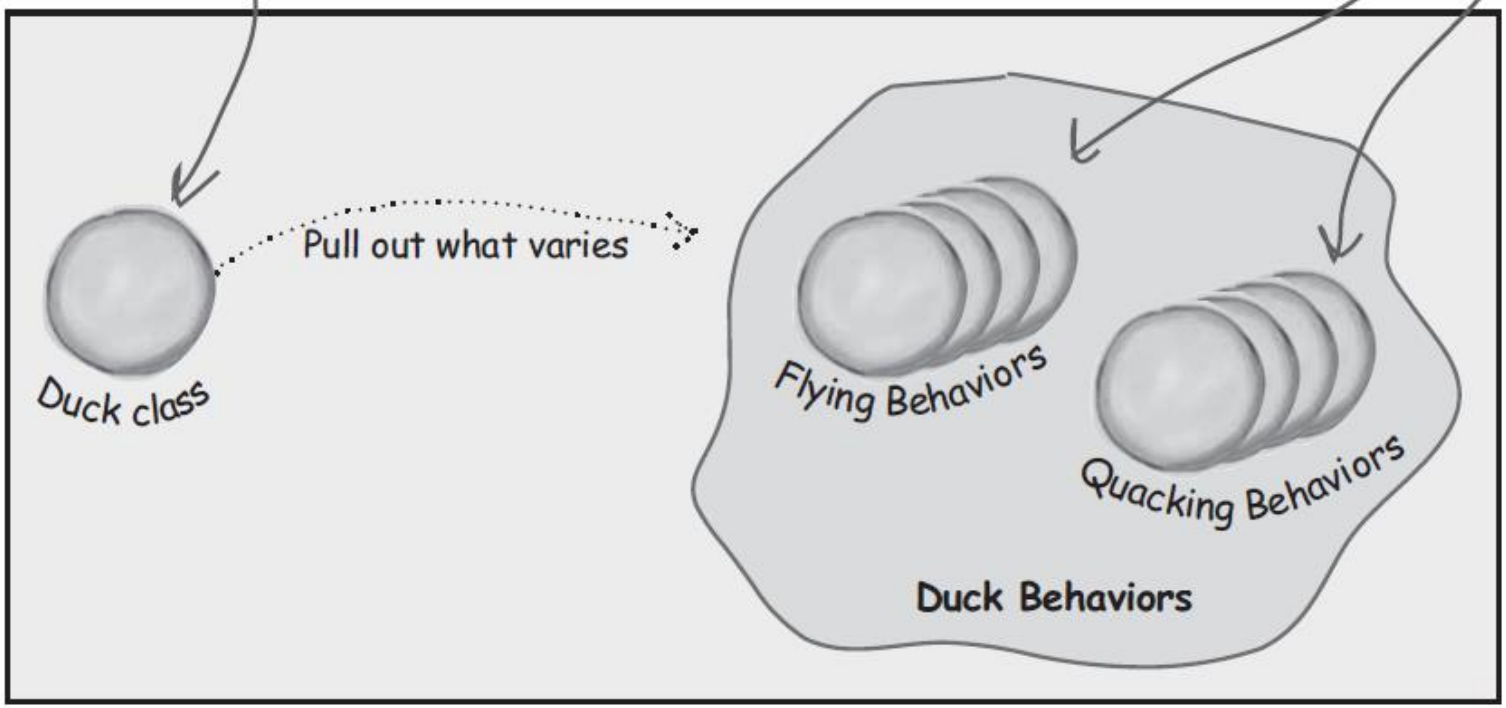
As simple as this concept is, it forms the basis for almost every design pattern. All patterns provide a way to let *some part of a system vary independently of all other parts.*

Separating what changes from what stays the same

The Duck class is still the superclass of all ducks, but we are pulling out the fly and quack behaviors and putting them into another class structure.

Now flying and quacking each get their own set of classes.

Various behavior implementations are going to live here.



Design The Behavior

- Program to an interface not an implementation
- We want to *assign* a behavior to the instance of Duck
- We Should include a Behavior setter methods in the duck class so we can *change* the Duck Behavior through runtime
- We Will implement a set of classes whose entire reason for living is to represent a Behavior
- The Point is to exploit Polymorphism by programming to as super type so that the actual runtime object isn't locked into the code

Design Principle

Program to an interface, not an implementation.

Before and After DP

Before

- The behavior Came from a concrete implementation in the Super Class Duck or a specialized implementation in the Sub Class Itself

From now on, the Duck behaviors will live in a separate class—a class that implements a particular behavior interface.

That way, the Duck classes won't need to know any of the implementation details for their own behaviors.

Interface Or Abstract Class

- Program to interface , Means Program to *Super Type*
- The Declared Type of the variables should be a *Super Type* usually an abstract class or an Interface so that the objects assigned to these variables can be of any concrete implementation of the super type
- The class Declaring the variable doesn't have to know about the actual object types

Programing to an Interface

Programming to an implementation would be:

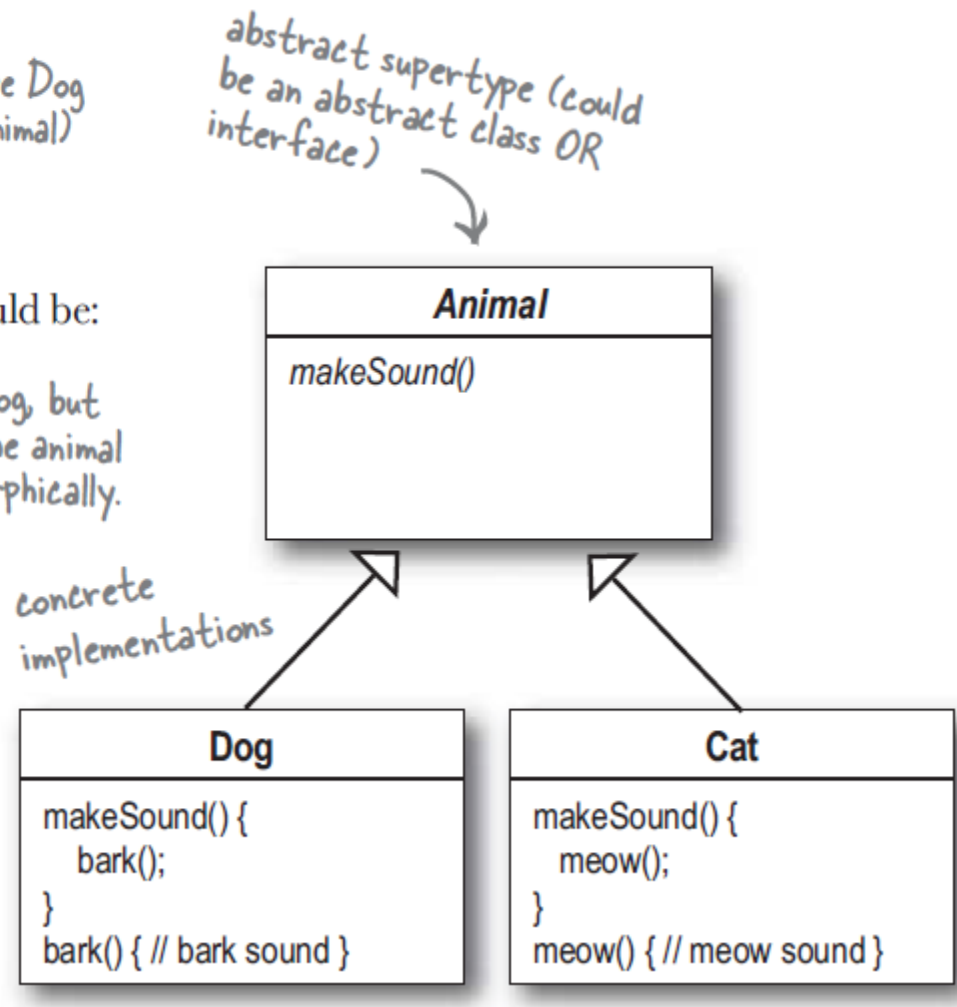
```
Dog d = new Dog();
d.bark();
```

Declaring the variable "d" as type Dog (a concrete implementation of Animal) forces us to code to a concrete implementation.

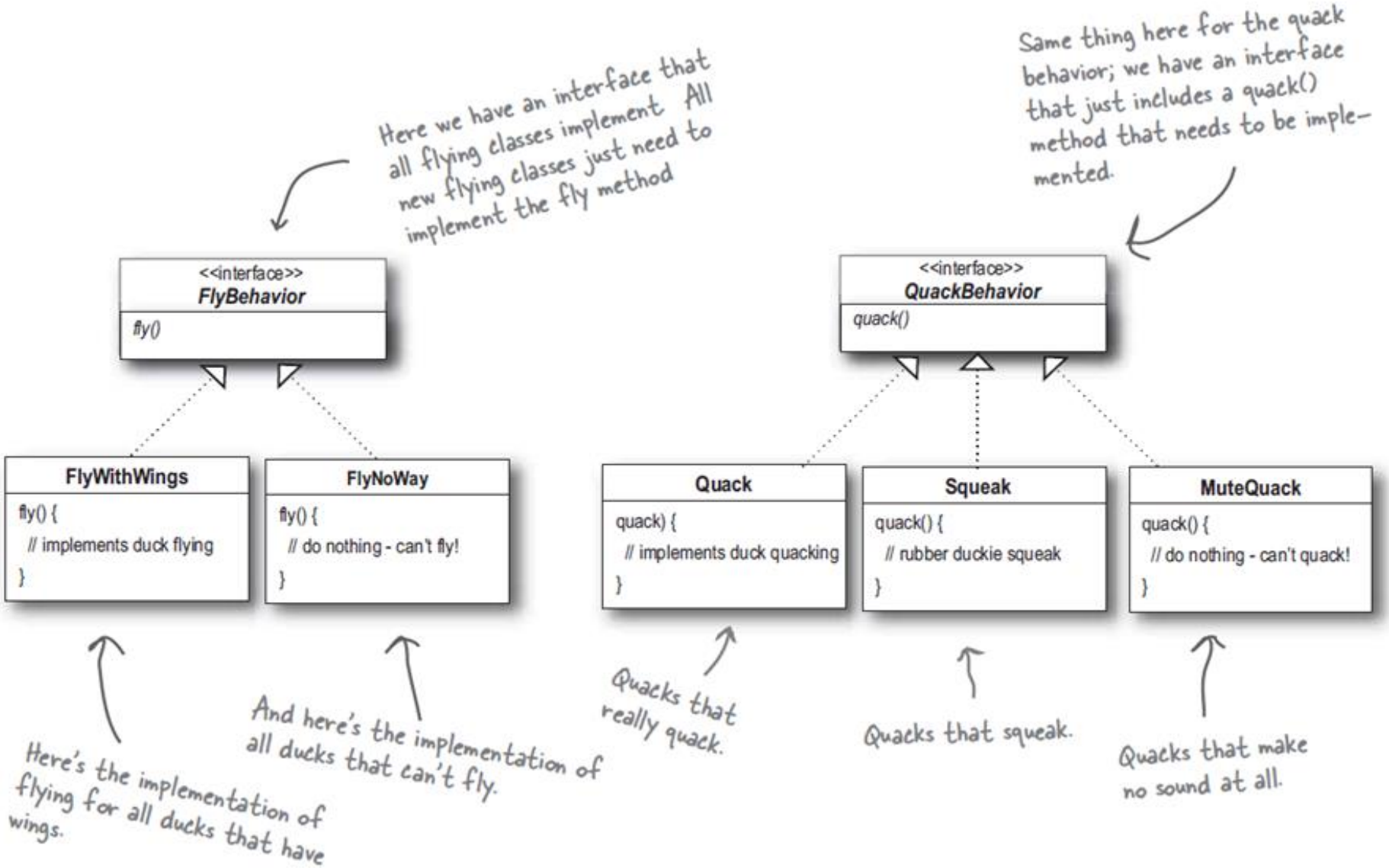
But **programming to an interface/supertype** would be:

```
Animal animal = new Dog();
animal.makeSound();
```

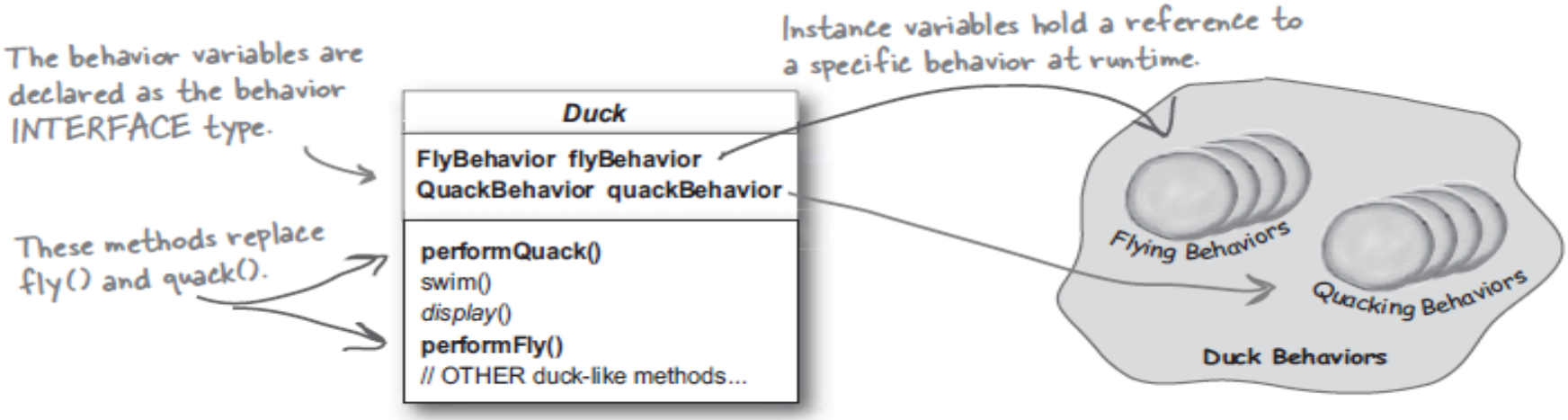
We know it's a Dog, but we can now use the animal reference polymorphically.



Duck Behaviors



The Duck

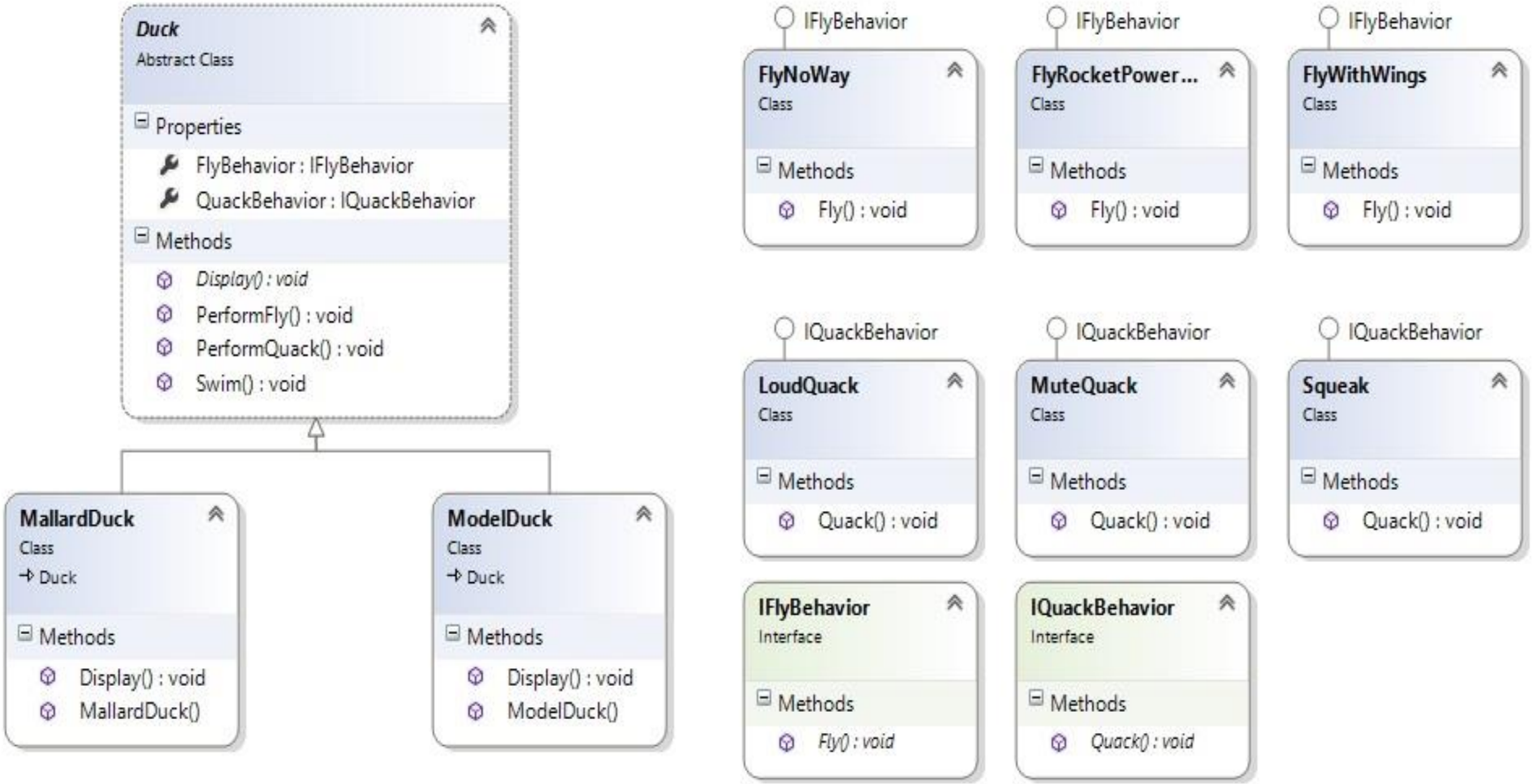



```
public class Duck {
    QuackBehavior quackBehavior;
    // more

    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.

Class Diagram



Duck Class

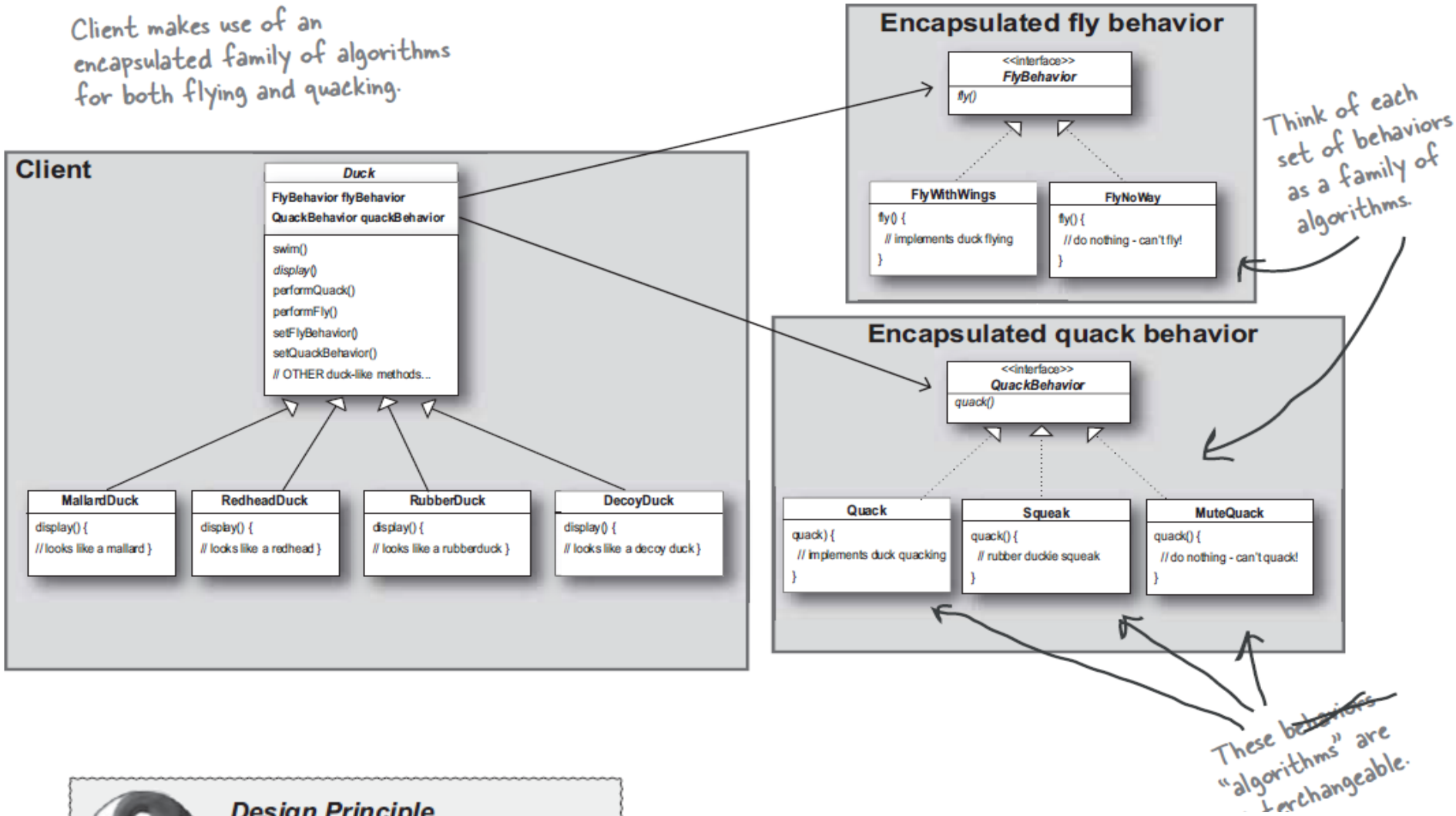
```
public abstract class Duck
    {
        public IFlyBehavior FlyBehavior { get; set; }
        public IQuackBehavior QuackBehavior { get; set; }
        public abstract void Display();

        public void PerformFly()
        {             FlyBehavior.Fly();        }
        public void PerformQuack()
        {             QuackBehavior.Quack();        }
        public void Swim()
        {Console.WriteLine("All ducks float, even decoys!");}
    }
```

Mallard Duck

```
public class MallardDuck : Duck
    {
        public MallardDuck()
        {
            QuackBehavior = new LoudQuack();
            FlyBehavior = new FlyWithWings();
        }

        override public void Display()
        {
            Console.WriteLine("I'm a real Mallard duck");
        }
    }
```

HAS-A can be better than IS-A

Client makes use of an encapsulated family of algorithms for both flying and quacking.



Think of each set of behaviors as a family of algorithms.

These ~~behaviors~~ "algorithms" are interchangeable.

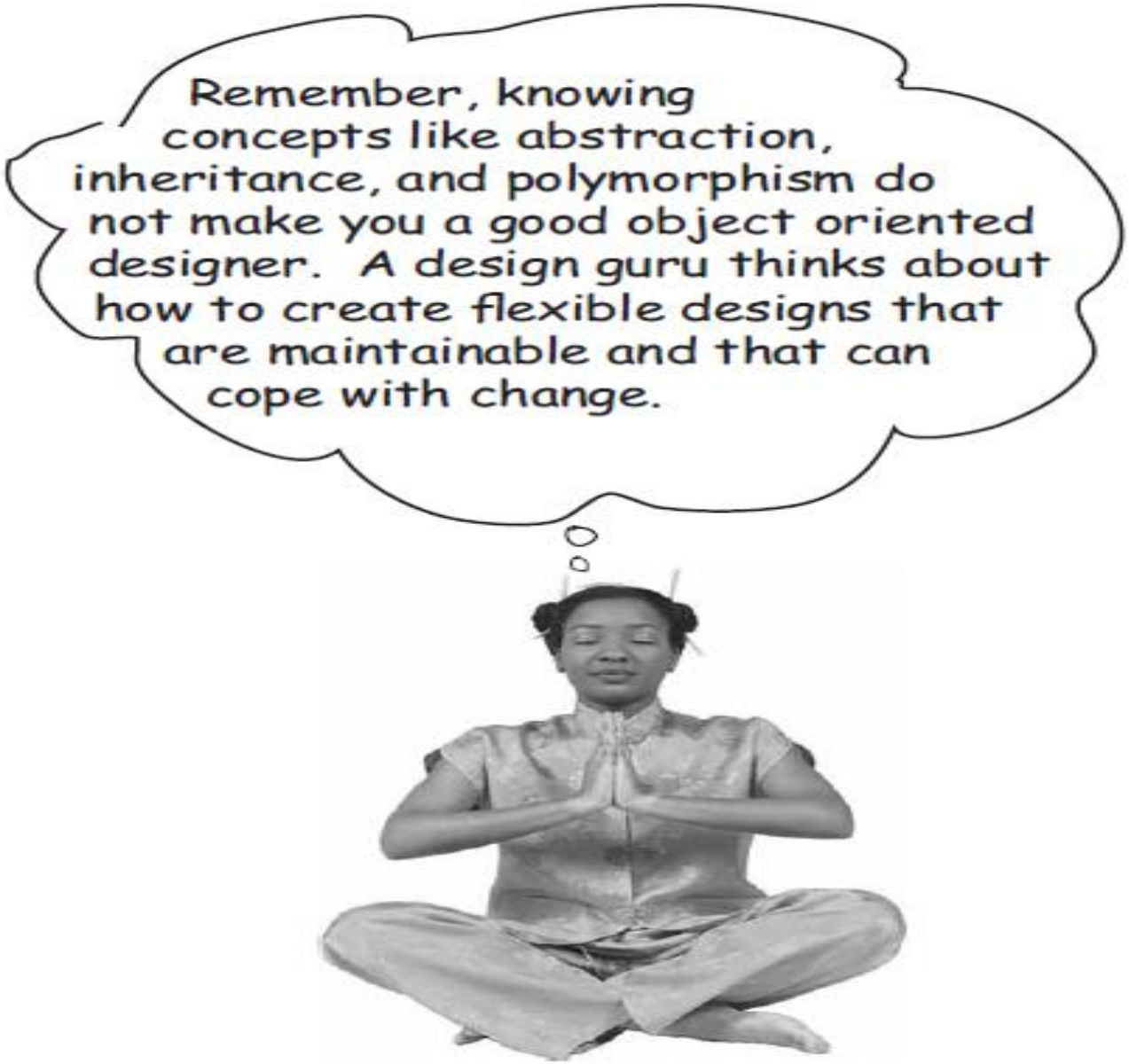
Design Principle

Favor composition over inheritance.

The HAS-A relationship is an interesting one: each duck has a FlyBehavior and a QuackBehavior to which it delegates flying and quacking.

Strategy Pattern:

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


Remember, knowing concepts like abstraction, inheritance, and polymorphism do not make you a good object oriented designer. A design guru thinks about how to create flexible designs that are maintainable and that can cope with change.

Open Closed Principle

- Software Entities (Classes , Modules , Functions , …) should be **Open** for **Extension** But **Closed** for **Modification .**
- **Open** for **Extension :** New Behavior can be added in the Future (In other words, we are able to change what the module does)
- **Closed** for **Modification :** Change to Source or Binary code are Not required Nor Permitted (No Compilation of Old Code is Required)

The Answer

- Rely on Abstraction
- C# Abstraction can be Achieved Via These Approaches
  - Abstract Classes
  - Interfaces
  - Procedural Programming Approach (Delegate & Lambda)

- If Client Class uses the Server Class ,and we want the Client to uses Different Server we need to change the Code .

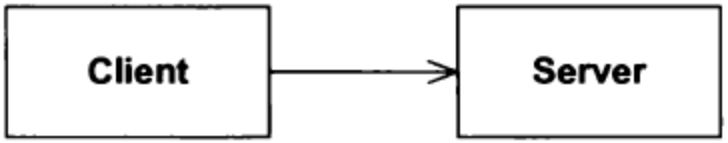
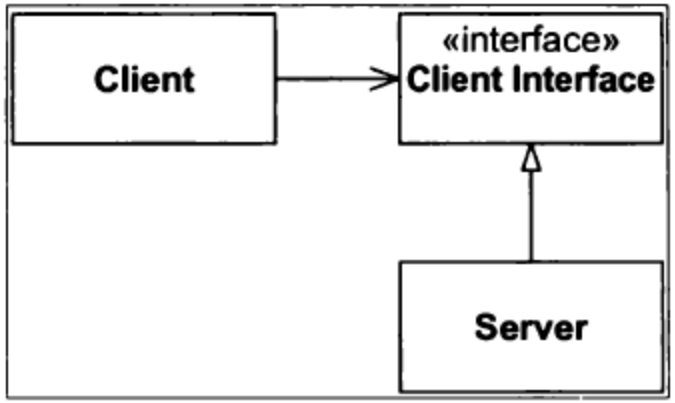


Figure 9-1 Client is not open and closed

The Answer

- Strategy Pattern
- Depend on Abstraction ,
- Pass in Implementation



SOLID Principle

http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

One last Word

OO Basics

Abstraction
Encapsulation
Polymorphism
Inheritance

We assume you know the OO basics of using classes polymorphically, how inheritance is like design by contract, and how encapsulation works. If you are a little rusty on these, pull out your Head First Java and review, then skim this chapter again.

OO Principles

Encapsulate what varies.
Favor composition over inheritence.
Program to interfaces, not implementations.

We'll be taking a closer look at these down the road and also adding a few more to the list

OO Patterns

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

Throughout the book think about how patterns rely on OO basics and principles.

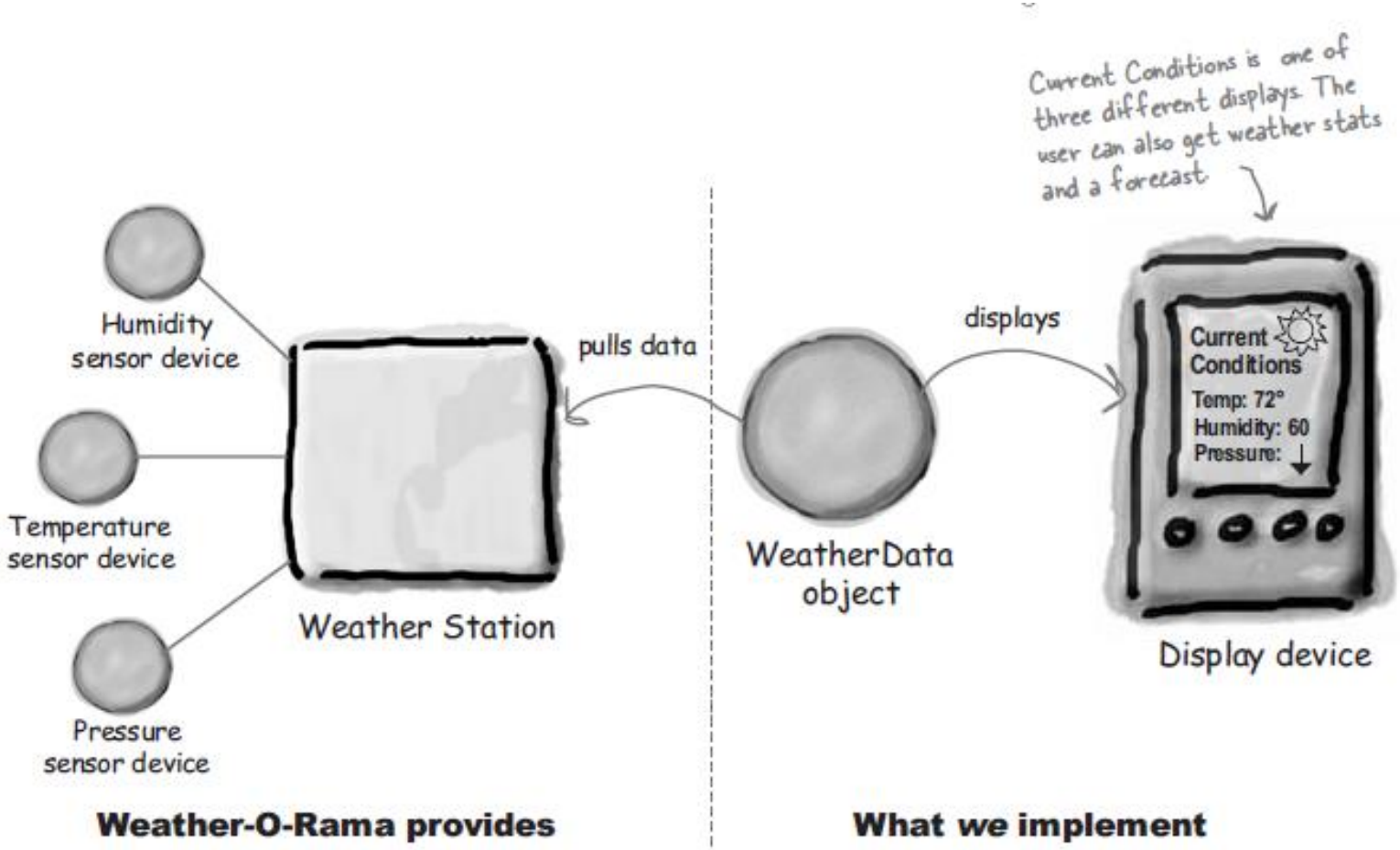
One down, many to go!

Final word

- Patterns aren't *invented*, they are *discovered*.
- Most patterns and principles address issues of *change* in software.
- Most patterns allow some part of a system to vary independently of all other parts.
- We often try to take what varies in a system and encapsulate it.

the Observer Pattern

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



What We Know

WeatherData

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

// other methods

These three methods return the most recent weather measurements for temperature, humidity and barometric pressure respectively.

We don't care HOW these variables are set; the WeatherData object knows how to get updated info from the Weather Station.

```
/*
 * This method gets called
 * whenever the weather measurements
 * have been updated
 *
 */
public void measurementsChanged() {
    // Your code goes here
}
```

The developers of the WeatherData object left us a clue about what we need to add--

Remember, this Current Conditions is just ONE of three different display screens.

Display device

Our job is to implement measurementsChanged() so that it updates the three displays for current conditions, weather stats, and forecast.

First Approach

```
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
}
```

Grab the most recent measuremets by calling the WeatherData's getter methods (already implemented).

Now update the displays...

Call each display element to update its display, passing it the most recent measurements.

What is Wrong?

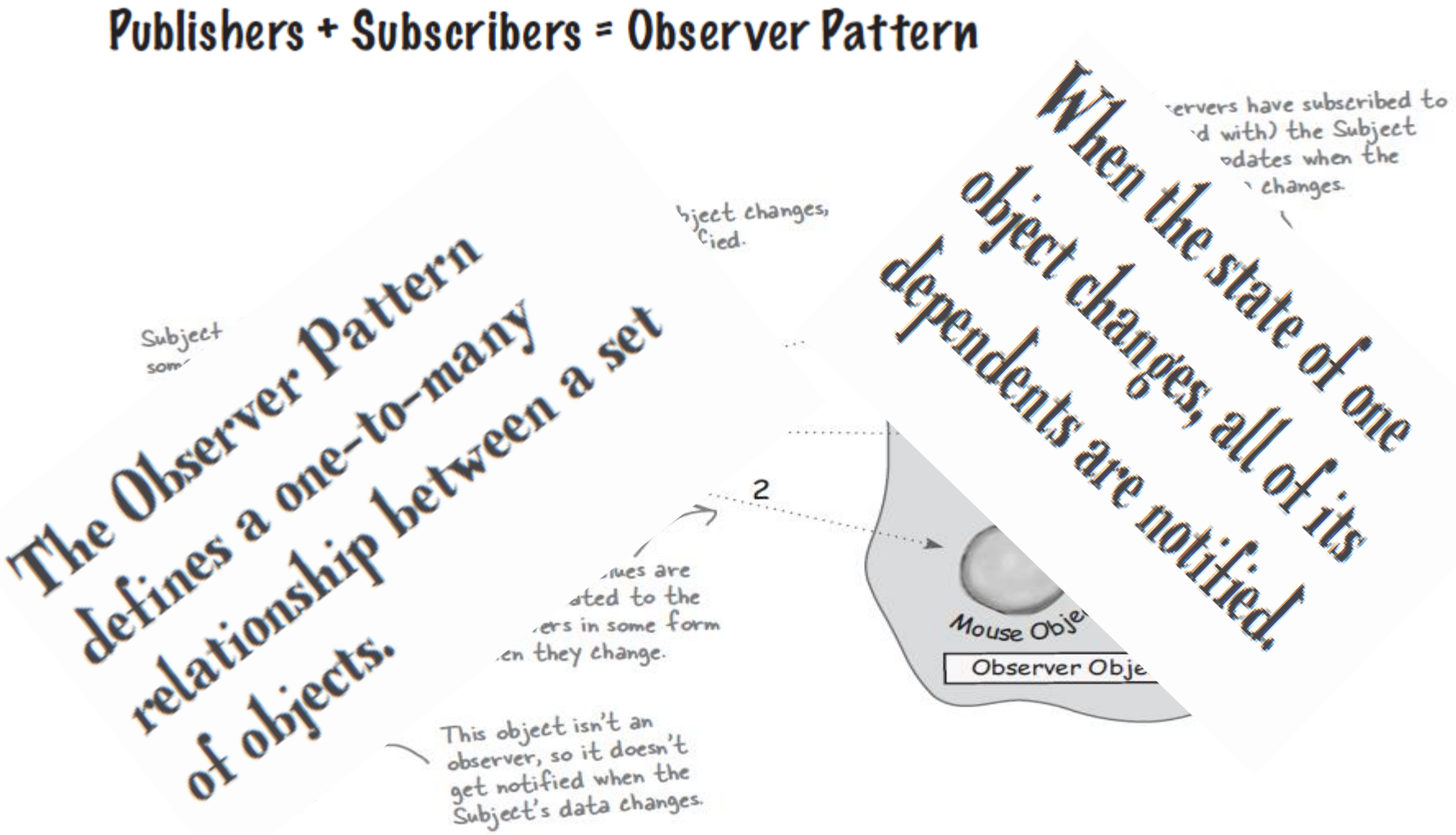
```
public void measurementsChanged() {

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

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

Area of change, we need to encapsulate this.

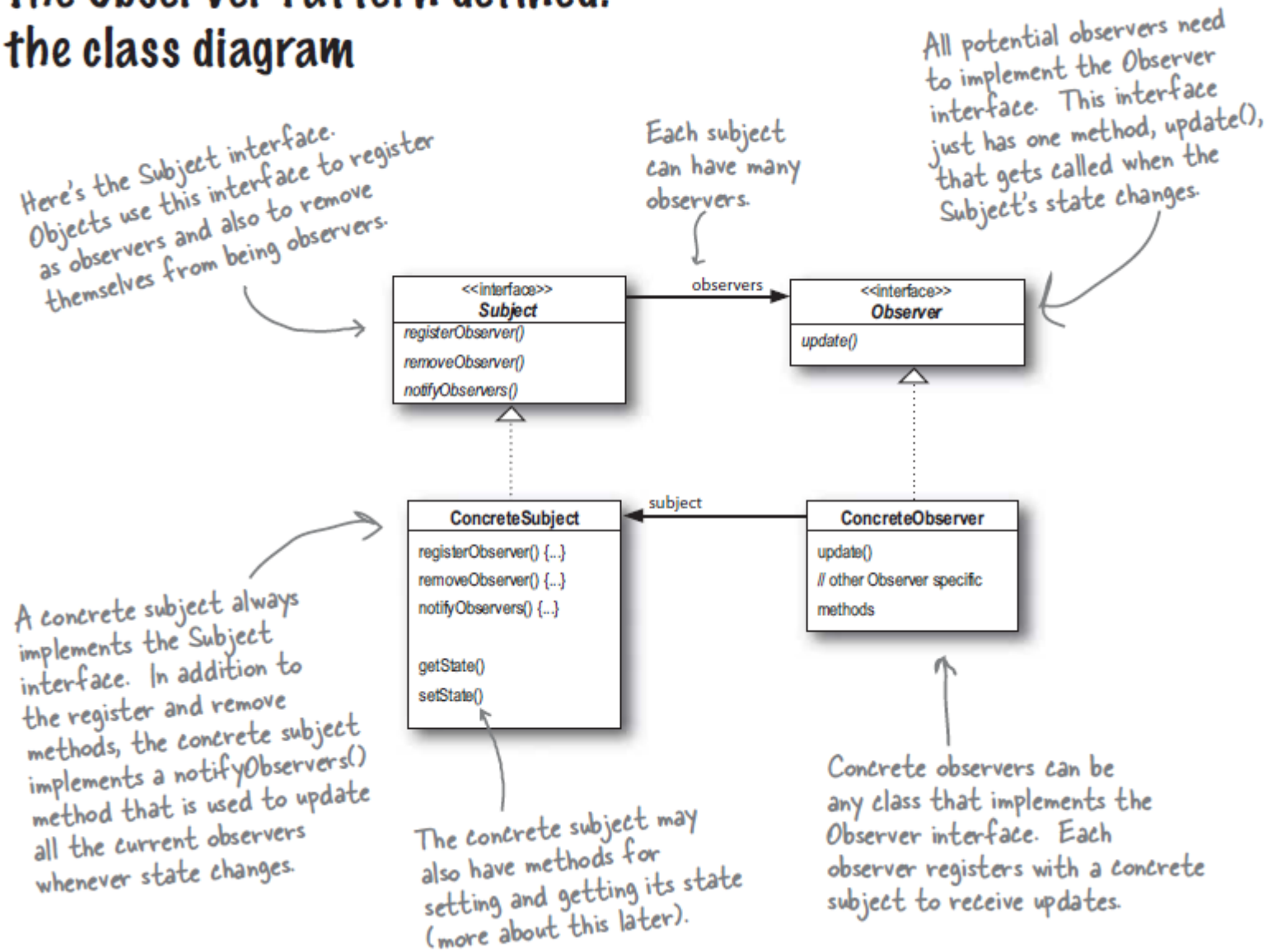
At least we seem to be using a common interface to talk to the display elements... they all have an update() method takes the temp, humidity, and pressure values.

By coding to concrete implementations we have no way to add or remove other display elements without making changes to the program.

Publisher Subscriber Model



The Observer Pattern defined: the class diagram



Loose Coupling

The power of Loose Coupling

When two objects are loosely coupled, they can interact, but have very little knowledge of each other.

The Observer Pattern provides an object design where subjects and observers are loosely coupled.

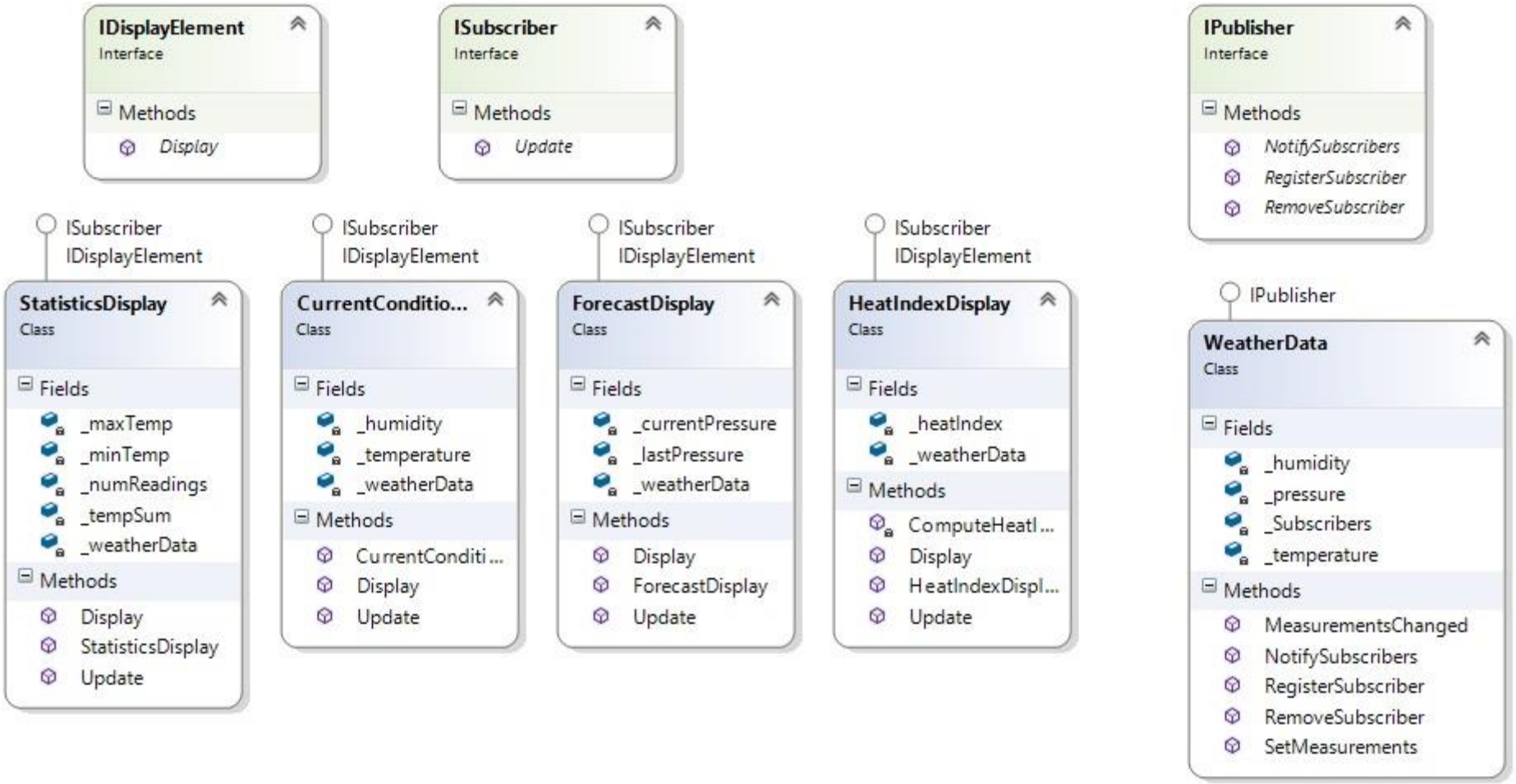
The only thing the subject knows about an observer is that it implements a certain interface (the Observer interface). It doesn't need to know the concrete class of the observer, what it does, or anything else about it.

Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the interdependency between objects.

Pros & Cons

- ***Abstract coupling between Subject and Observer***. All a subject knows is that it has a **list of observers**, each conforming to the **simple interface** of the abstract Observer class. The subject doesn't know the concrete class of any observer. Thus the coupling between subjects and observers is abstract and minimal.
- ***Support for broadcast communication***. Unlike an ordinary request, the notification that a subject sends needn't specify its receiver. The notification is broadcast automatically to all interested objects that subscribed to it. The subject doesn't care how many interested objects exist; its only responsibility is to notify its observers. This gives you the freedom to add and remove observers at any time. It's up to the observer to handle or ignore a notification.
- ***Unexpected updates***. Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject. A seemingly innocent operation on the subject may cause a cascade of updates to observers and their dependent objects. Moreover, dependency criteria that aren't well-defined or maintained usually lead to spurious updates, which can be hard to track down.

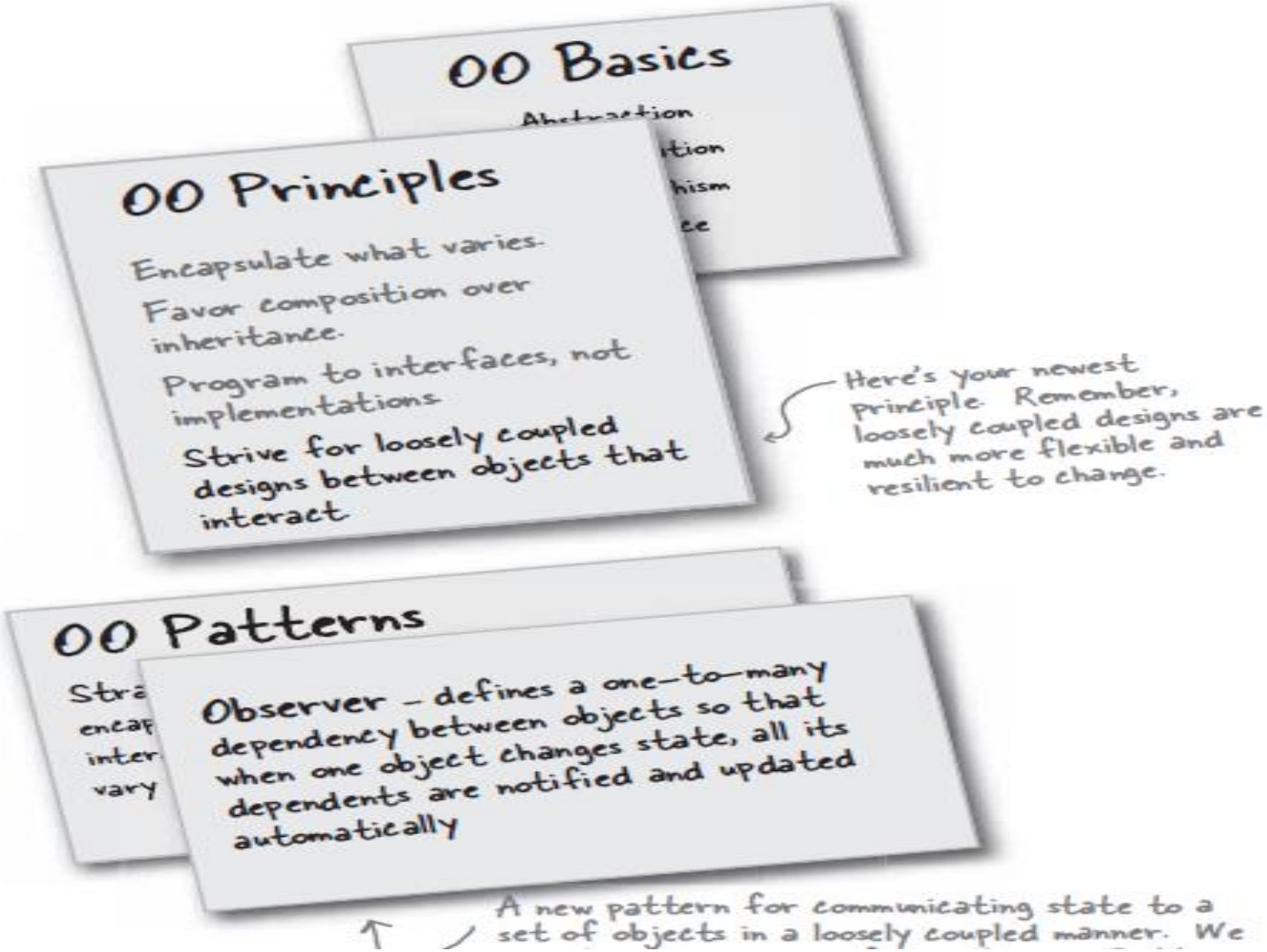
Class Diagram



Dot Net Observer Built in Pattern

- Delegates
- Events
- Event Arguments
- UI Controls

Last Word


OO Basics
OO Principles
Encapsulate what varies.
Favor composition over inheritance.
Program to interfaces, not implementations.
Strive for loosely coupled designs between objects that interact.
Here's your newest principle. Remember, loosely coupled designs are much more flexible and resilient to change.
OO Patterns
Observer – defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
A new pattern for communicating state to a set of objects in a loosely coupled manner. We

the Decorator Pattern

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

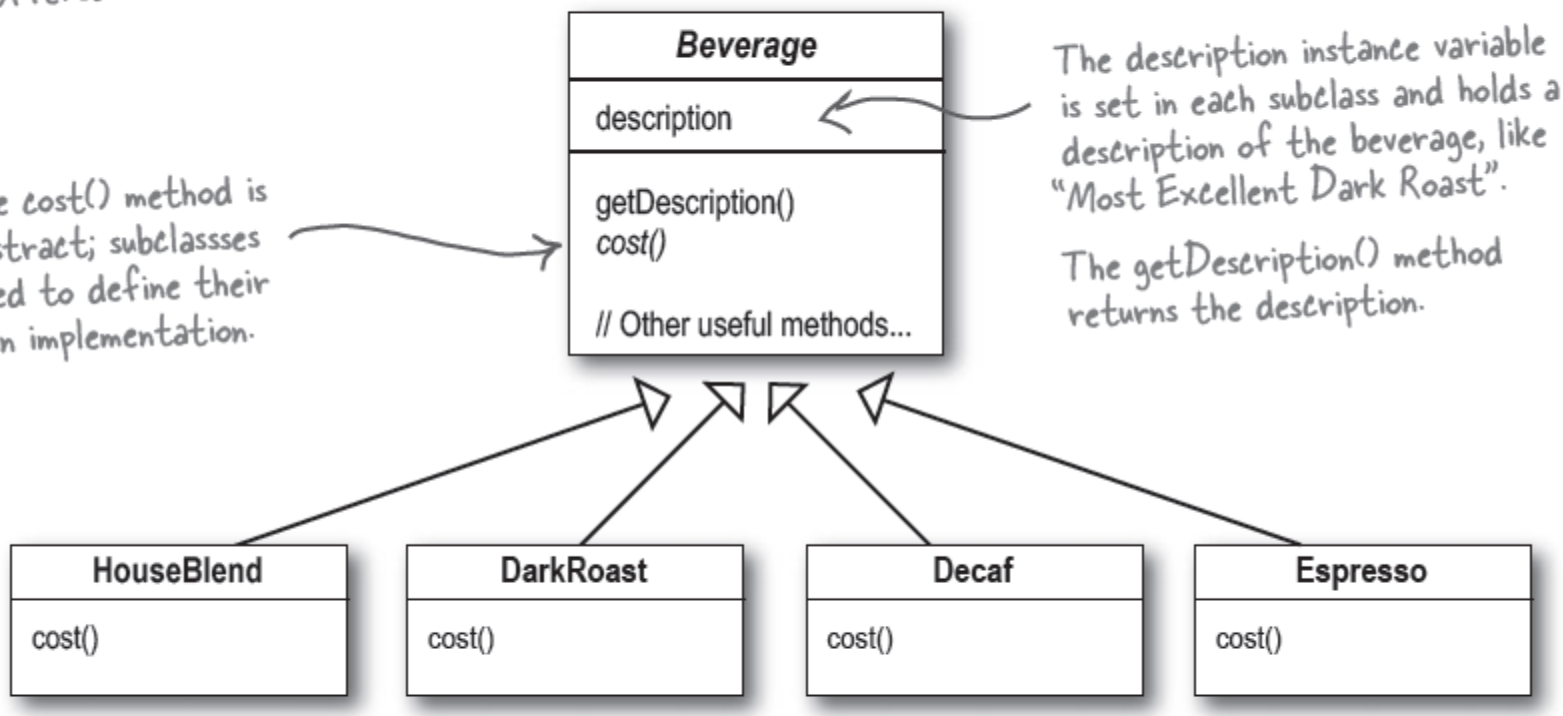
Starbuzz Coffee

Beverage is an abstract class, subclassed by all beverages offered in the coffee shop.

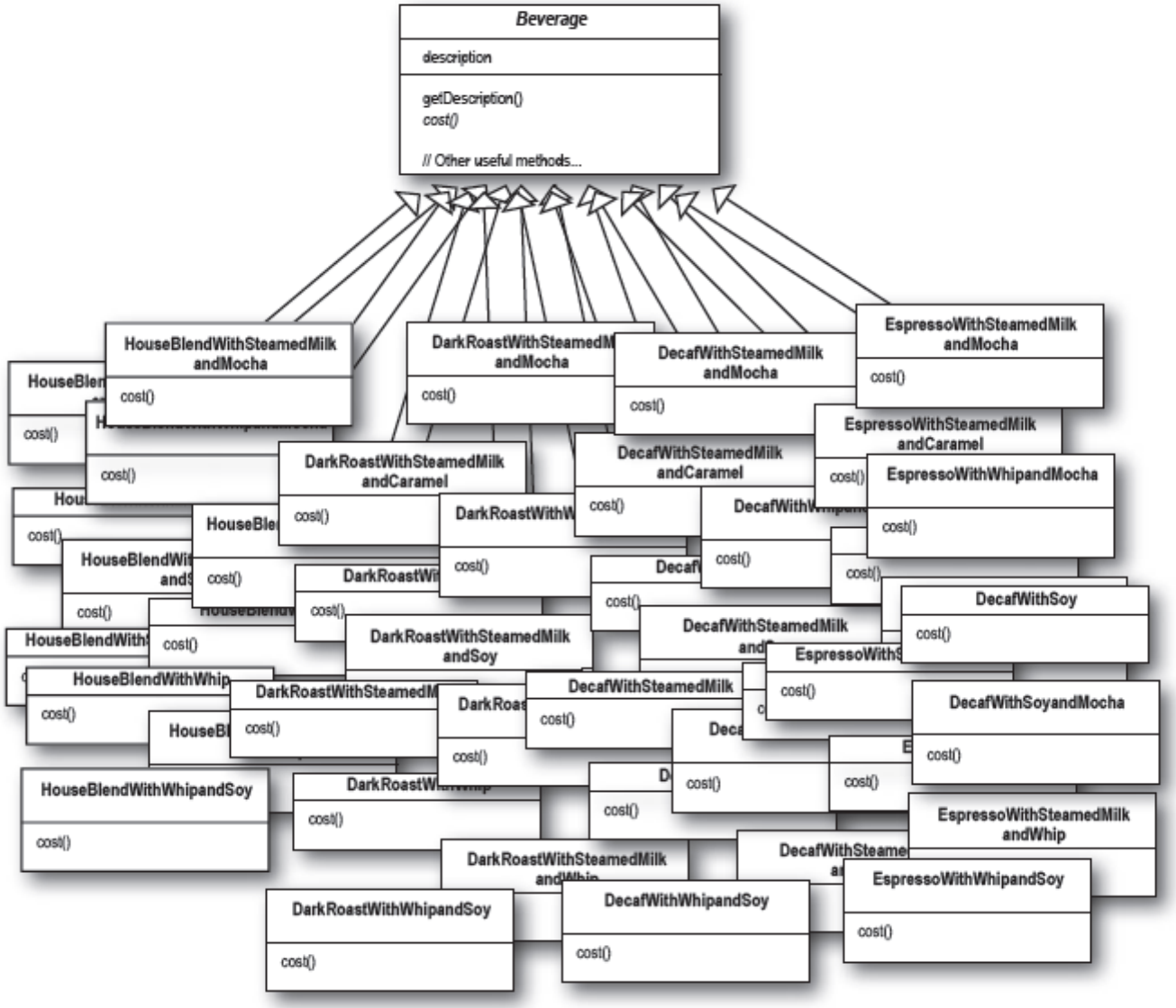
The cost() method is abstract; subclassses need to define their own implementation.

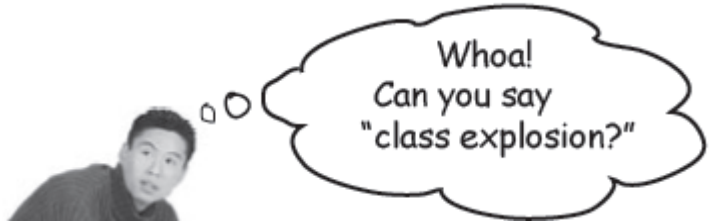
The description instance variable is set in each subclass and holds a description of the beverage, like "Most Excellent Dark Roast".

The getDescription() method returns the description.



Each subclass implements cost() to return the cost of the beverage.





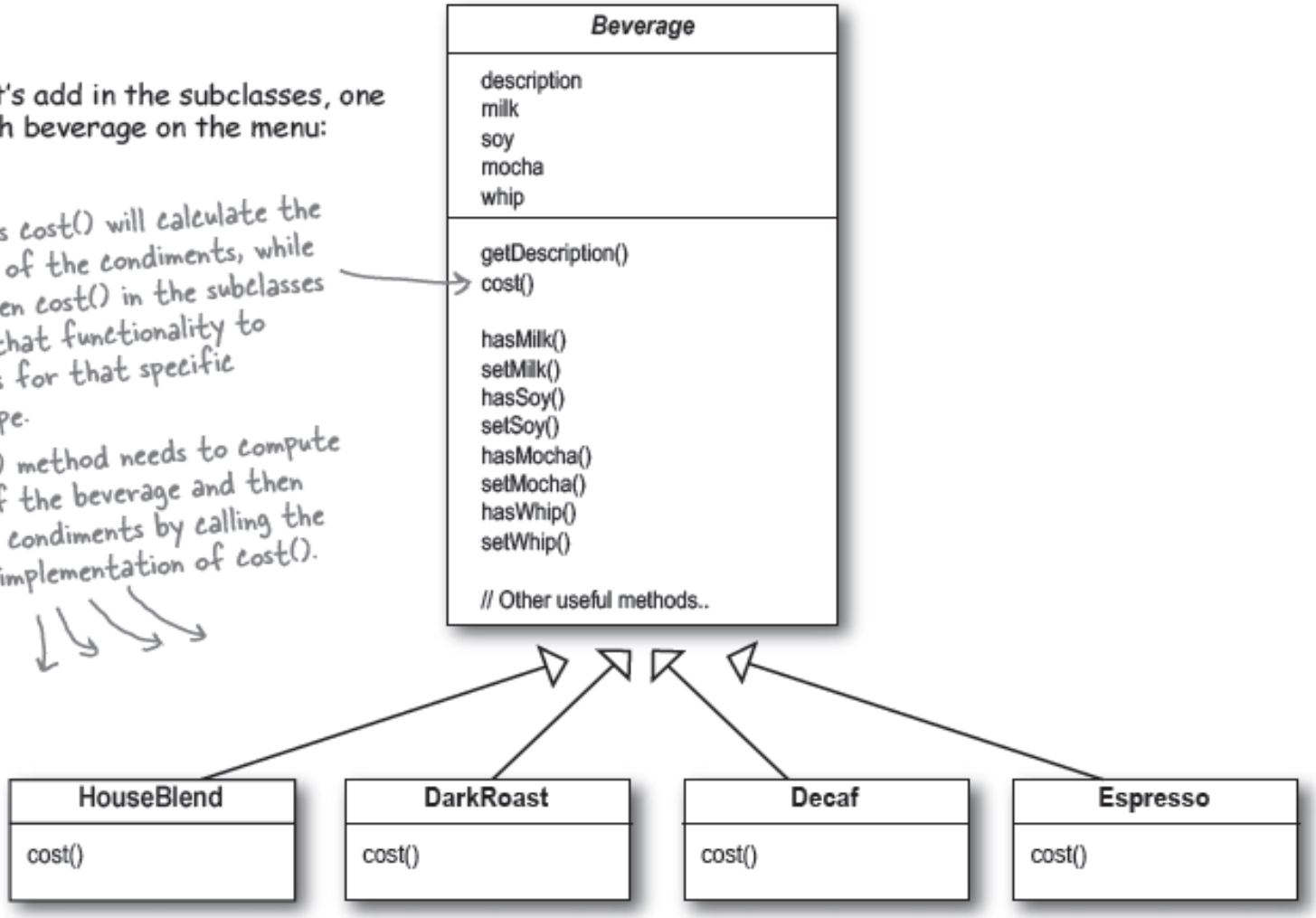
Each cost method computes the cost of the coffee along with the other condiments in the order.

Another Solution

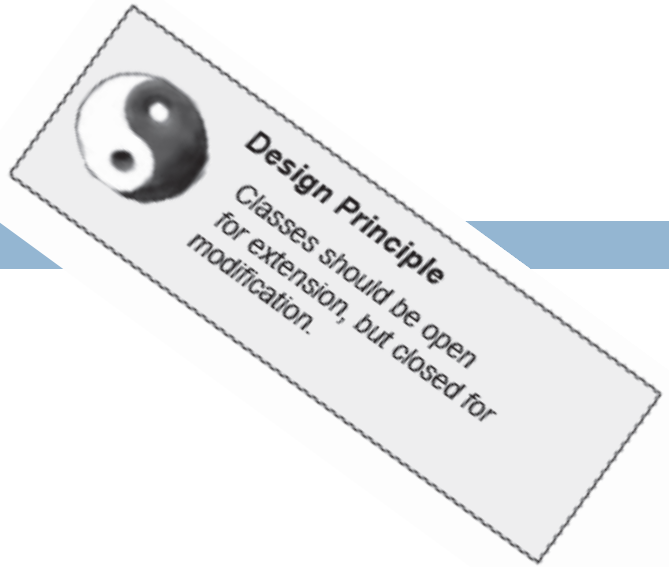
Now let's add in the subclasses, one for each beverage on the menu:

The superclass cost() will calculate the costs for all of the condiments, while the overridden cost() in the subclasses will extend that functionality to include costs for that specific beverage type.

Each cost() method needs to compute the cost of the beverage and then add in the condiments by calling the superclass implementation of cost().



Things to Remember



Our goal is to allow classes to be easily extended to incorporate new behavior without modifying existing code. What do we get if we accomplish this? Designs that are resilient to change and flexible enough to take on new functionality to meet changing requirements.

While it may seem like a contradiction, there are techniques for allowing code to be extended without direct modification.

Be careful when choosing the areas of code that need to be extended; applying the Open-Closed Principle EVERYWHERE is wasteful, unnecessary, and can lead to complex, hard to understand code.

Decorator VS. Extension Methods ?

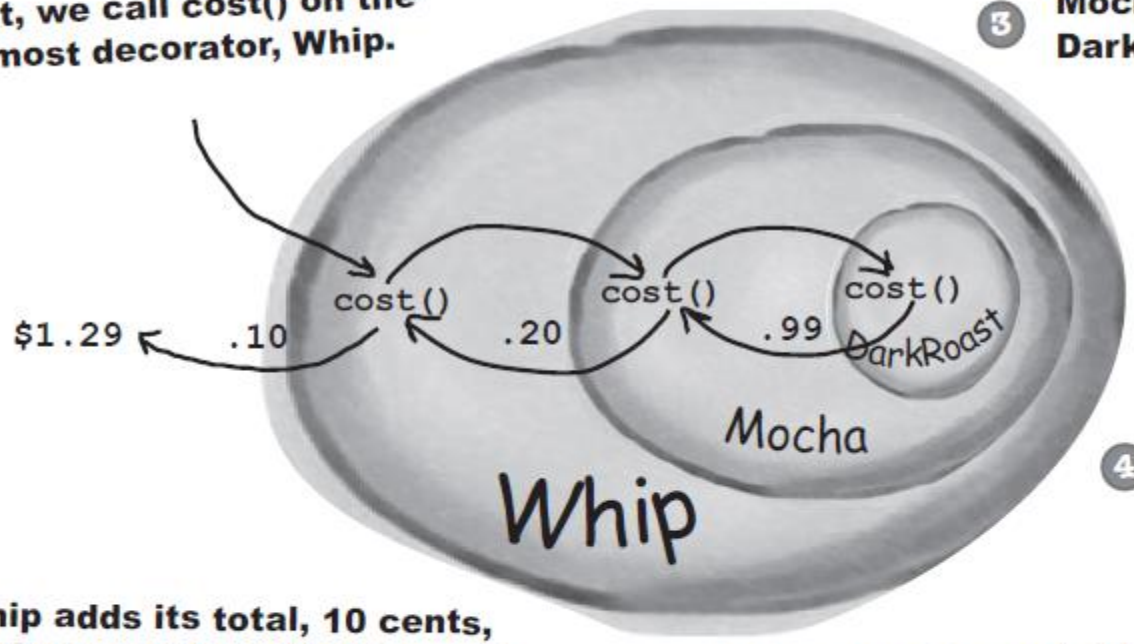
4 **Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.**

(You'll see how in a few pages.)

2 **Whip calls cost() on Mocha.**

1 **First, we call cost() on the outmost decorator, Whip.**

3 **Mocha calls cost() on DarkRoast.**



4 **DarkRoast returns its cost, 99 cents.**

5 **Whip adds its total, 10 cents, to the result from Mocha, and returns the final result—$1.29.**

5 **Mocha adds its cost, 20 cents, to the result from DarkRoast, and returns the new total, $1.19.**

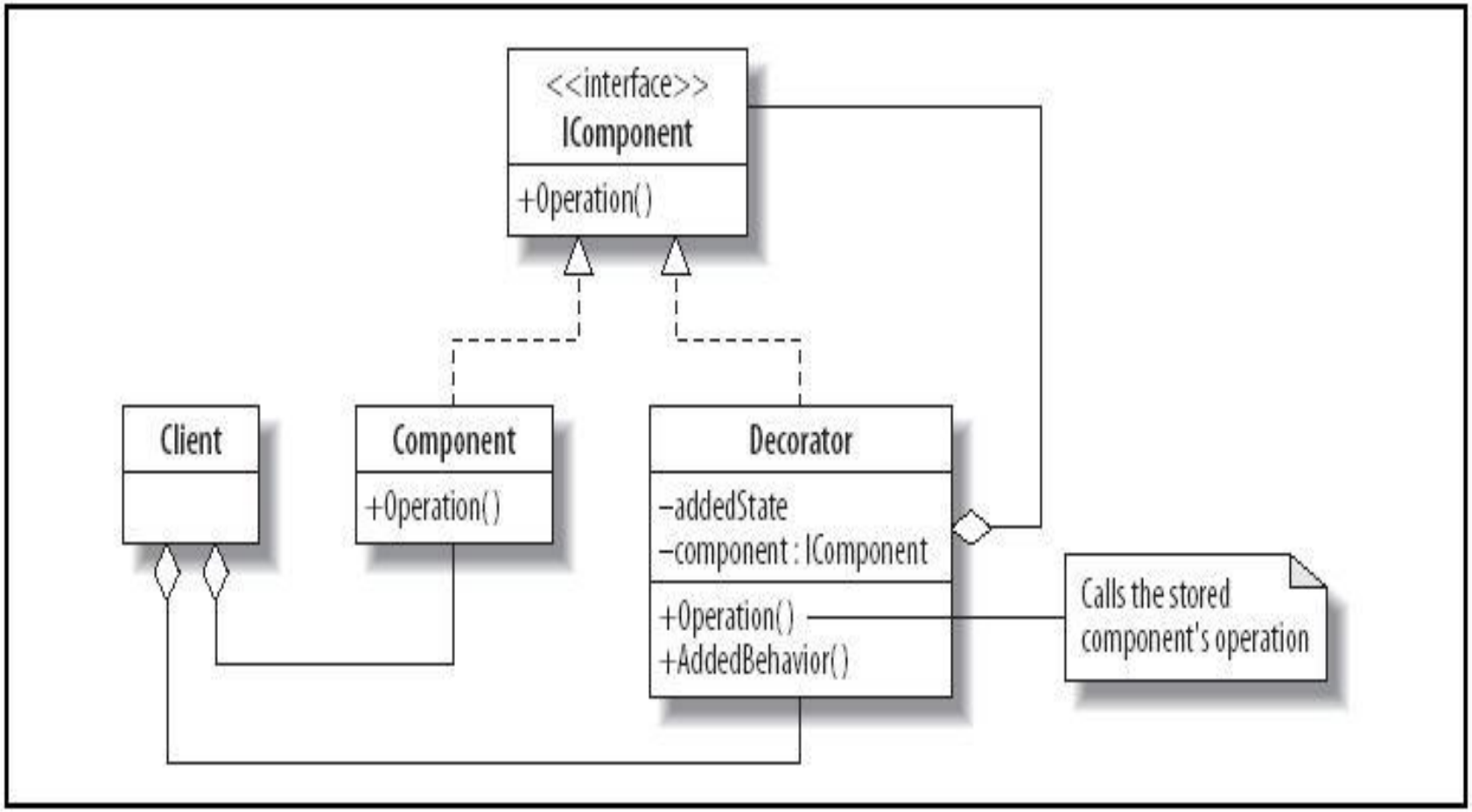
Decorator

Okay, here's what we know so far...

- Decorators have the same supertype as the objects they decorate.
- You can use one or more decorators to wrap an object.
- Given that the decorator has the same supertype as the object it decorates, we can pass around a decorated object in place of the original (wrapped) object.
- The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job.
- Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like.

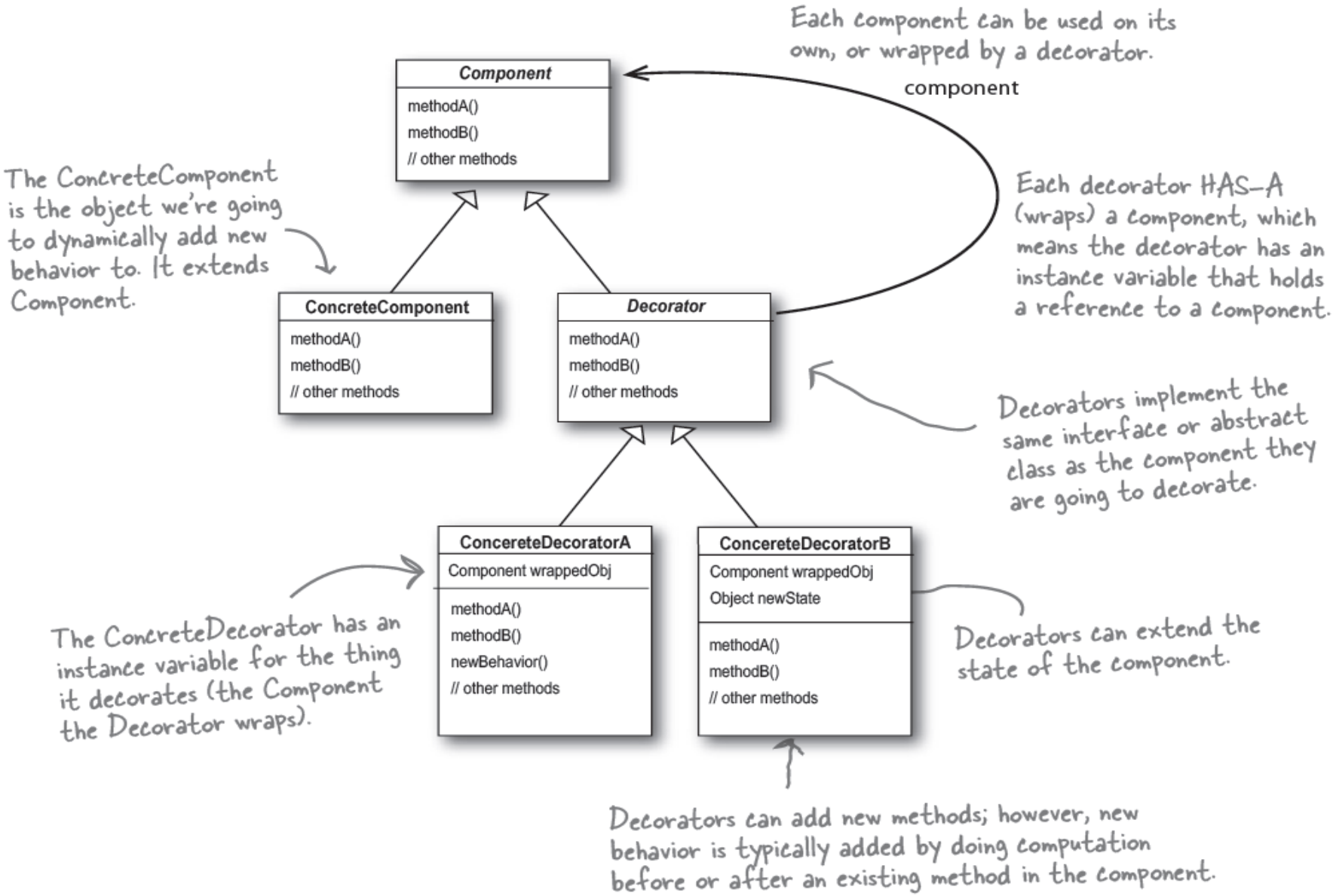
Key Point!

Decorator

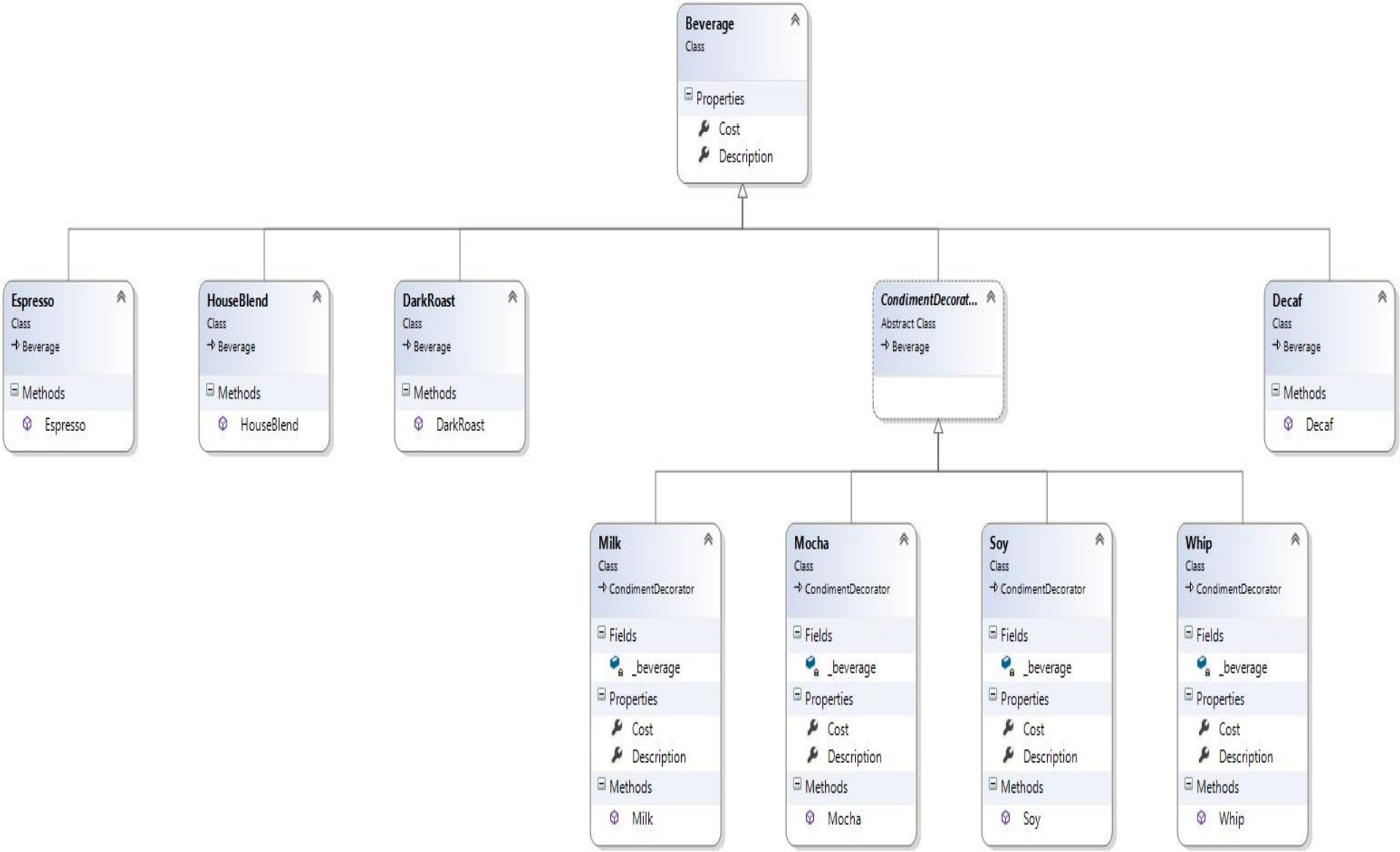

<<interface>>
IComponent
+Operation()
Client
Component
+Operation()
Decorator
–addedState
–component : IComponent
+Operation()
+AddedBehavior()
Calls the stored component's operation

Decorator

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



Decorating our Beverage



Mocha is a decorator, so we extend CondimentDecorator.

Remember, CondimentDecorator extends Beverage.

```
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();
    }
}
```

We're going to instantiate Mocha with a reference to a Beverage using:

(1) An instance variable to hold the beverage we are wrapping.

(2) A way to set this instance variable to the object we are wrapping. Here, we're going to pass the beverage we're wrapping to the decorator's constructor.

We want our description to not only include the beverage – say "Dark Roast" – but also to include each item decorating the beverage, for instance, "Dark Roast, Mocha". So we first delegate to the object we are decorating to get its description, then append ", Mocha" to that description.

Now we need to compute the cost of our beverage with Mocha. First, we delegate the call to the object we're decorating, so that it can compute the cost; then, we add the cost of Mocha to the result.

Decorated with Mocha

```
public class StarbuzzCoffee {

    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());
    }
}
```

Order up an espresso, no condiments and print its description and cost.

Make a DarkRoast object.

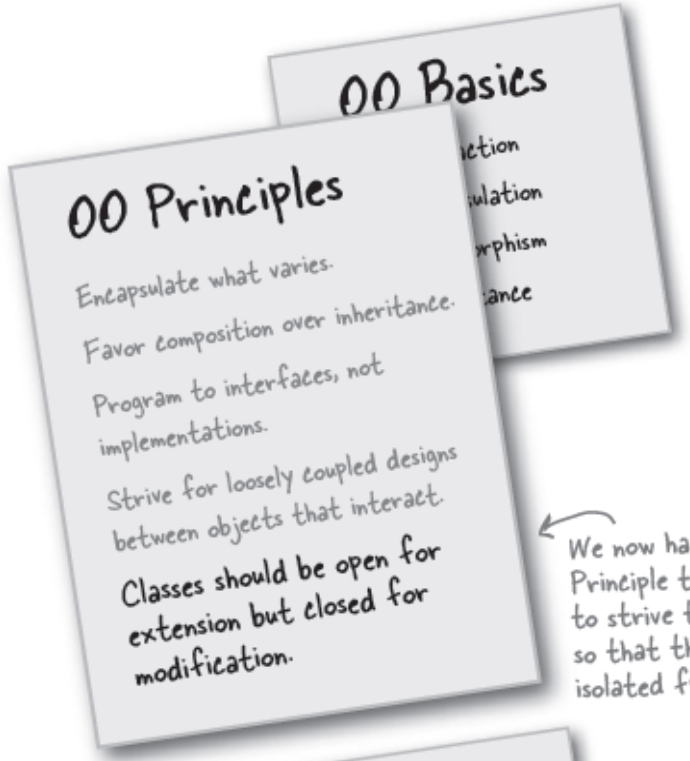
Wrap it with a Mocha.

Wrap it in a second Mocha.

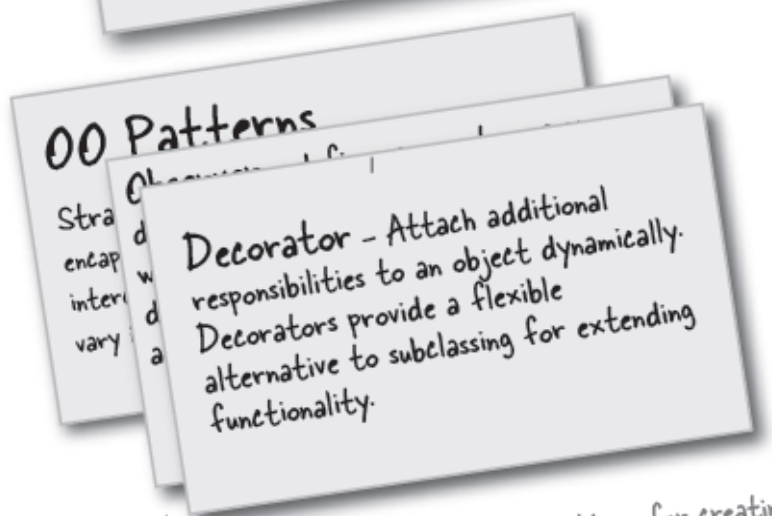
Wrap it in a Whip.

Finally, give us a HouseBlend with Soy, Mocha, and Whip.

Last Word



We now have the Open-Closed Principle to guide us. We're going to strive to design our system so that the closed parts are isolated from our new extensions.



And here's our first pattern for creating designs that satisfy the Open-Closed Principle. Or was it really the first? Is there another pattern we've used that follows this principle as well?

DAY TWO

New means Concrete

```
Duck duck = new MallardDuck();
```

We want to use interfaces to keep code flexible.

But we have to create an instance of a concrete class!

```
Duck duck;

if (picnic) {
    duck = new MallardDuck();
} else if (hunting) {
    duck = new DecoyDuck();
} else if (inBathTub) {
    duck = new RubberDuck();
}
```

We have a bunch of different duck classes, and we don't know until runtime which one we need to instantiate.

Pizza Factory

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

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

When you see "new", thi

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

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("greek") {
            pizza = new GreekPizza();
        } else if (type.equals("pepperoni") {
            pizza = new PepperoniPizza();
        }

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

We're now passing in the type of pizza to orderPizza.

Based on the type of pizza, we instantiate the correct concrete class and assign it to the pizza instance variable. Note that each pizza here has to implement the Pizza interface.

Once we have a Pizza, we prepare it (you know, roll the dough, put on the sauce and add the toppings & cheese), then we bake it, cut it and box it!

Each Pizza subtype (CheesePizza, VeggiePizza, etc.) knows how to prepare itself.

What Varies and What Stay

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

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("greek") {
            pizza = new GreekPizza();
        } else if (type.equals("pepperoni") {
            pizza = new PepperoniPizza();
        } else if (type.equals("clam") {
            pizza = new ClamPizza();
        } else if (type.equals("veggie") {
            pizza = new eggiePizza();
        }

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

This code is NOT closed for modification. If the Pizza Shop changes its pizza offerings, we have to get into this code and modify it.

This is what varies. As the pizza selection changes over time, you'll have to modify this code over and over.

This is what we expect to stay the same. For the most part, preparing, cooking, and packaging a pizza has remained the same for years and years. So, we don't expect this code to change, just the pizzas it operates on.

Encapsulating Object Creation

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

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

First we pull the object creation code out of the orderPizza Method

What's going to go here?

Then we place that code in an object that is only going to worry about how to create pizzas. If any other object needs a pizza created, this is the object to come to.

Creating The Factory

Now we give PizzaStore a reference to a SimplePizzaFactory.

```
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;
    }

    // other methods here
}
```

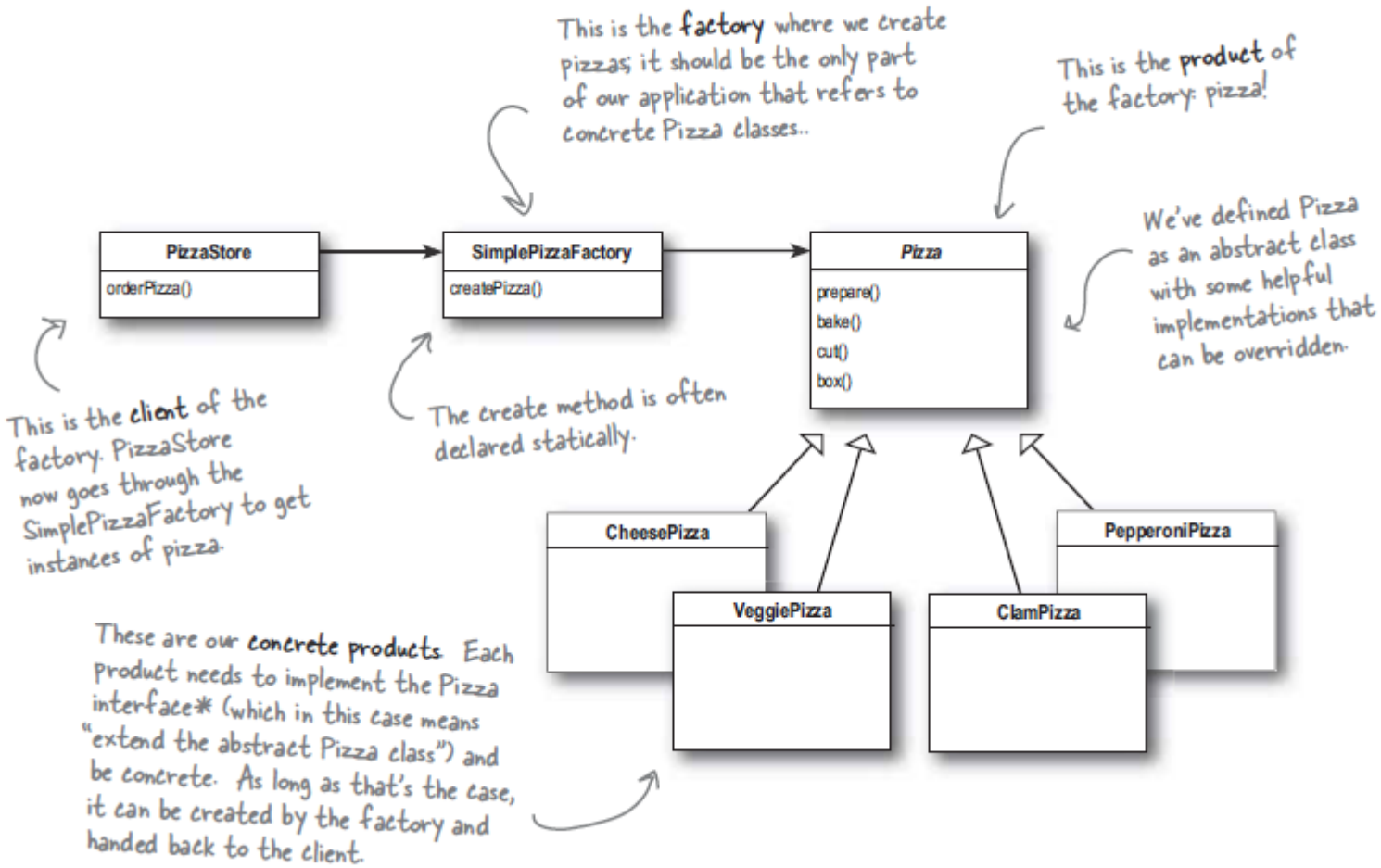
PizzaStore gets the factory passed to it in the constructor.

And the orderPizza() method uses the factory to create its pizzas by simply passing on the type of the order.

Notice that we've replaced the **new operator** with a create **method** on the factory object. No more concrete instantiations here!

The Simple Factory

- It is Not a REAL Design Pattern (YET)



Example

```
interface IProduct {
  string ShipFrom( );
}

class ProductA : IProduct {
  public String ShipFrom ( ) {
    return " from South Africa";
  }
}

class ProductB : IProduct {
  public String ShipFrom ( )
    return "from Spain";
  }
}
```

```
class Creator {
  public IProduct FactoryMethod(int month) {
    if (month >= 4 && month <=11)
        return new ProductA( );
    else
    if (month == 1 || month == 2 || month == 12)
        return new ProductB( );
    else return new DefaultProduct( );
  }
static void Main( ) {
  Creator c = new Creator( );
  IProduct product;

  for (int i=1; i<=12; i++) {
    product = c.FactoryMethod(i);
    Console.WriteLine("Avocados "+product.ShipFrom( ));
  }
}
```

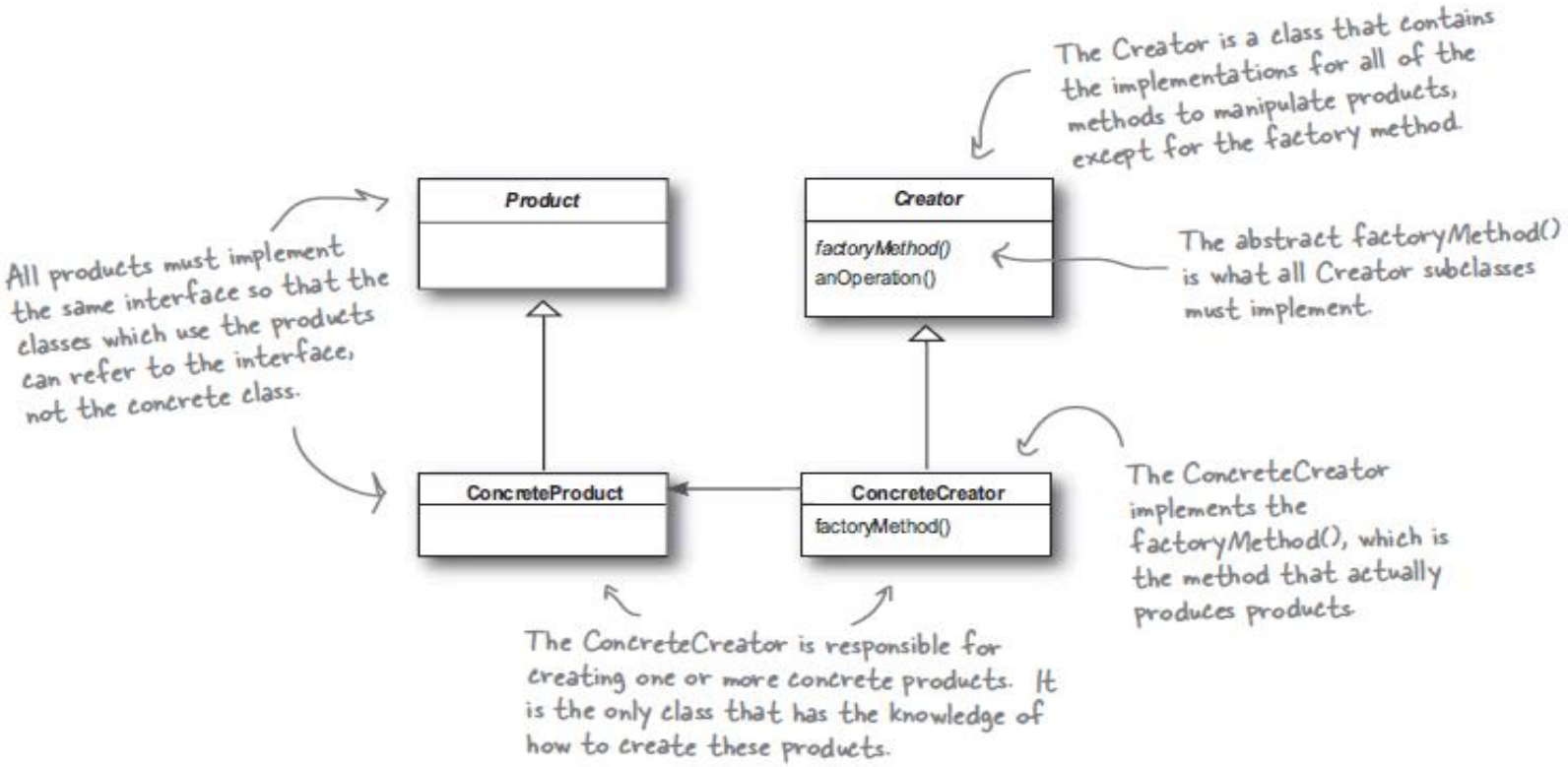
Still Not A Pattern

Introducing the Factory Method Pattern

Factory Method Pattern

Product P = Creator.FactoryMethod( )

The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.



Franchising the pizza store

```
public abstract class PizzaStore {

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

    protected abstract Pizza createPizza(String type);

    // other methods here
}
```

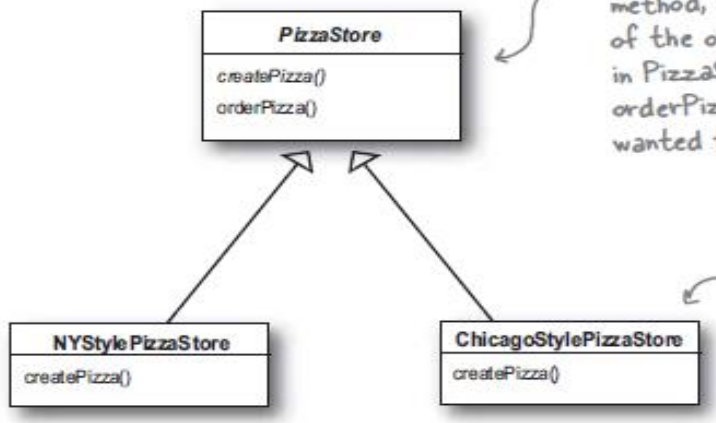
The subclasses of PizzaStore handle object instantiation for us in the createPizza() method.

NYStylePizzaStore
createPizza()

ChicagoStylePizzaStore
createPizza()

All the responsibility for instantiating Pizzas has been moved into a **method** that acts as a **factory**.

Allowing the subclasses to decide



Each subclass overrides the createPizza() method, while all subclasses make use of the orderPizza() method defined in PizzaStore. We could make the orderPizza() method final if we really wanted to enforce this.

If a franchise wants NY style pizzas for its customers, it uses the NY subclass, which has its own createPizza() method, creating NY style pizzas.

Remember: createPizza() is abstract in PizzaStore, so all pizza store subtypes MUST implement the method.

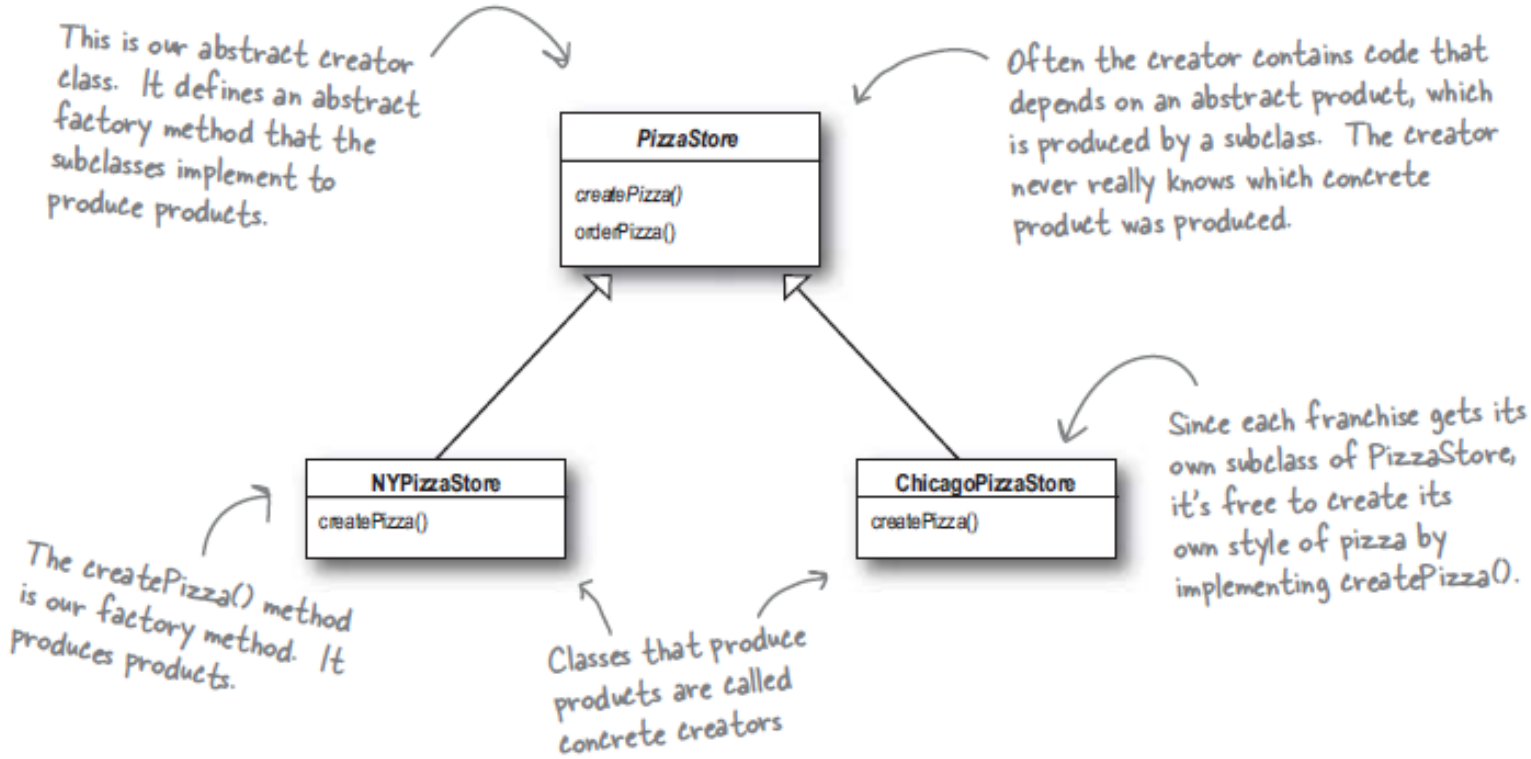
Similarly, by using the Chicago subclass, we get an implementation of createPizza() with Chicago ingredients.

```
public Pizza createPizza(type) {
    if (type.equals("cheese")) {
        pizza = new NYStyleCheesePizza();
    } else if (type.equals("pepperoni") {
        pizza = new NYStylePepperoniPizza();
    } else if (type.equals("clam") {
        pizza = new NYStyleClamPizza();
    } else if (type.equals("veggie") {
        pizza = new NYStyleVeggiePizza();
    }
```

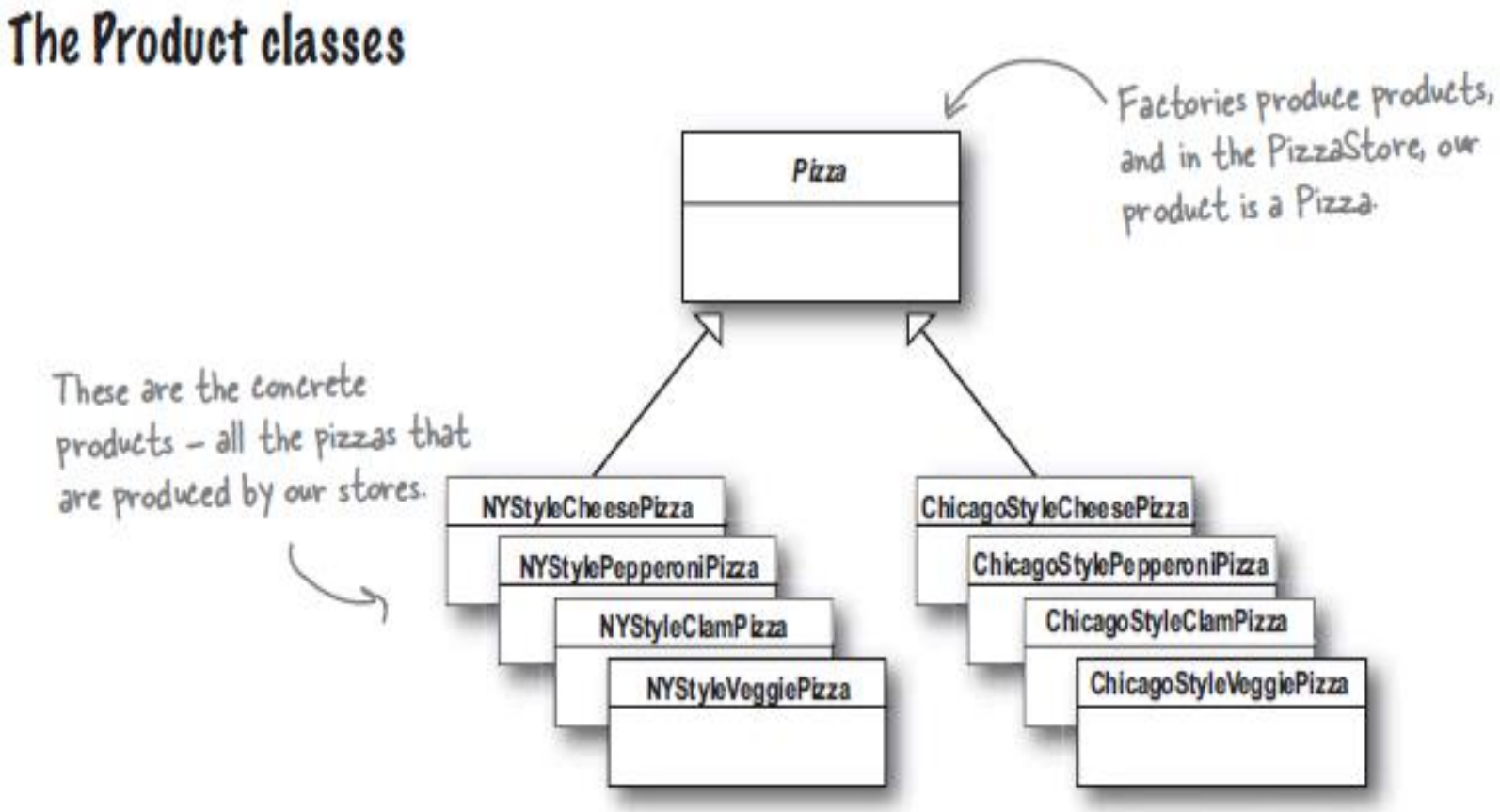
```
public Pizza createPizza(type) {
    if (type.equals("cheese")) {
        pizza = new ChicagoStyleCheesePizza();
    } else if (type.equals("pepperoni") {
        pizza = new ChicagoStylePepperoniPizza();
    } else if (type.equals("clam") {
        pizza = new ChicagoStyleClamPizza();
    } else if (type.equals("veggie") {
        pizza = new ChicagoStyleVeggiePizza();
    }
}
```

Creator

The Creator classes



Product


The Product classes
Factories produce products, and in the PizzaStore, our product is a Pizza.
Pizza
These are the concrete products – all the pizzas that are produced by our stores.
NYStyleCheesePizza
NYStylePepperoniPizza
NYStyleClamPizza
NYStyleVeggiePizza
ChicagoStyleCheesePizza
ChicagoStylePepperoniPizza
ChicagoStyleClamPizza
ChicagoStyleVeggiePizza

The Client

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.order("Veggie");
```

Here we create a factory for making NY style pizzas.

Then we create a PizzaStore and pass it a reference to the NY factory.

...and when we make pizzas, we get NY-styled pizzas.

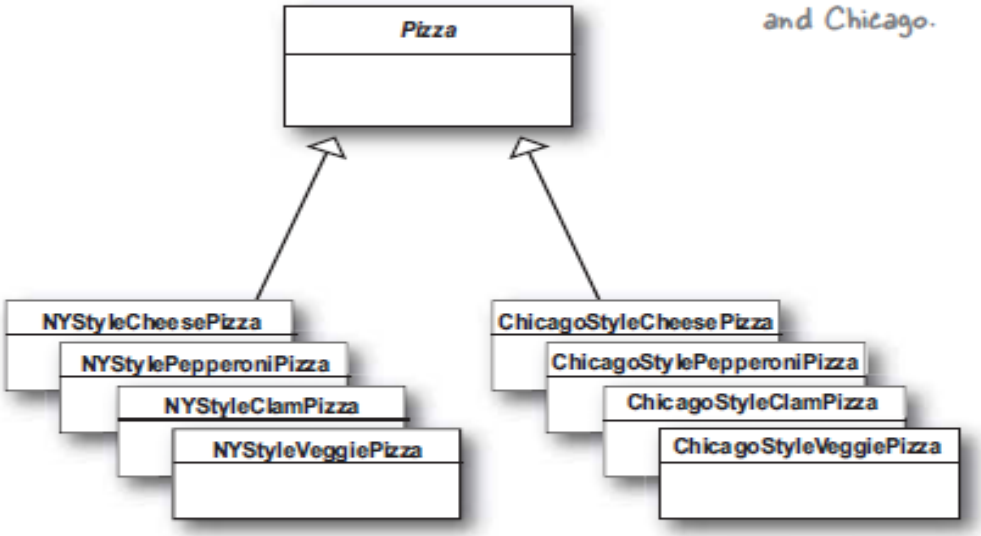
```
ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();
PizzaStore chicagoStore = new PizzaStore(chicagoFactory);
chicagoStore.order("Veggie");
```

Likewise for the Chicago pizza stores: we create a factory for Chicago pizzas and create a store that is composed with a Chicago factory. When we make pizzas, we get the Chicago flavored ones

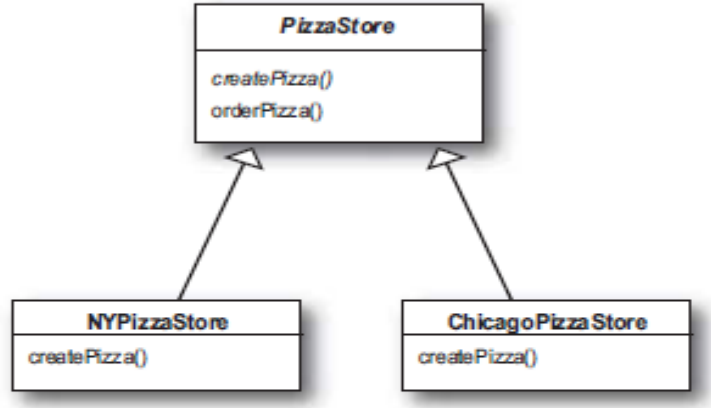
Another perspective: parallel class hierarchies

Notice how these class hierarchies are parallel: both have abstract classes that are extended by concrete classes, which know about specific implementations for NY and Chicago.

The Product classes



The Creator classes

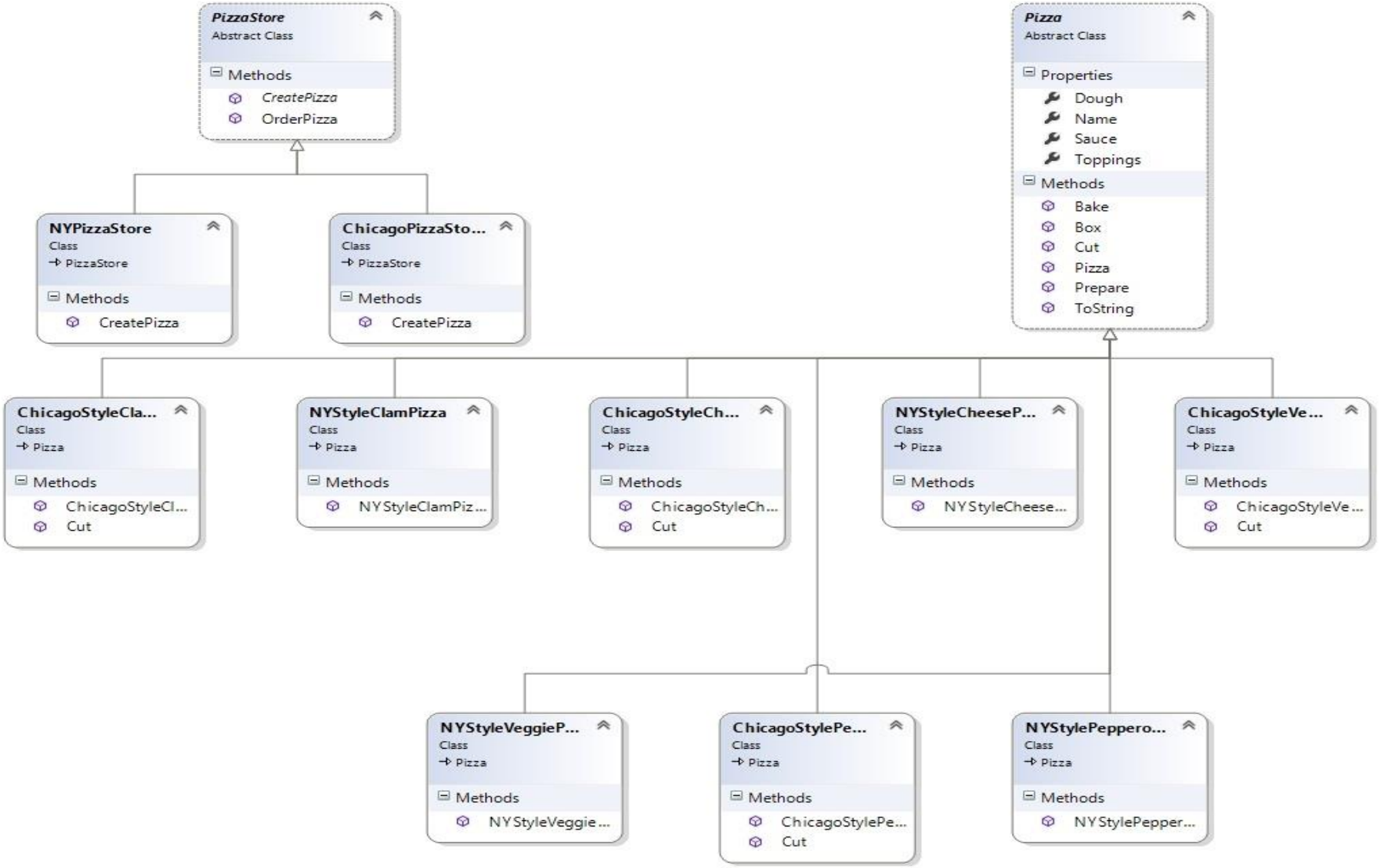


The NYPizzaStore encapsulates all the knowledge about how to make NY style pizzas.

The ChicagoPizzaStore encapsulates all the knowledge about how to make Chicago style pizzas.

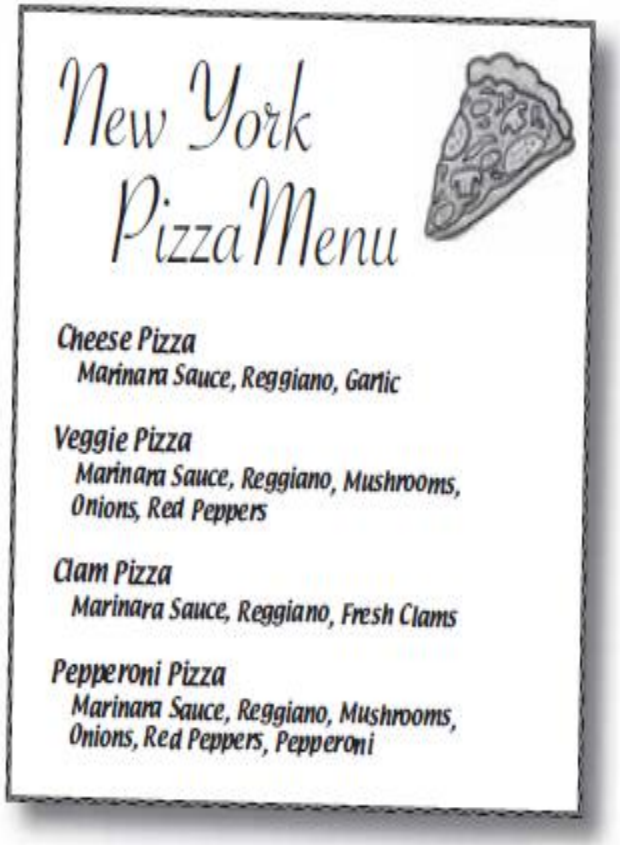
The factory method is the key to encapsulating this knowledge.

UML Class Diagram



Back to Pizza

We've got the same product families (dough, sauce, cheese, veggies, meats) but different implementations based on region.

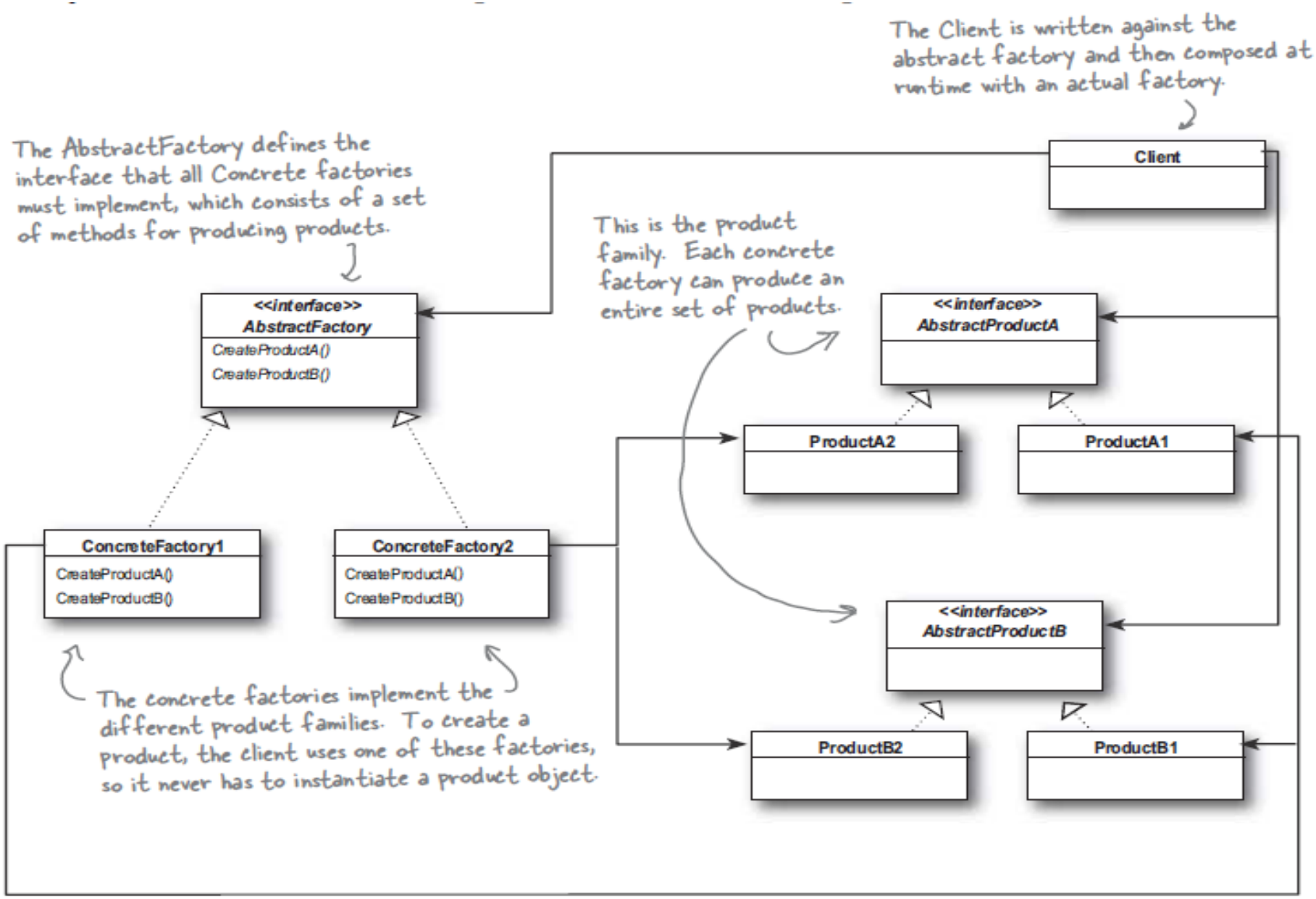


Abstract Factory

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

- This pattern supports the creation of products that exist in families and are designed to be produced together.
- The abstract factory can be refined to concrete factories, each of which can create different products of different types and in different combinations.
- The pattern isolates the product definitions and their class names from the client so that the only way to get one of them is through a factory.
- For this reason ,product families can easily be interchanged or updated without upsetting the structure of the client.

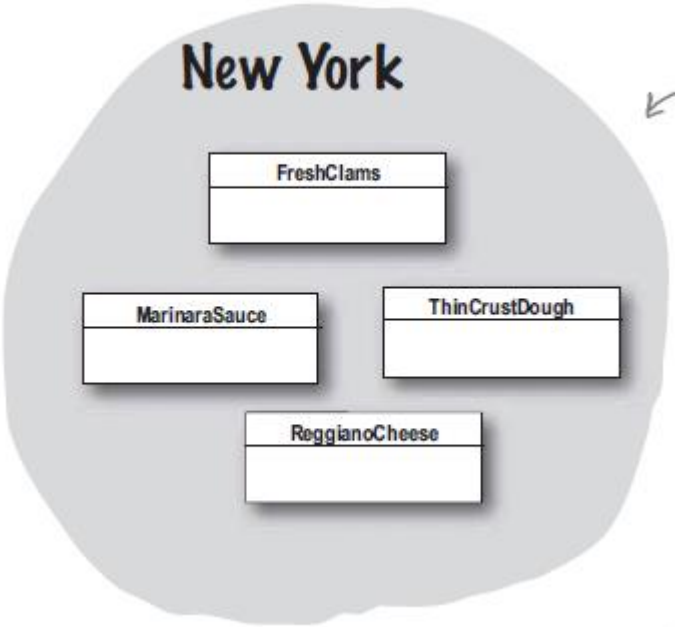
UML

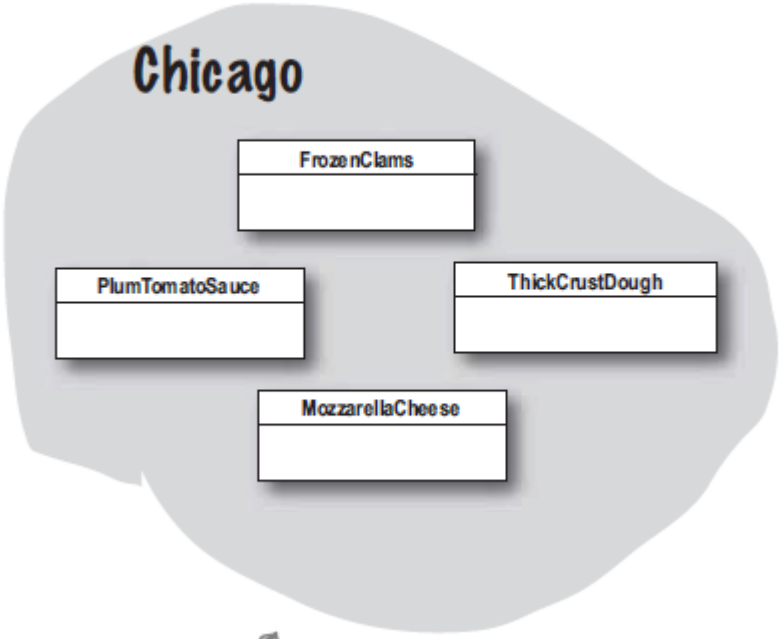

The Client is written against the abstract factory and then composed at runtime with an actual factory.
Client
The AbstractFactory defines the interface that all Concrete factories must implement, which consists of a set of methods for producing products.
<<interface>>
AbstractFactory
CreateProductA()
CreateProductB()
This is the product family. Each concrete factory can produce an entire set of products.
<<interface>>
AbstractProductA
ProductA2
ProductA1
ConcreteFactory1
CreateProductA()
CreateProductB()
ConcreteFactory2
CreateProductA()
CreateProductB()
<<interface>>
AbstractProductB
The concrete factories implement the different product families. To create a product, the client uses one of these factories, so it never has to instantiate a product object.
ProductB2
ProductB1

Meanwhile, back at the PizzaStore...

Families of ingredients...

New York uses one set of ingredients and Chicago another. Given the popularity of Objectville Pizza it won't be long before you also need to ship another set of regional ingredients to California,





Each family consists of a type of dough, a type of sauce, a type of cheese, and a seafood topping (along with a few more we haven't shown, like veggies and spices).

Building the ingredient factories

```
public interface PizzaIngredientFactory {

        Dough createDough();
        Sauce createSauce();
        Cheese createCheese();
        Veggies[] createVeggies();
        Pepperoni createPepperoni();
        Clams createClam();

}
```

For each ingredient we define a create method in our interface.

Lots of new classes here, one per ingredient.

If we'd had some common "machinery" to implement in each instance of factory, we could have made this an abstract class instead...

Building the New York ingredient factory

```
public class NYPizzaIngredientFactory    :       PizzaIngredientFactory {

    public Dough createDough() {
        return new ThinCrustDough();
    }

    public Sauce createSauce() {
        return new MarinaraSauce();
    }

    public Cheese createCheese() {
        return new ReggianoCheese();
    }

    public Veggies[] createVeggies() {
        Veggies veggies[] = { new Garlic(), new Onion(), new Mushroom(), new RedPepper() };
        return veggies;
    }

    public Pepperoni createPepperoni() {
        return new SlicedPepperoni();
    }
    public Clams createClam() {
        return new FreshClams();
    }
}
```

For each ingredient in the ingredient family, we create the New York version.

For veggies, we return an array of Veggies. Here we've hardcoded the veggies. We could make this more sophisticated, but that doesn't really add anything to learning the factory pattern, so we'll keep it simple.

New York is on the coast; it gets fresh clams. Chicago has to settle for frozen.

The best sliced pepperoni. This is shared between New York and Chicago. Make sure you use it on the next page when you get to implement the Chicago factory yourself

ChicagoPizzaIngredientFactory.

```
public class ChicagoPizzaIngredientFactory : IPizzaIngredientFactory {

   public IDough CreateDough()      { return new ThickCrustDough();}

   public ISauce CreateSauce()      { return new PlumTomatoSauce();}

   public ICheese CreateCheese()    { return new MozzarellaCheese(); }

   public IVeggies[] CreateVeggies()    {
         return new IVeggies[] = { new BlackOlives(), new Spinach(),new
                  Eggplant() }; }

   public IPepperoni CreatePepperoni()  { return new SlicedPepperoni(); }

   public IClams CreateClam()       { return new FrozenClams(); }
 }
```

pIZZA

```
public abstract class Pizza {
    String name;
    Dough dough;
    Sauce sauce;
    Veggies veggies[];
    Cheese cheese;
    Pepperoni pepperoni;
    Clams clam;

    abstract void prepare();

    void bake() {
                        ("Bake for 25 minutes at 350");


    void cut() {
                        ("Cutting the pizza into diagonal slices");


    void box() {
                        ("Place pizza in official PizzaStore box");


    void setName(String name) {
        this.name = name;
    }

    String getName() {
        return name;
    }



    }
}
```

Each pizza holds a set of ingredients that are used in its preparation.

We've now made the prepare method abstract. This is where we are going to collect the ingredients needed for the pizza, which of course will come from the ingredient factory.

Our other methods remain the same, with the exception of the prepare method.

Pizza

```
public class CheesePizza   :     Pizza {
    PizzaIngredientFactory ingredientFactory;

    public CheesePizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }

    void prepare() {

        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
    }
}
```

To make a pizza now, we need a factory to provide the ingredients. So each Pizza class gets a factory passed into its constructor, and it's stored in an instance variable.

Here's where the magic happens!

```
sauce = ingredientFactory.createSauce();
```

We're setting the Pizza instance variable to refer to the specific sauce used in this pizza.

This is our ingredient factory. The Pizza doesn't care which factory is used, as long as it is an ingredient factory.

The createSauce() method returns the sauce that is used in its region. If this is a NY ingredient factory, then we get marinara sauce.

NYPizzaStore Again

```
public class NYPizzaStore extends PizzaStore {

    protected Pizza createPizza(String item) {
        Pizza pizza = null;
        PizzaIngredientFactory ingredientFactory =
            new NYPizzaIngredientFactory();

        if (item.equals("cheese")) {

            pizza = new CheesePizza(ingredientFactory);
            pizza.setName("New York Style Cheese Pizza");

        } else if (item.equals("veggie")) {

            pizza = new VeggiePizza(ingredientFactory);
            pizza.setName("New York Style Veggie Pizza");

        } else if (item.equals("clam")) {

            pizza = new ClamPizza(ingredientFactory);
            pizza.setName("New York Style Clam Pizza");

        } else if (item.equals("pepperoni")) {
            pizza = new PepperoniPizza(ingredientFactory);
            pizza.setName("New York Style Pepperoni Pizza");

        }
        return pizza;
    }
}
```

The NY Store is composed with a NY pizza ingredient factory. This will be used to produce the ingredients for all NY style pizzas.

We now pass each pizza the factory that should be used to produce its ingredients.

Look back one page and make sure you understand how the pizza and the factory work together!

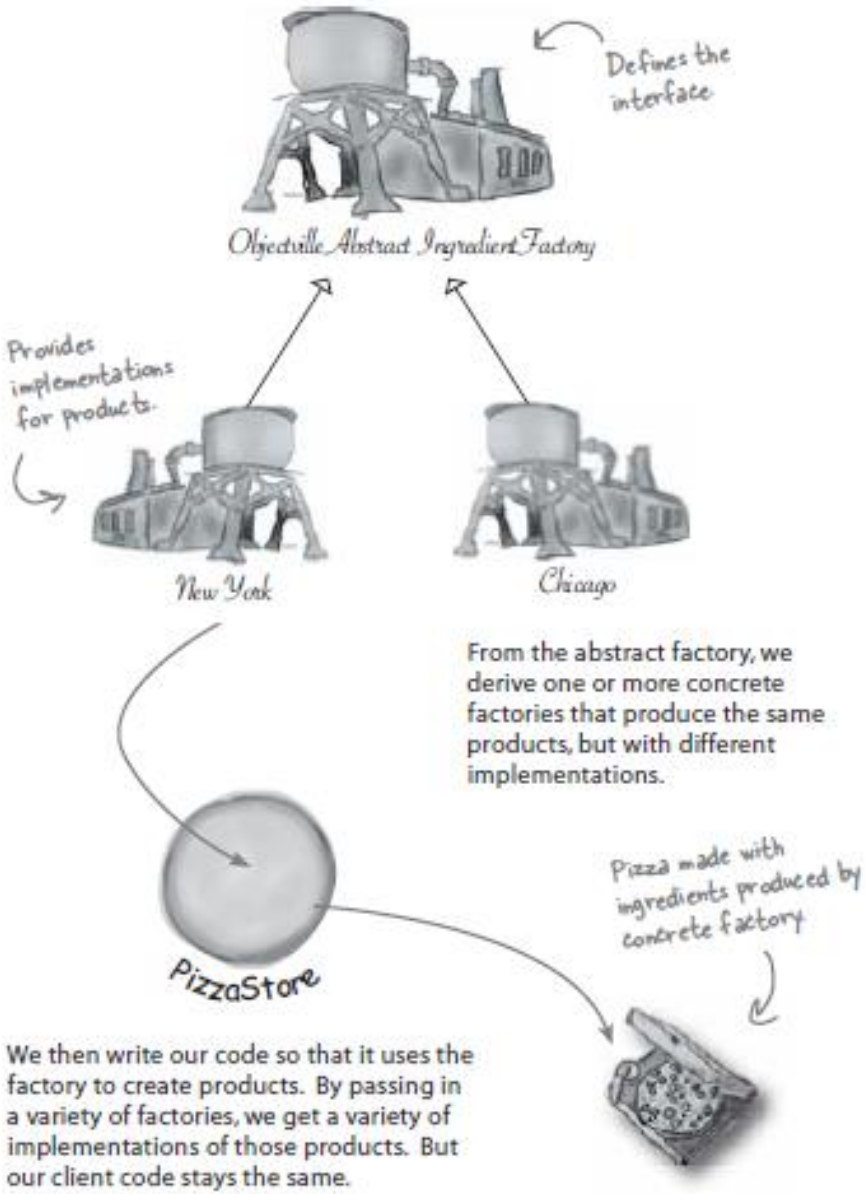
For each type of Pizza, we instantiate a new Pizza and give it the factory it needs to get its ingredients.

What Really Happens

We provided a means of creating a family of ingredients for pizzas by introducing a new type of factory called an Abstract Factory.

An Abstract Factory gives us an interface for creating a family of products. By writing code that uses this interface, we decouple our code from the actual factory that creates the products. That allows us to implement a variety of factories that produce products meant for different contexts – such as different regions, different operating systems, or different look and feels.

An Abstract Factory provides an interface for a family of products. What's a family? In our case it's all the things we need to make a pizza: dough, sauce, cheese, meats and veggies.



From the abstract factory, we derive one or more concrete factories that produce the same products, but with different implementations.

We then write our code so that it uses the factory to create products. By passing in a variety of factories, we get a variety of implementations of those products. But our client code stays the same.

The Pizza Story

5 **Next we need to prepare the pizza. Once the prepare() method is called, the factory is asked to prepare ingredients:**

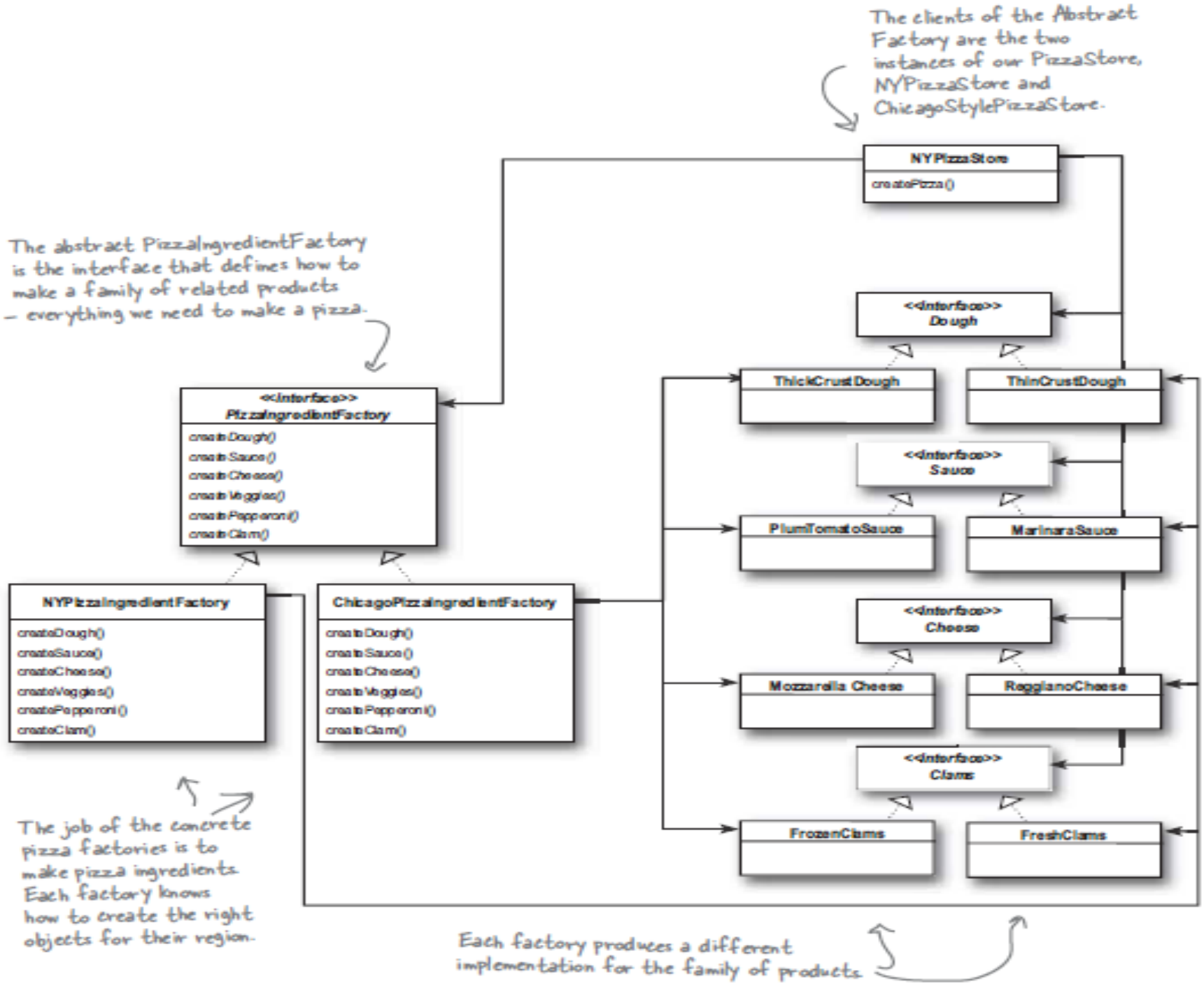
```
void prepare() {
    dough = factory.createDough();
    sauce = factory.createSauce();
    cheese = factory.createCheese();
}
```

Thin crust

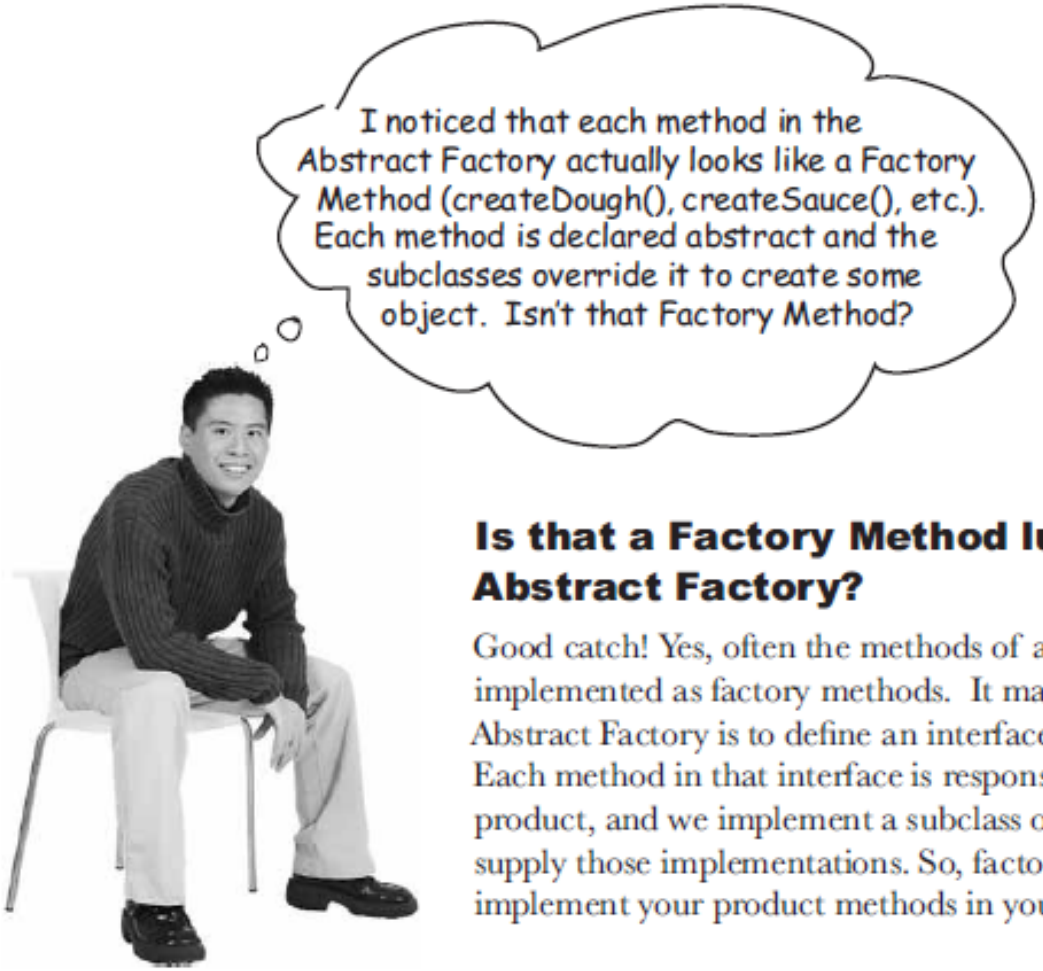
Marinara

Reggiano

For Ethan's pizza the New York ingredient factory is used, and so we get the NY ingredients.


The clients of the Abstract Factory are the two instances of our PizzaStore, NYPizzaStore and ChicagoStylePizzaStore.
NYPizzaStore
createPizza()
The abstract PizzaIngredientFactory is the interface that defines how to make a family of related products – everything we need to make a pizza.
<<Interface>>
PizzaIngredientFactory
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
<<Interface>>
Dough
ThickCrustDough
ThinCrustDough
<<Interface>>
Sauce
PlumTomatoSauce
MarinaraSauce
<<Interface>>
Cheese
Mozzarella Cheese
ReggianoCheese
<<Interface>>
Clams
FrozenClams
FreshClams
NYPizzaIngredientFactory
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
ChicagoPizzaIngredientFactory
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
The job of the concrete pizza factories is to make pizza ingredients. Each factory knows how to create the right objects for their region.
Each factory produces a different implementation for the family of products.

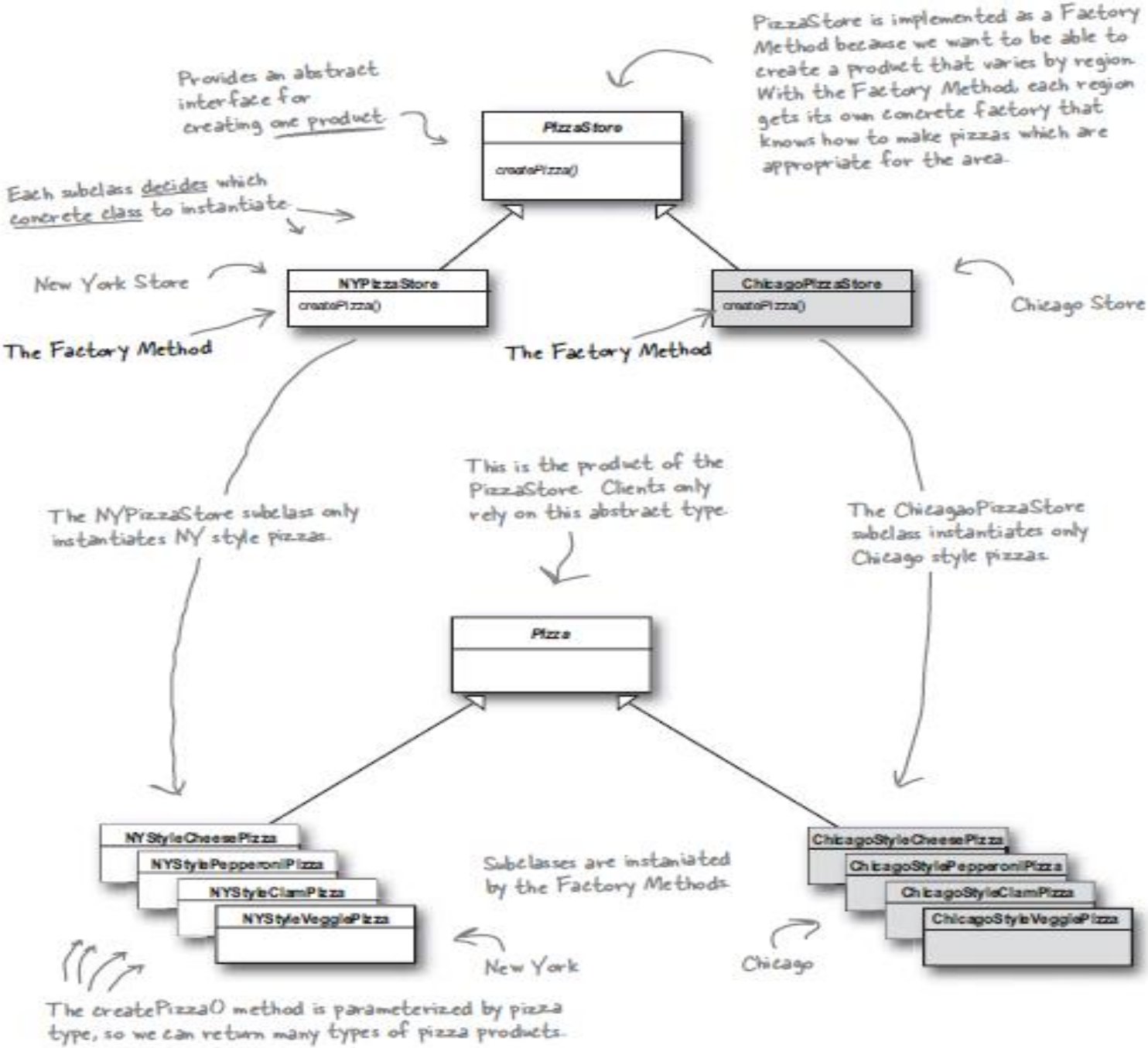
Abstract Factory VS. Factory Method

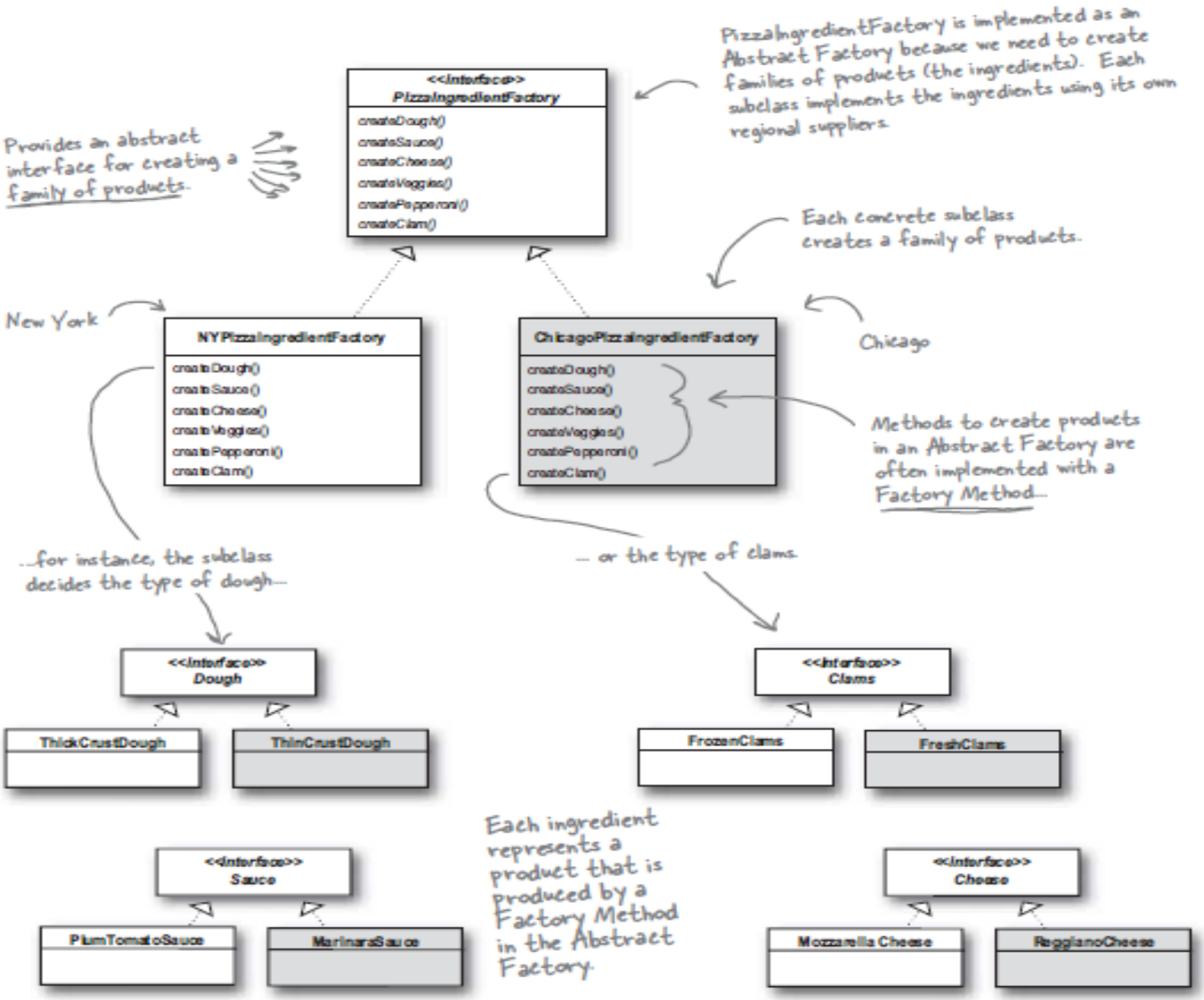


Is that a Factory Method lurking inside the Abstract Factory?

Good catch! Yes, often the methods of an Abstract Factory are implemented as factory methods. It makes sense, right? The job of an Abstract Factory is to define an interface for creating a set of products. Each method in that interface is responsible for creating a concrete product, and we implement a subclass of the Abstract Factory to supply those implementations. So, factory methods are a natural way to implement your product methods in your abstract factories.

Factory Method and Abstract Factory compared




PizzaIngredientFactory is implemented as an Abstract Factory because we need to create families of products (the ingredients). Each subclass implements the ingredients using its own regional suppliers.
<<Interface>>
PizzaIngredientFactory
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
Provides an abstract interface for creating a family of products.
Each concrete subclass creates a family of products.
New York
NYPizzaIngredientFactory
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
ChicagoPizzaIngredientFactory
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
Chicago
Methods to create products in an Abstract Factory are often implemented with a Factory Method...
...for instance, the subclass decides the type of dough...
... or the type of clams.
<<Interface>>
Dough
ThickCrustDough
ThinCrustDough
<<Interface>>
Clams
FrozenClams
FreshClams
Each ingredient represents a product that is produced by a Factory Method in the Abstract Factory.
<<Interface>>
Sauce
PlumTomatoSauce
MarinaraSauce
<<Interface>>
Cheese
MozzarellaCheese
ReggianoCheese

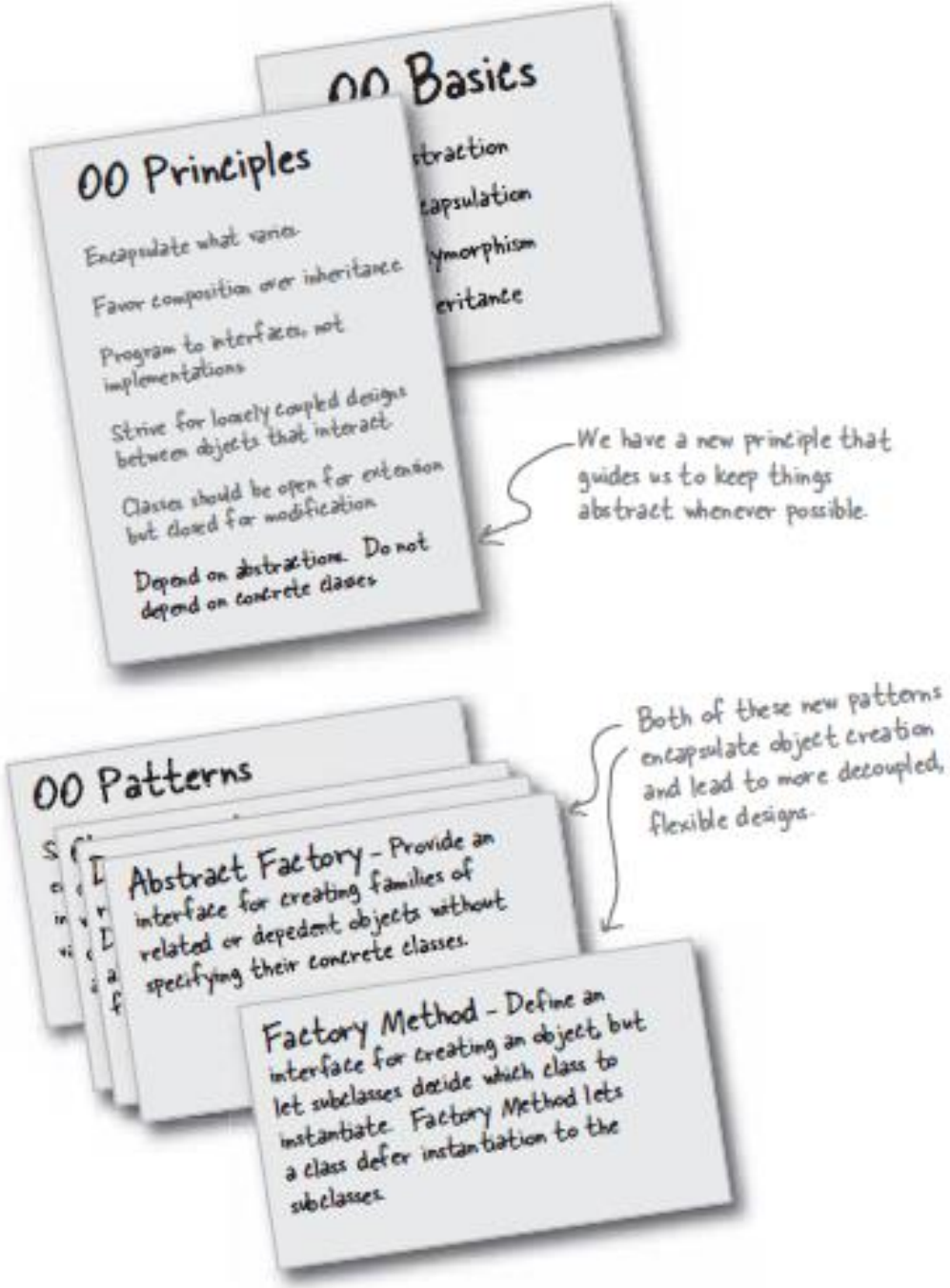
Factory Method vs Abstract Factory

- **Factory Method** is used to create one product only but **Abstract Factory** is about creating families of related or dependent products.
- **Factory Method** pattern exposes a method to the client for creating the object whereas in case of **Abstract Factory** they expose a family of related objects which may consist of these Factory methods.
- **Factory Method** pattern hides the construction of single object where as **Abstract factory method** hides the construction of a family of related objects. Abstract factories are usually implemented using (a set of) factory methods.
- **Abstract Factory** pattern uses composition to delegate responsibility of creating object to another class while **Factory** design pattern uses inheritance and relies on derived class or sub class to create object.
- The idea behind the **Factory Method** pattern is that it allows for the case where a client doesn't know what concrete classes it will be required to create at runtime, but just wants to get a class that will do the job while **Abstract Factory** pattern is best utilized when your system has to create multiple families of products or you want to provide a library of products without exposing the implementation details.!

Factory Method: Sure. Both Abstract Factory and I create objects – that's our jobs. But I do it through inheritance...

Abstract Factory: ...and I do it through object composition.

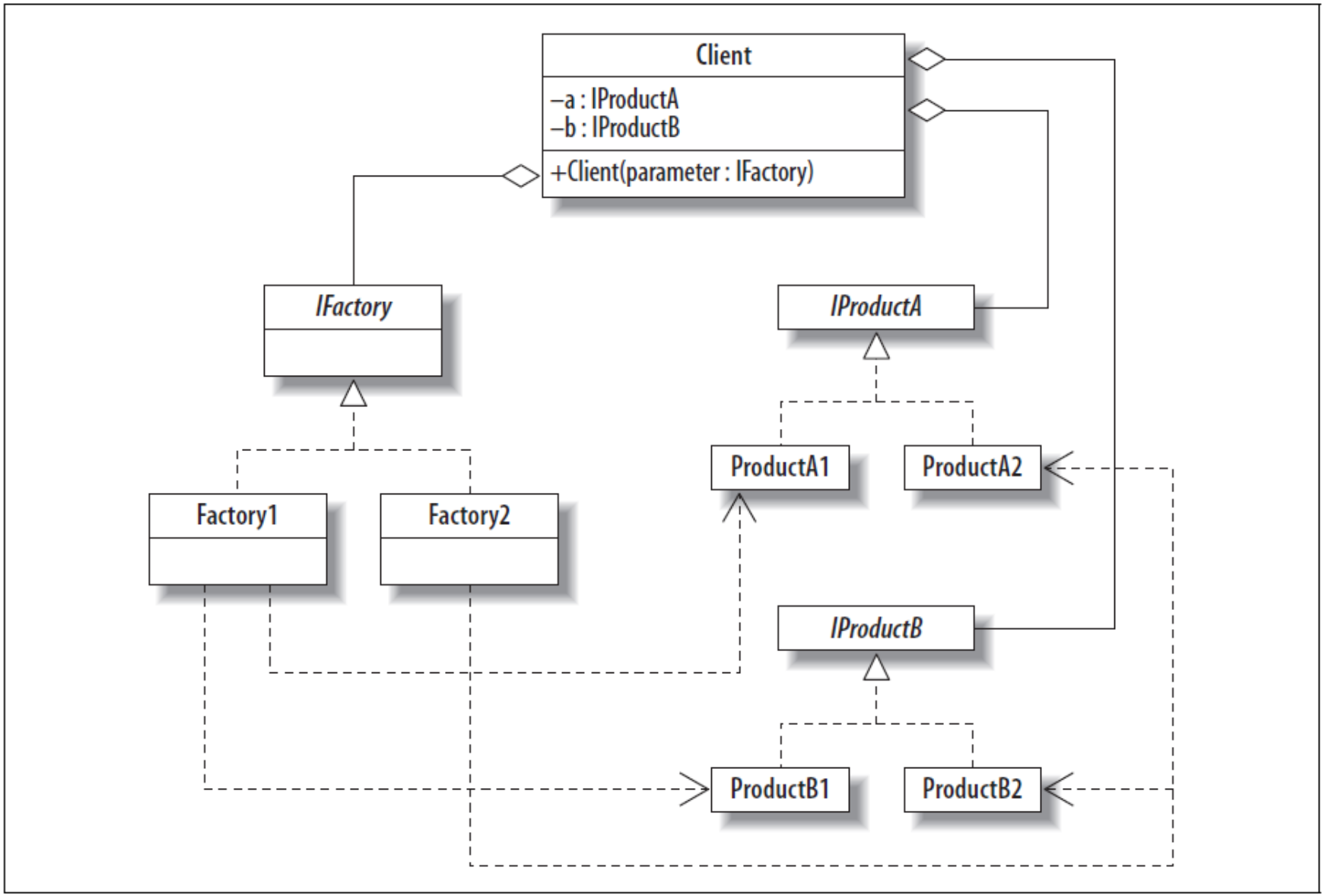
Last Word


OO Basics
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.
We have a new principle that guides us to keep things abstract whenever possible.
OO Patterns
Abstract Factory - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
Factory Method - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to the subclasses.
Both of these new patterns encapsulate object creation and lead to more decoupled, flexible designs.

Points to Remember

- All factories encapsulate object creation.
- Simple Factory, while not a bona fide design pattern, is a simple way to decouple your clients from concrete classes.
- Factory Method relies on inheritance: object creation is delegated to subclasses which implement the factory method to create objects.
- Abstract Factory relies on object composition: object creation is implemented in methods exposed in the factory interface.
- All factory patterns promote loose coupling by reducing the dependency of your application on concrete classes.
- The intent of Factory Method is to allow a class to defer instantiation to its subclasses.
- The intent of Abstract Factory is to create families of related objects without having to depend on their concrete classes.
- The Dependency Inversion Principle guides us to avoid dependencies on concrete types and to strive for abstractions.
- Factories are a powerful technique for coding to abstractions, not concrete classes

Another Example

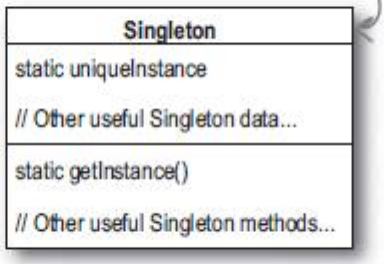


the Singleton Pattern

One of a Kind Objects

The Singleton Pattern ensures a class has only one instance, and provides a global point of access to it.

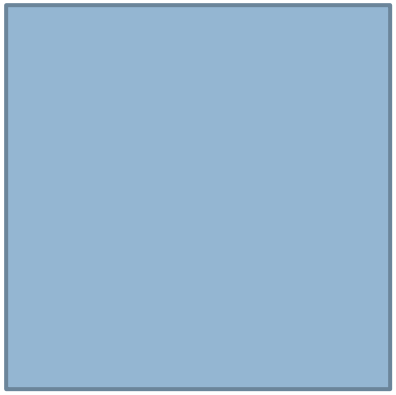
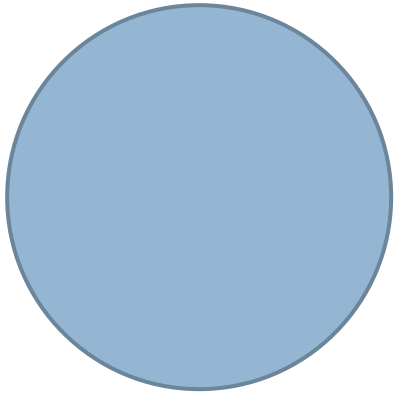
The getInstance() method is static, which means it's a class method, so you can conveniently access this method from anywhere in your code using Singleton.getInstance(). That's just as easy as accessing a global variable, but we get benefits like lazy instantiation from the Singleton.

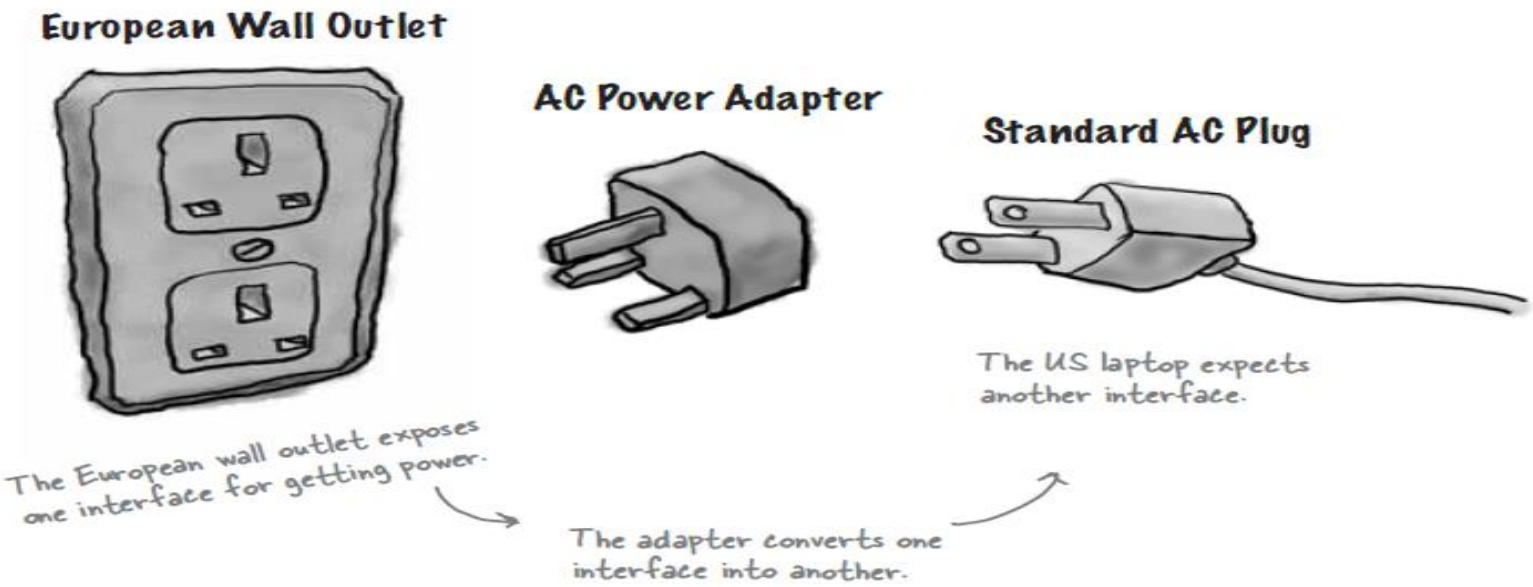


The uniqueInstance class variable holds our one and only instance of Singleton.

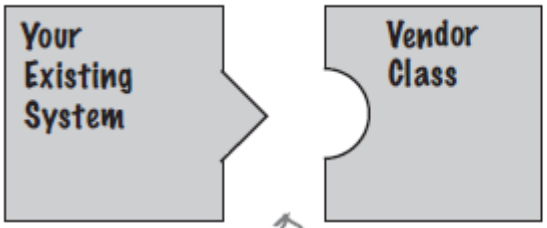
A class implementing the Singleton Pattern is more than a Singleton; it is a general purpose class with its own set of data and methods.

the Adapter

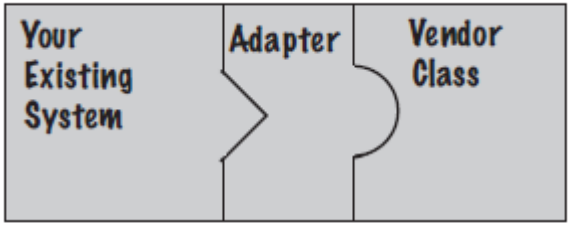
Putting  Into 



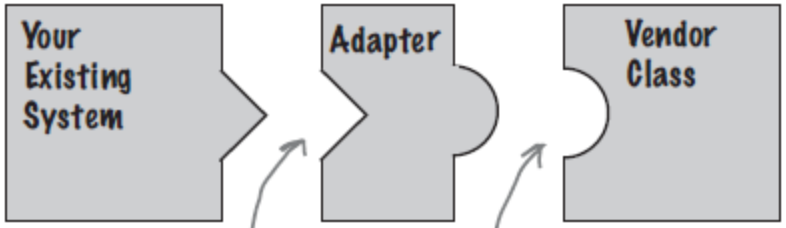
Object oriented adapters



Their interface doesn't match the one you've written your code against. This isn't going to



No code changes. New code. No code changes.



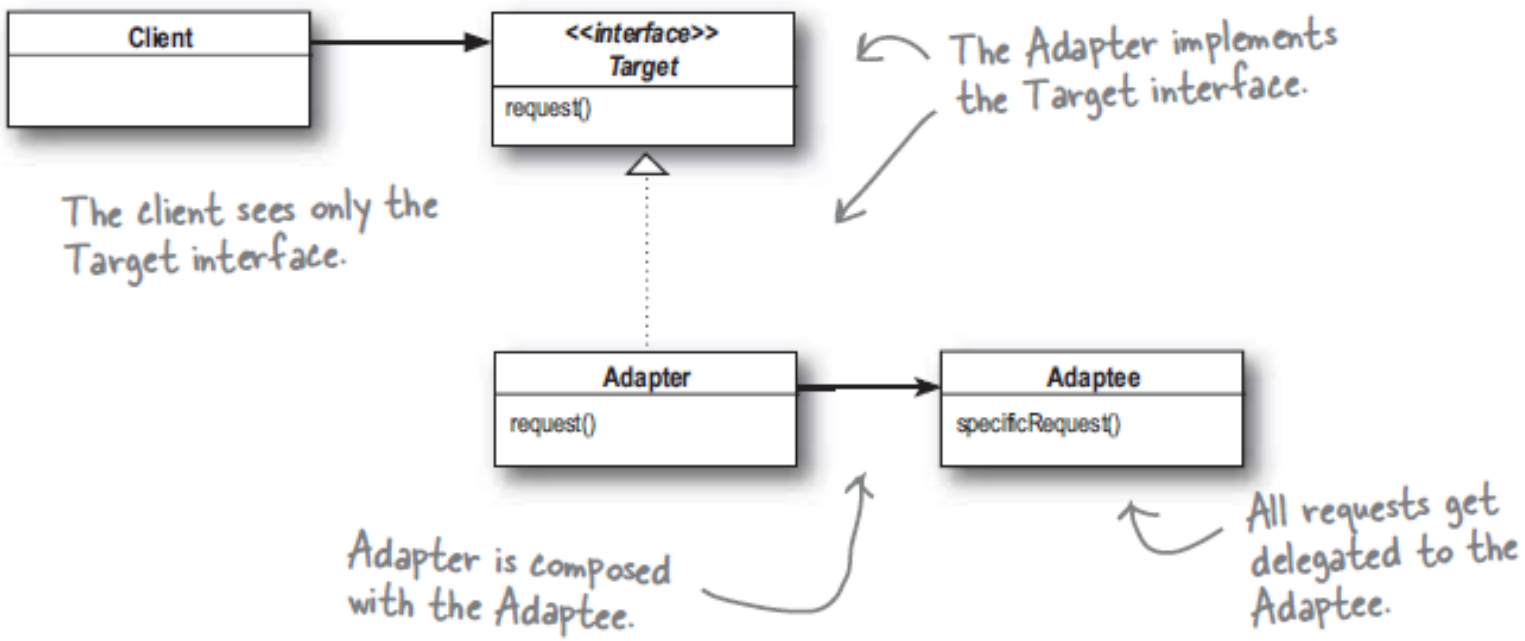
The adapter implements the interface your classes expect.

And talks to the vendor interface to service your requests.

How It Works

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

1. **The client makes a request to the adapter by calling a method on it using the target interface.**
2. **The adapter translates that request into one or more calls on the adaptee using the adaptee interface.**
3. **The client receives the results of the call and never knows there is an adapter doing the translation.**



A Duck Or a Turkey

If it walks like a duck and quacks like a duck, then it ~~must~~ might be a ~~duck~~ turkey wrapped with a duck adapter...

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

This time around, our ducks implement a Duck interface that allows Ducks to quack and fly.

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

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

Simple implementations: the duck just prints out what it is doing.

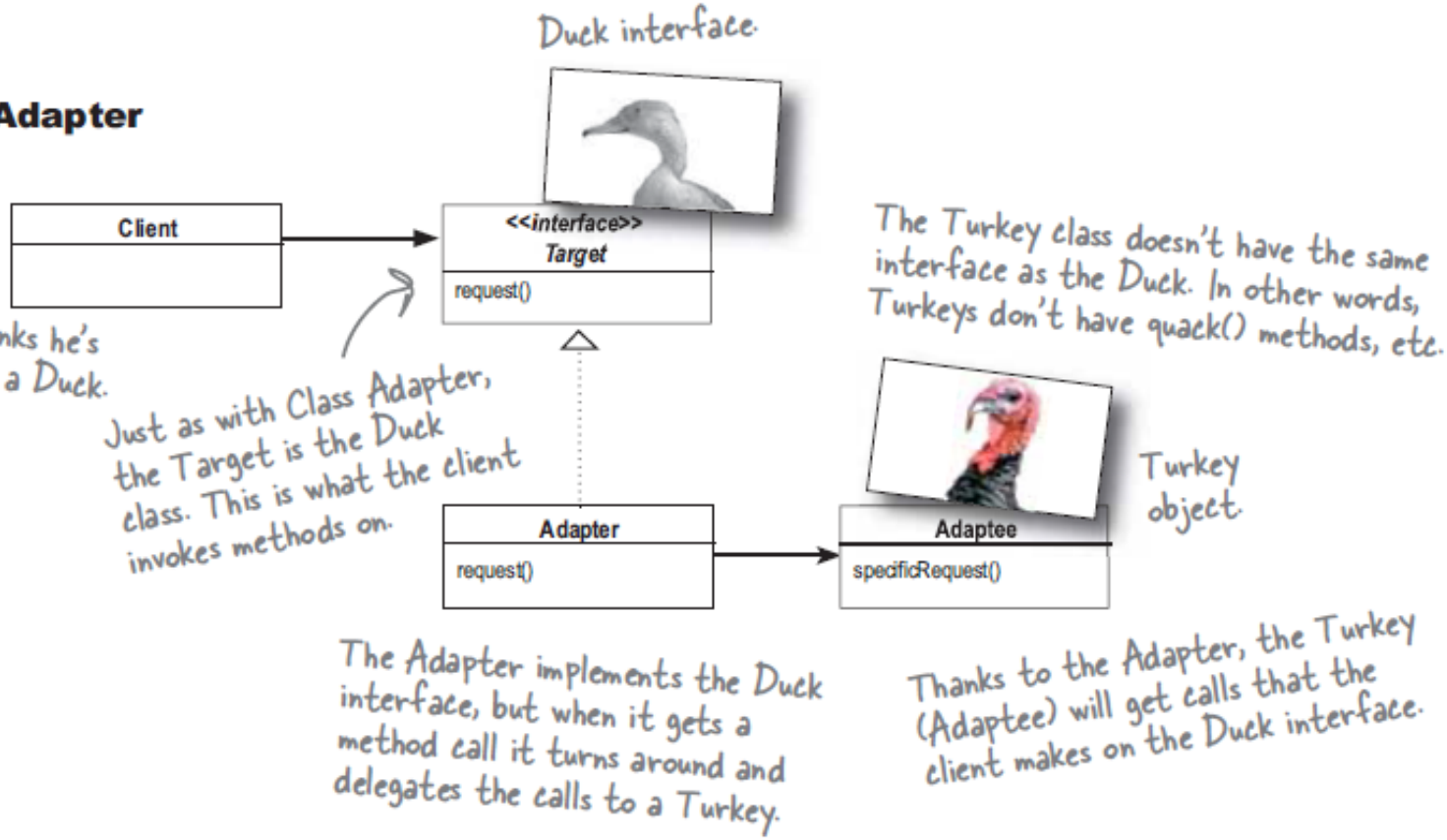
```
public interface Turkey {
    public void gobble();
    public void fly();
}
```

Turkeys don't quack, they gobble.

Turkeys can fly, although they can only fly short distances.

the adapter pattern

Object Adapter

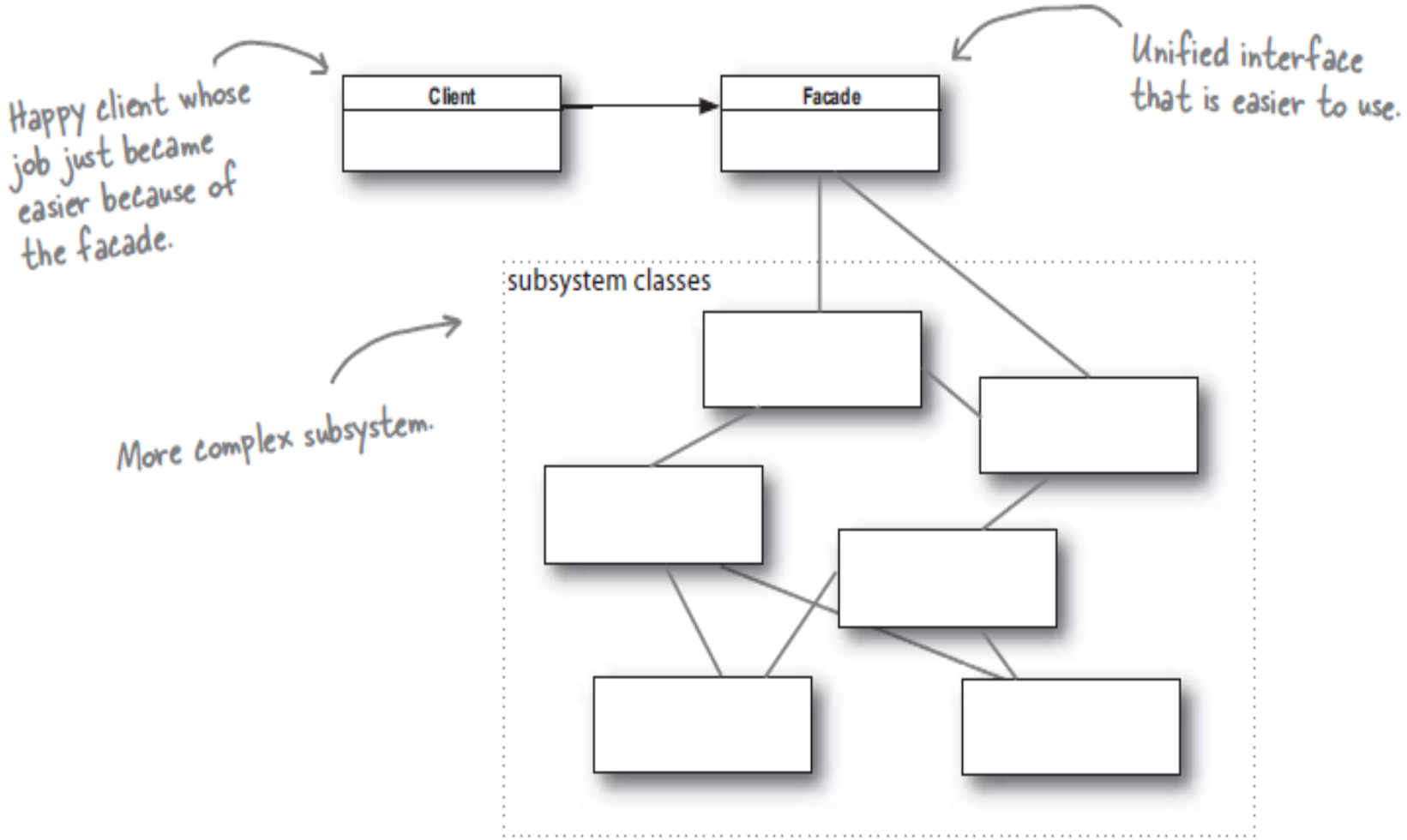


Façade Pattern

- Role

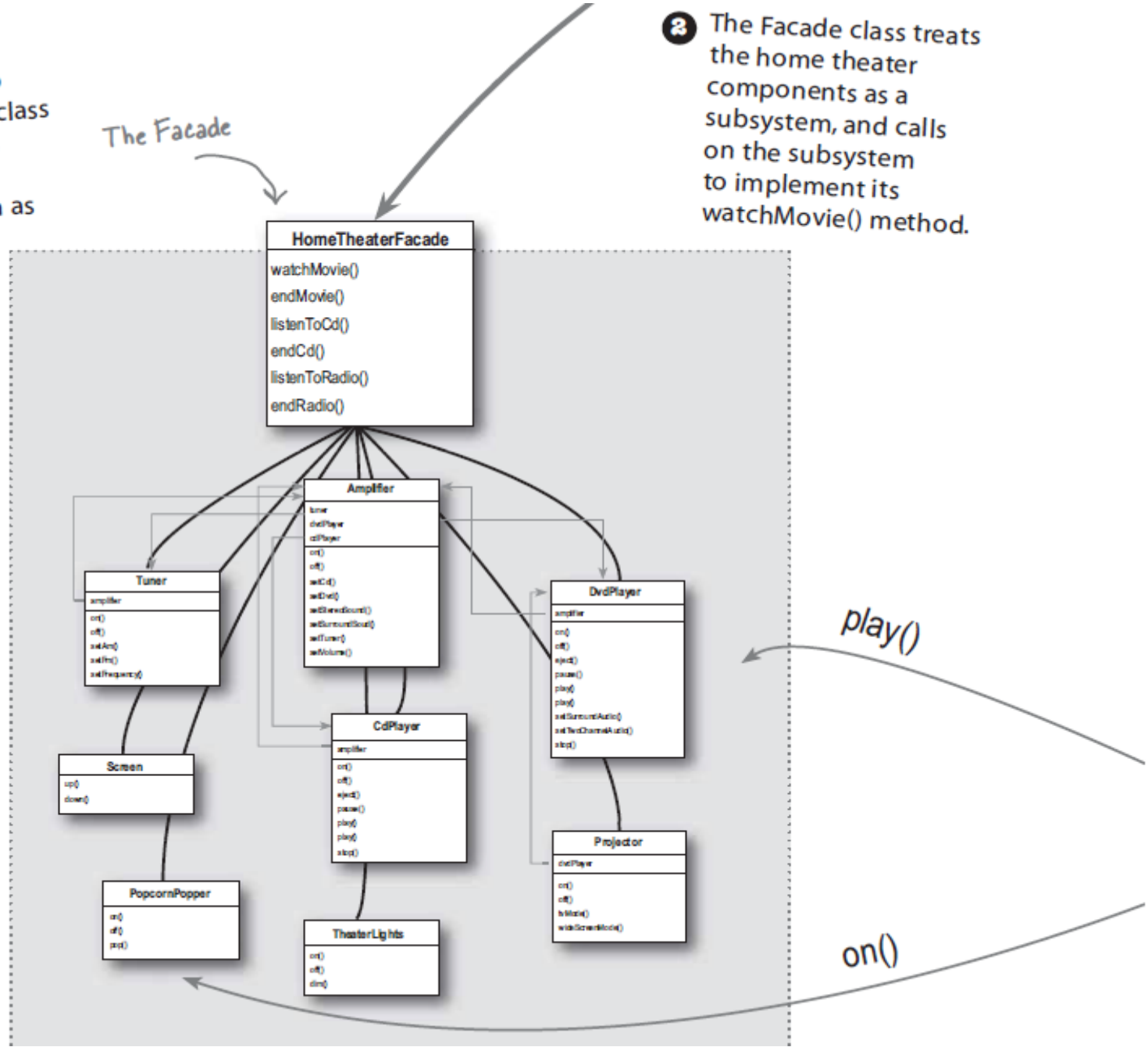
  The role of the Façade pattern is to provide different high-level views of subsystems whose details are hidden from users. In general, the operations that might be desirable from a user's perspective could be made up of different selections of parts of the subsystems.

UML


Happy client whose job just became easier because of the facade.
Client
Facade
Unified interface that is easier to use.
subsystem classes
More complex subsystem.

1. Okay, time to create a Facade for the home theater system. To do this we create a new class HomeTheaterFacade, which exposes a few simple methods such as watchMovie().

2. The Facade class treats the home theater components as a subsystem, and calls on the subsystem to implement its watchMovie() method.



Watching a movie (the hard way)

```
popper.on();
popper.pop();

lights.dim(10);

screen.down();

projector.on();
projector.setInput(dvd);
projector.wideScreenMode();

amp.on();
amp.setDvd(dvd);
amp.setSurroundSound();
amp.setVolume(5);

dvd.on();
dvd.play(movie);
```

Turn on the popcorn popper and start popping...

Dim the lights to 10%...

Put the screen down...

Turn on the projector and put it in wide screen mode for the movie...

Turn on the amp, set it to DVD, put it in surround sound mode and set the volume to 5...

Turn on the DVD player... and FINALLY, play the movie!

Six different classes involved!

Use the Façade pattern when…

A system has several identifiable subsystems and:

- The abstractions and implementations of a subsystem are tightly coupled.
- The system evolves and gets more complex, but early adopters might want to retain their simple views.
- You want to provide alternative novice, intermediate, and "power user" interfaces.
- There is a need for an entry point to each level of layered software.

Choose the Façade you need…

Opaque

Subsystem operations can only be called through the Façade.

Transparent

Subsystem operations can be called directly as well as through the Façade.

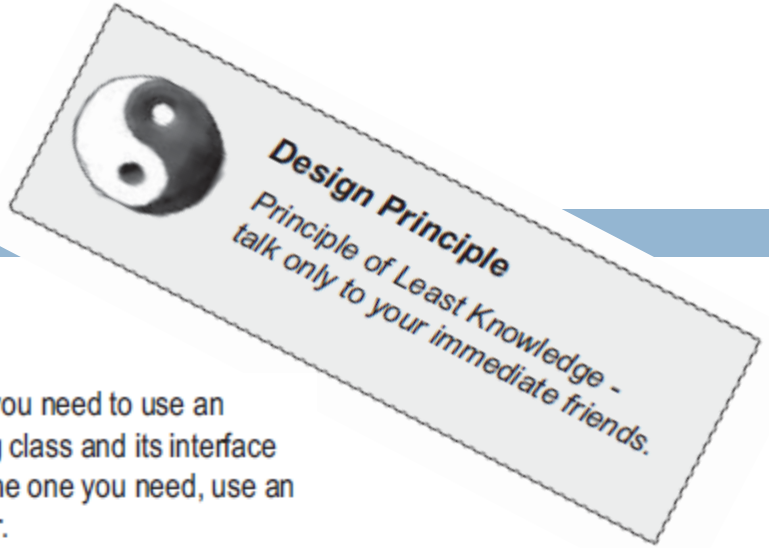
Singleton

Only one instance of the Façade is meaningful.

Last Few Words

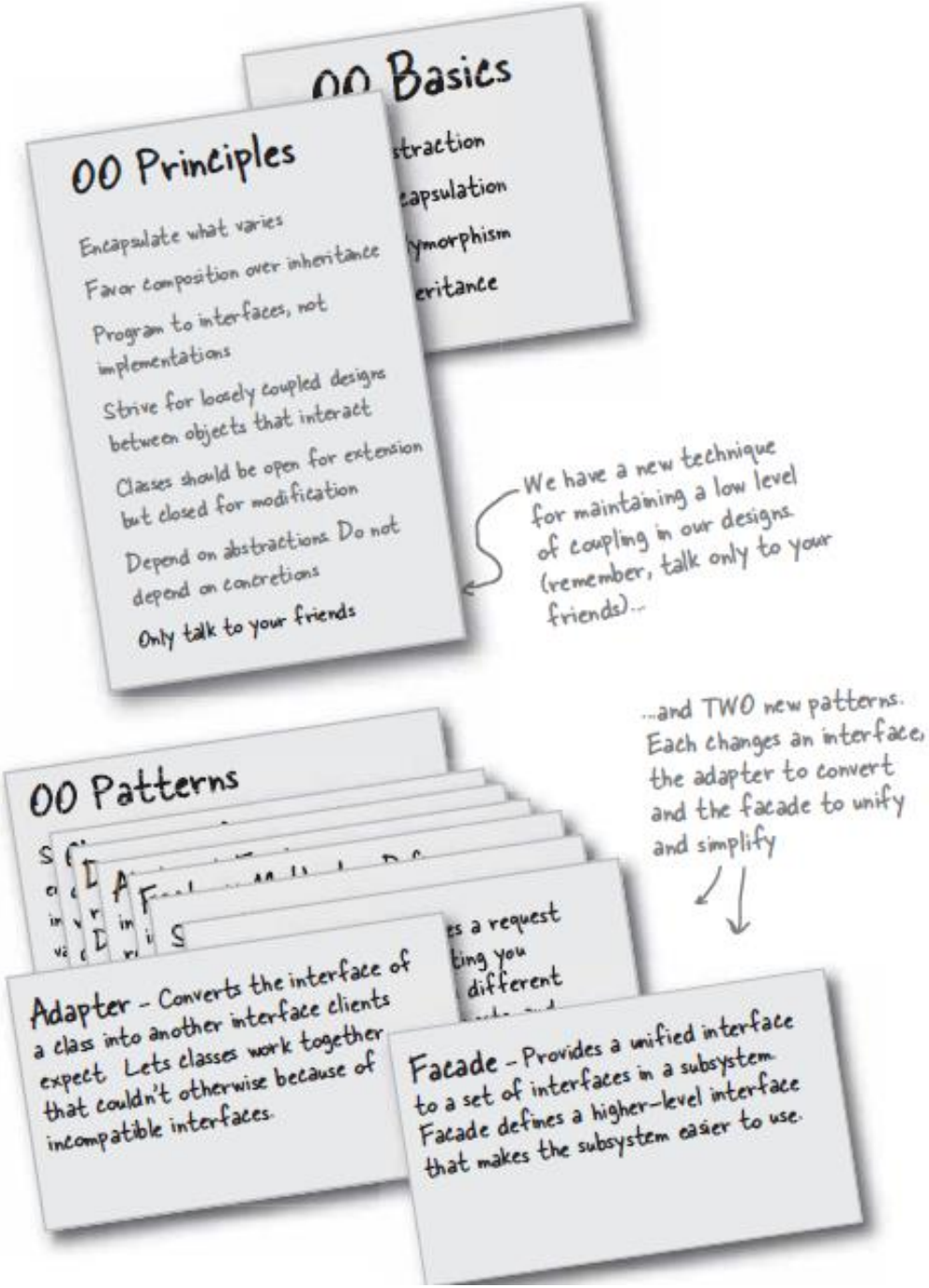
A facade not only simplifies an interface, it decouples a client from a subsystem of components.

Facades and adapters may wrap multiple classes, but a facade's intent is to simplify, while an adapter's is to convert the interface to something different.



- When you need to use an existing class and its interface is not the one you need, use an adapter.
- When you need to simplify and unify a large interface or complex set of interfaces, use a facade.
- An adapter changes an interface into one a client expects.
- A facade decouples a client from a complex subsystem.
- Implementing an adapter may require little work or a great deal of work depending on the size and complexity of the target interface.
- Implementing a facade requires that we compose the facade with its subsystem and use delegation to perform the work of the facade.
- There are two forms of the Adapter Pattern: object and class adapters. Class adapters require multiple inheritance.
- You can implement more than one facade for a subsystem.
- An adapter wraps an object to change its interface, a decorator wraps an object to add new behaviors and responsibilities, and a facade "wraps" a set of objects to simplify.

More Last Words



Starbuzz Coffee Barista Training Manual

Baristas! Please follow these recipes precisely when preparing Starbuzz beverages.

Starbuzz Coffee Recipe

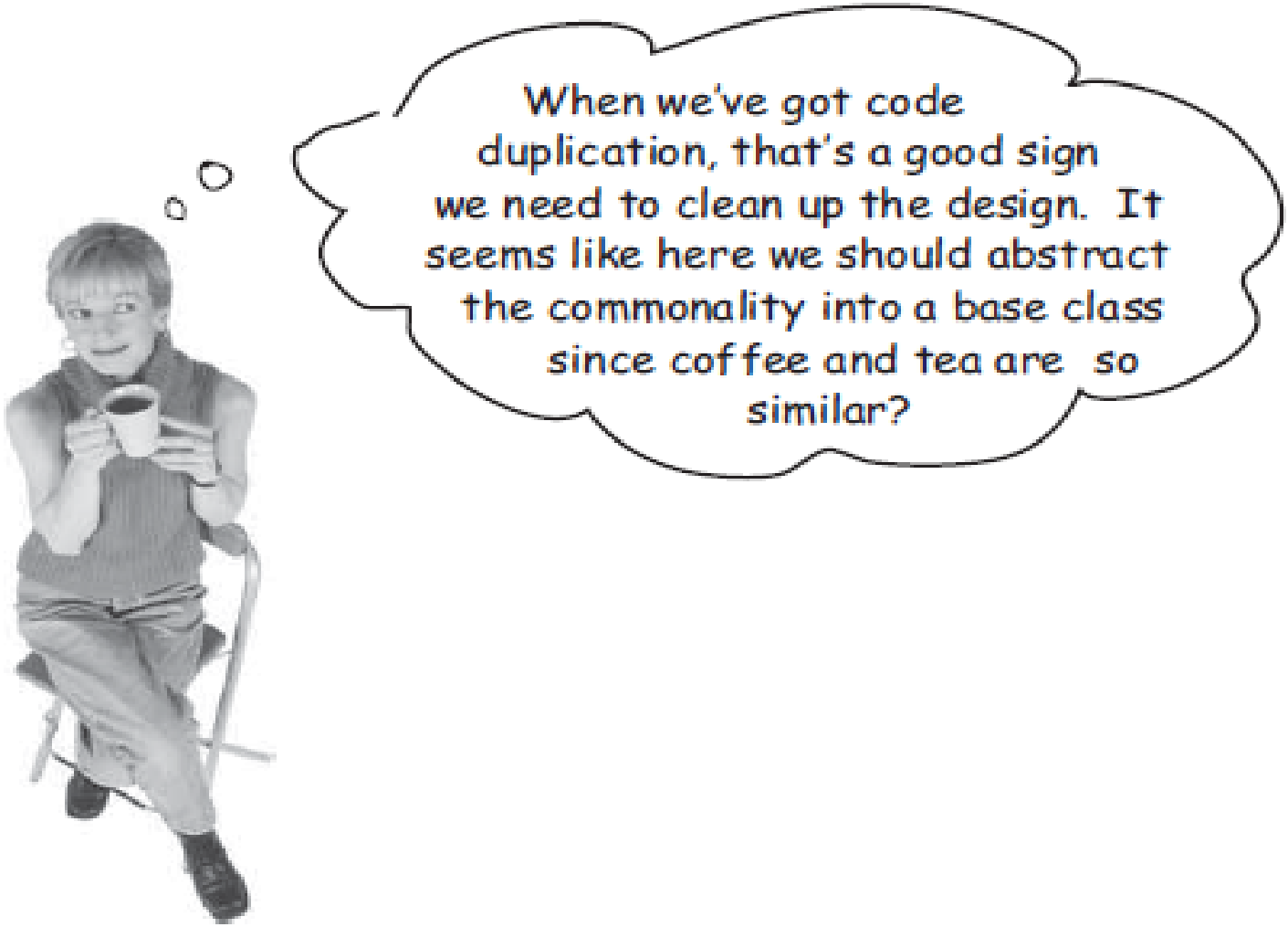
(1) Boil some water
(2) Brew coffee in boiling water
(3) Pour coffee in cup
(4) Add sugar and milk

Starbuzz Tea Recipe

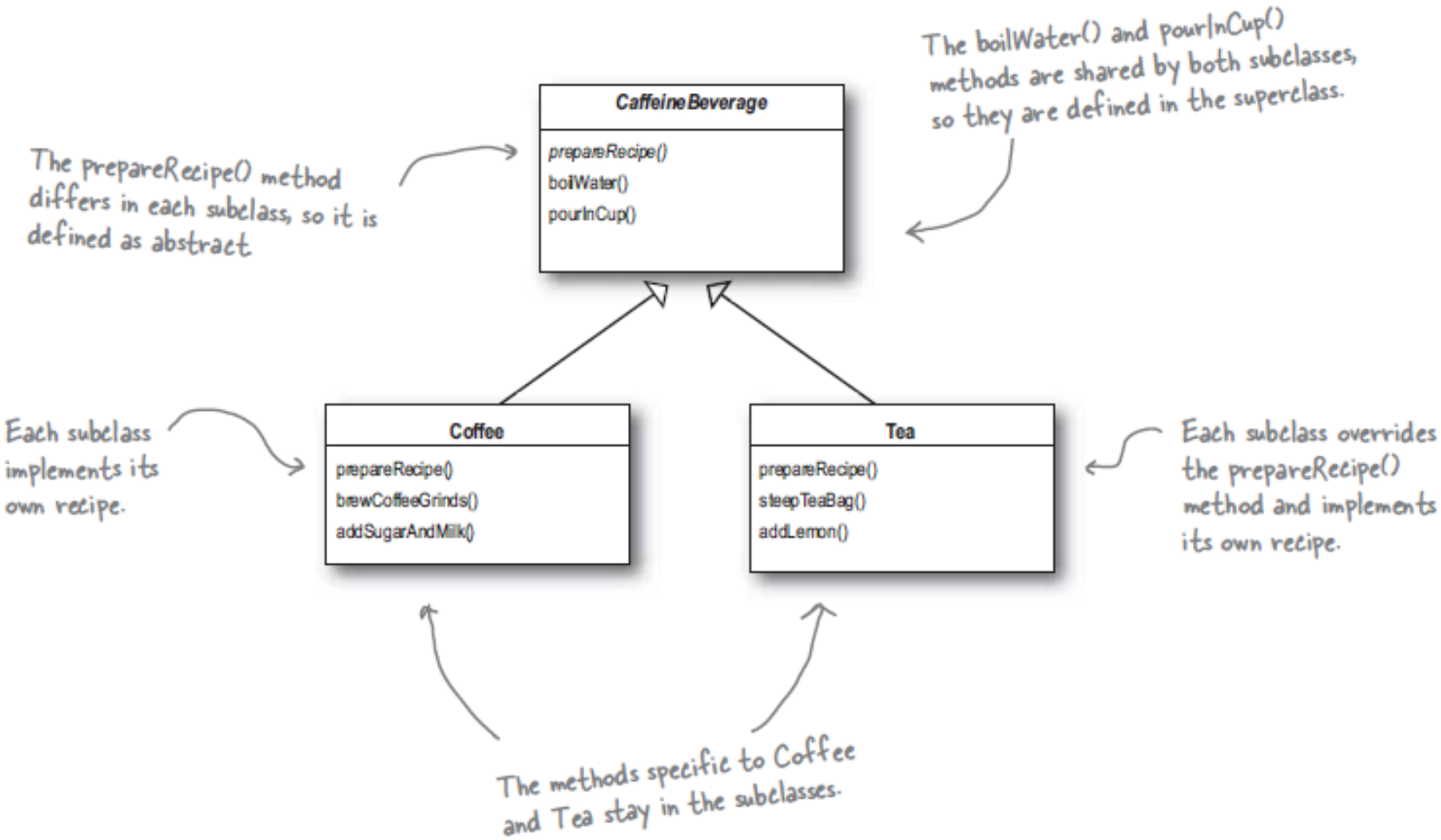
(1) Boil some water
(2) Steep tea in boiling water
(3) Pour tea in cup
(4) Add lemon

All recipes are Starbuzz Coffee trade secrets and should be kept strictly confidential.

Template Method



Abstraction



Abstracting prepareRecipe()

Notice that both recipes follow the same algorithm:

1. Boil some water.
2. Use the hot water to extract the coffee or tea.
3. Pour the resulting beverage into a cup.
4. Add the appropriate condiments to the beverage.

These aren't abstracted, but are the same, they just apply to different beverages.

These two are already abstracted into the base class.

Coffee

```
void prepareRecipe() {
    boilWater();
    brewCoffeeGrinds();
    pourInCup();
    addSugarAndMilk();
}
```

Tea

```
void prepareRecipe() {
    boilWater();
    steepTeaBag();
    pourInCup();
    addLemon();
}
```

Putting Prepare Recipe in Place

CaffeineBeverage is abstract, just like in the class design.

Now, the same prepareRecipe() method will be used to make both Tea and Coffee.

```
public abstract class CaffeineBeverage {

        void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
                        ("Boiling water");
    }

    void pourInCup() {
                        ("Pouring into cup");
    }
}
```

Because Coffee and Tea handle these methods in different ways, they're going to have to be declared as abstract. Let the subclasses worry about that stuff!

Remember, we moved these into the CaffeineBeverage class (back in our class diagram).

Tea Or Coffee

```
public class Tea        CaffeineBeverage {
    public void brew() {
        System.out.println("Steeping the tea");
    }
    public void addCondiments() {
        System.out.println("Adding Lemon");
    }
}
```

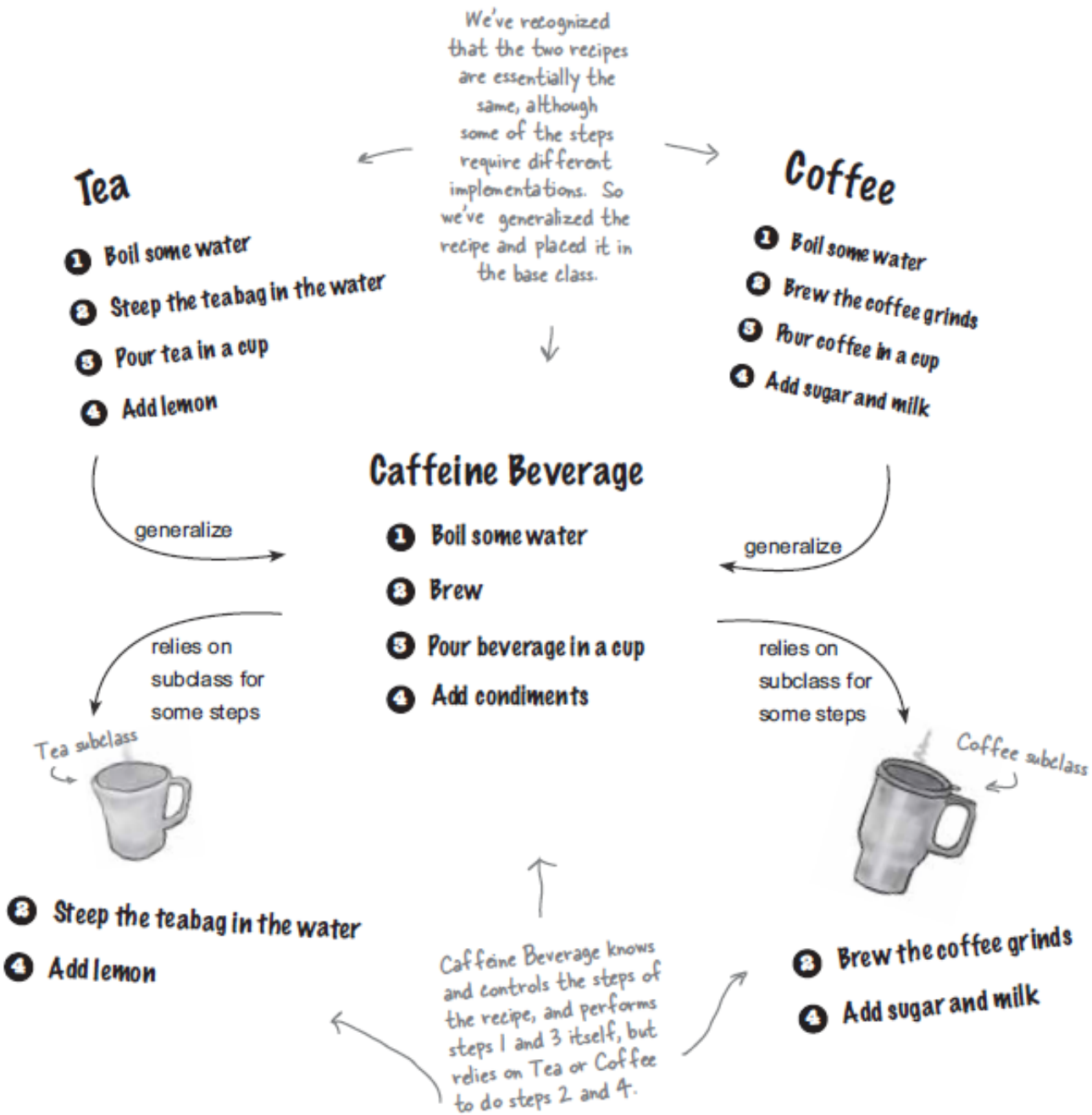
As in our design, Tea and Coffee now extend CaffeineBeverage.

Tea needs to define brew() and addCondiments() – the two abstract methods from Beverage.

Same for Coffee, except Coffee deals with coffee, and sugar and milk instead of tea bags and lemon.

```
public class Coffee        CaffeineBeverage {
    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }
    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

What have we done?



Meet the Template Method

```
public abstract class CaffeineBeverage {

    void final prepareRecipe() {

        boilWater();

        brew();

        pourInCup();

        addCondiments();

    }

    abstract void brew();

    abstract void addCondiments();
```

prepareRecipe() is our **template method**. Why?

Because:

(1) It is a method, after all.

(2) It serves as a template for an algorithm, in this case, an algorithm for making caffeinated beverages.

In the template, each step of the algorithm is represented by a method.

Some methods are handled by this class...

...and some are handled by the subclass.

The methods that need to be supplied by a subclass are declared abstract.

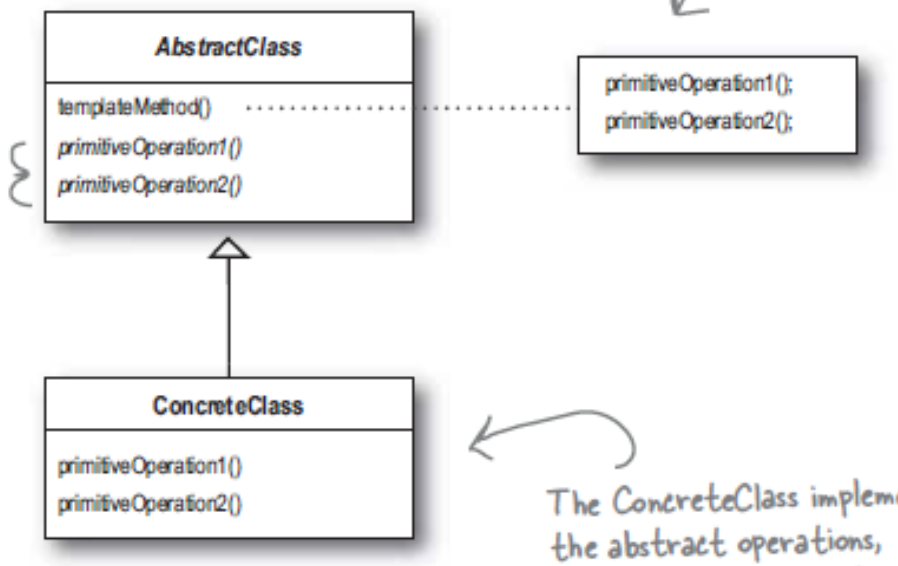
Template Method

The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

The template method makes use of the primitiveOperations to implement an algorithm. It is decoupled from the actual implementation of these operations.

The AbstractClass contains the template method.

...and abstract versions of the operations used in the template method.



There may be many ConcreteClasses, each implementing the full set of operations required by the template method.

The ConcreteClass implements the abstract operations, which are called when the templateMethod() needs them.

Hook Methods

```
abstract class AbstractClass {

        void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
        hook();
    }

    abstract void primitiveOperation1();

    abstract void primitiveOperation2();

        void concreteOperation() {
        // implementation here
    }

    void hook() {}

}
```

We still have our primitive methods; these are abstract and implemented by concrete subclasses.

A concrete operation is defined in the abstract class.

A concrete method, but it does nothing!

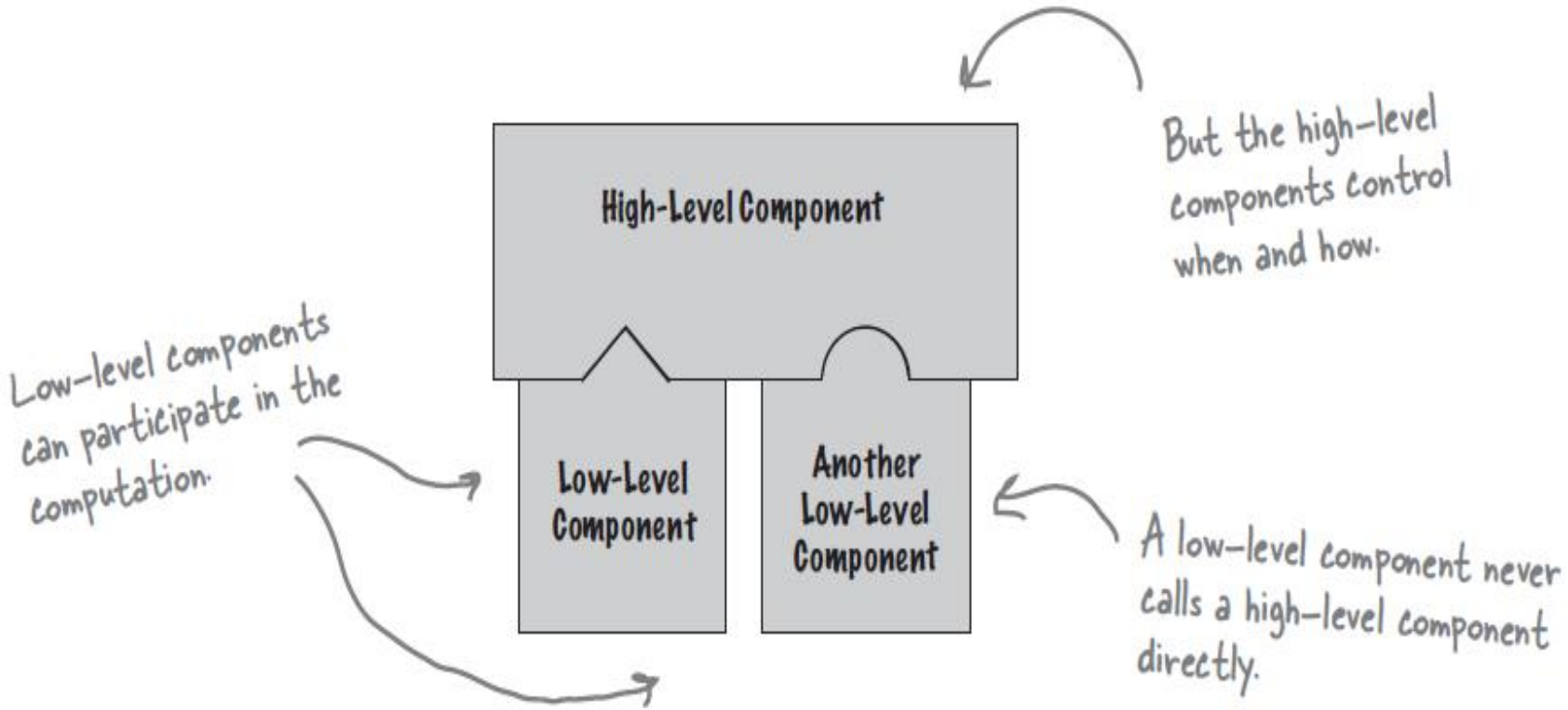
We can also have concrete methods that do nothing by default; we call these "hooks." Subclasses are free to override these but don't have to. We're going to see how these are useful on the next page.

Can We Do Better ?

IComparable ?

Partial Methods?

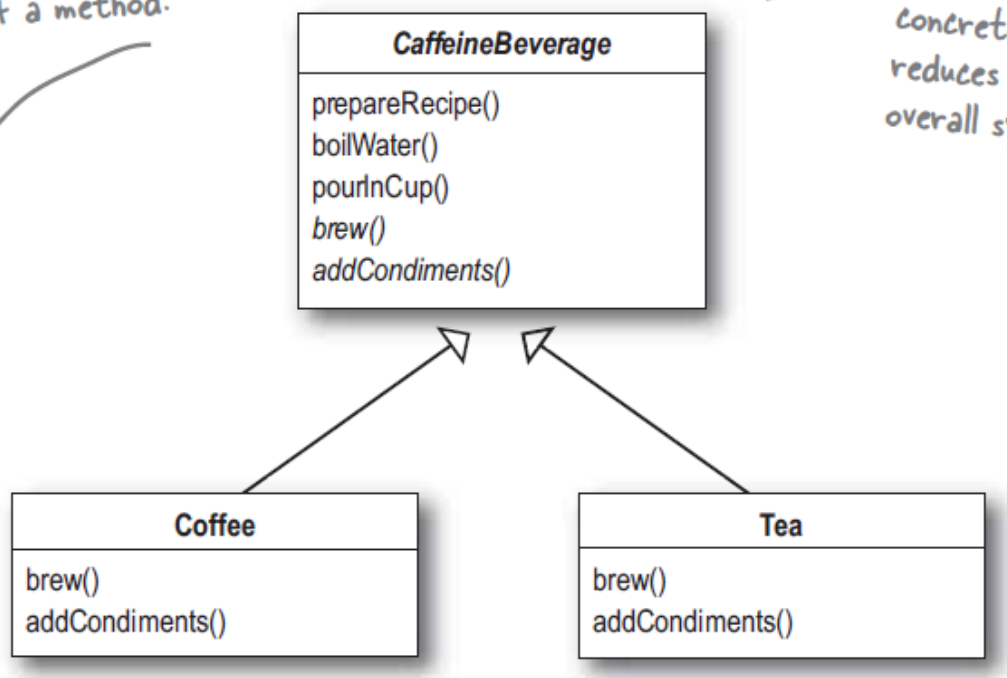
The Hollywood Principle



The Hollywood Principle

CaffeineBeverage is our high-level component. It has control over the algorithm for the recipe, and calls on the subclasses only when they're needed for an implementation of a method.

Clients of beverages will depend on the CaffeineBeverage abstraction rather than a concrete Tea or Coffee, which reduces dependencies in the overall system.

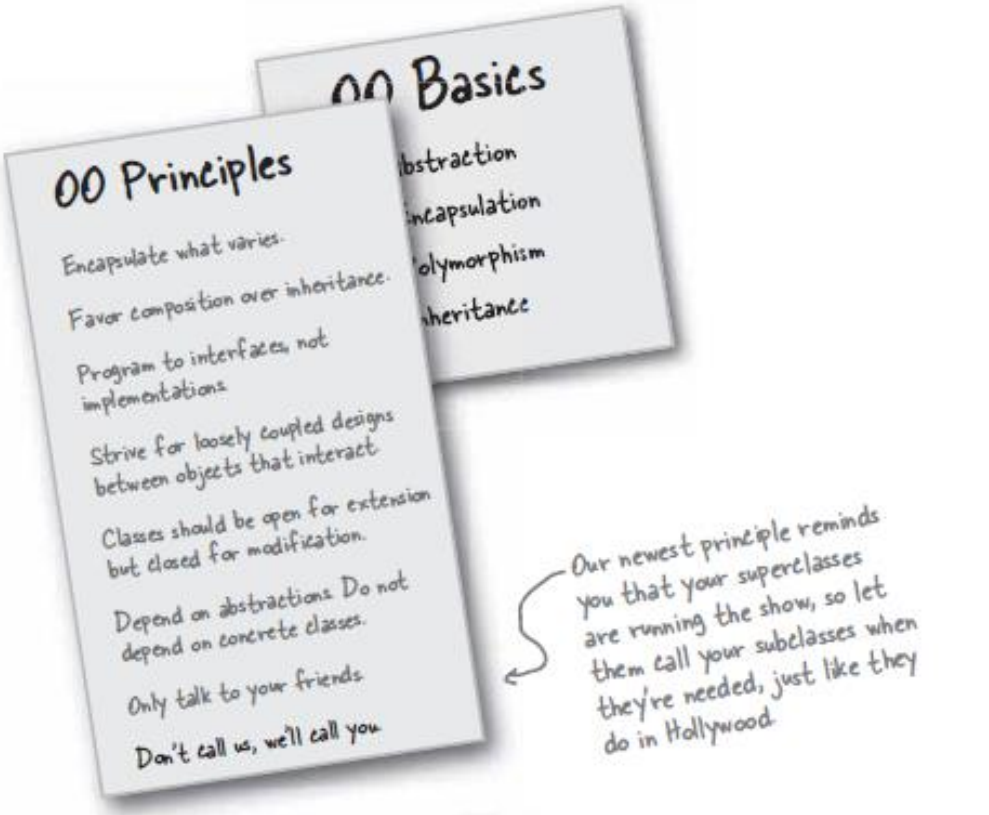


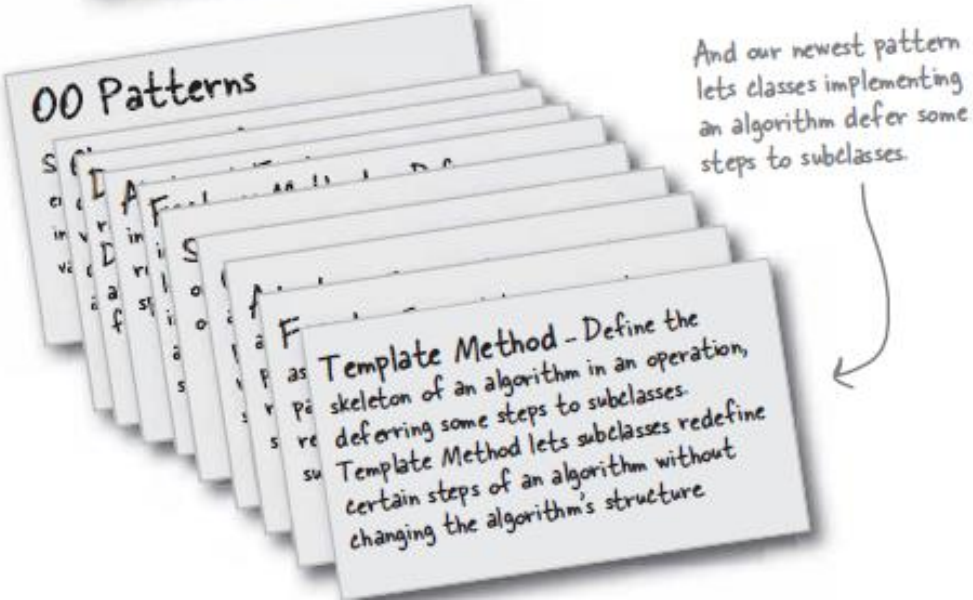
The subclasses are used simply to provide implementation details.

Tea and Coffee never call the abstract class directly without being "called" first.

Final Word

- A "template method" defines the steps of an algorithm, deferring to subclasses for the implementation of those steps.
- The Template Method Pattern gives us an important technique for code reuse.
- The template method's abstract class may define concrete methods, abstract methods and hooks.
- Abstract methods are implemented by subclasses.
- Hooks are methods that do nothing or default behavior in the abstract class, but may be overridden in the subclass.
- The Hollywood Principle guides us to put decision-making in high-level modules that can decide how and when to call low level modules.
- You'll see lots of uses of the Template Method Pattern in real world code, but don't expect it all (like any pattern) to be designed "by the book."
- The Strategy and Template Method Patterns both encapsulate algorithms, one by inheritance and one by composition.
- The Factory Method is a specialization of Template Method.





the State Pattern

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

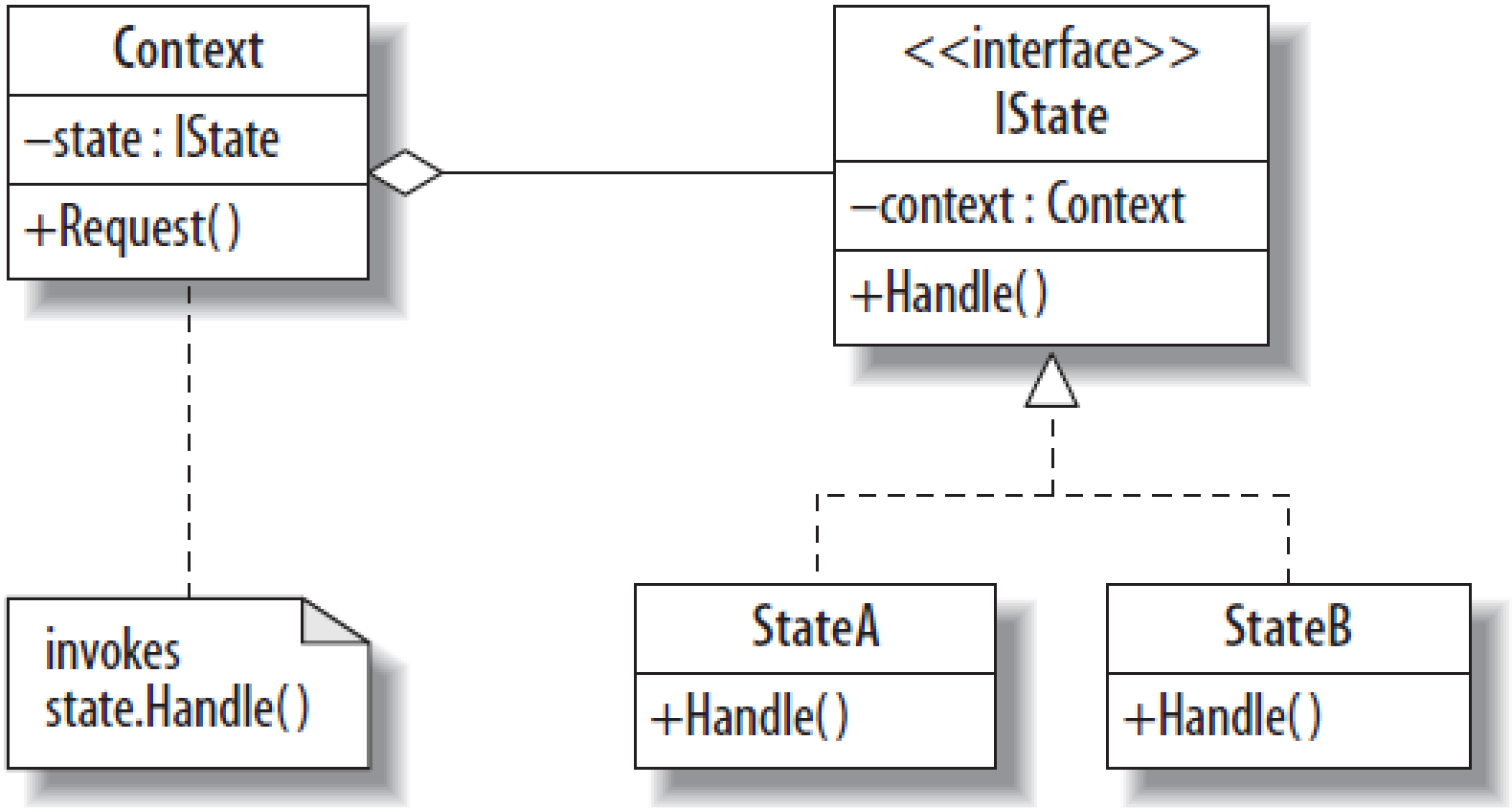
A little known fact: the Strategy and State Patterns were twins separated at birth.

- the State pattern, can be seen as a dynamic version of the Strategy pattern.
- When the state inside an object changes, it can change its behavior by switching to a set of different operations.
- This is achieved by an object variable changing its subclass, within a hierarchy.

Applicability

- 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.

UML


Context
–state : IState
+Request()
<<interface>>
IState
–context : Context
+Handle()
invokes
state.Handle()
StateA
+Handle()
StateB
+Handle()

Participants

- **Context**
  - defines the interface of interest to clients.
  - maintains an instance of a ConcreteState subclass that defines the current state.
- **State**
  - defines an interface for encapsulating the behavior associated with a particular state of the Context.
- **ConcreteState subclasses**
  - each subclass implements a behavior associated with a state of the Context.

Collaborations

- Context **delegates** state-specific requests to the current ConcreteState object.
- A context **may pass itself as an argument** to the State object handling the request. This lets the State object access the context if necessary.
- **Context** is the **primary interface for clients**. Clients can configure a context with State objects. Once a context is configured, its clients don't have to deal with the State objects directly.
- Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.

Who defines the state transitions

- The State pattern does not specify which participant defines the criteria for state transitions.
- If the **criteria are fixed**, then they can be implemented entirely in the Context. It is generally more flexible and appropriate,
- however, to let the State subclasses themselves specify their successor state and when to make the transition. This requires adding an interface to the Context that lets State objects set the Context's current state explicitly.

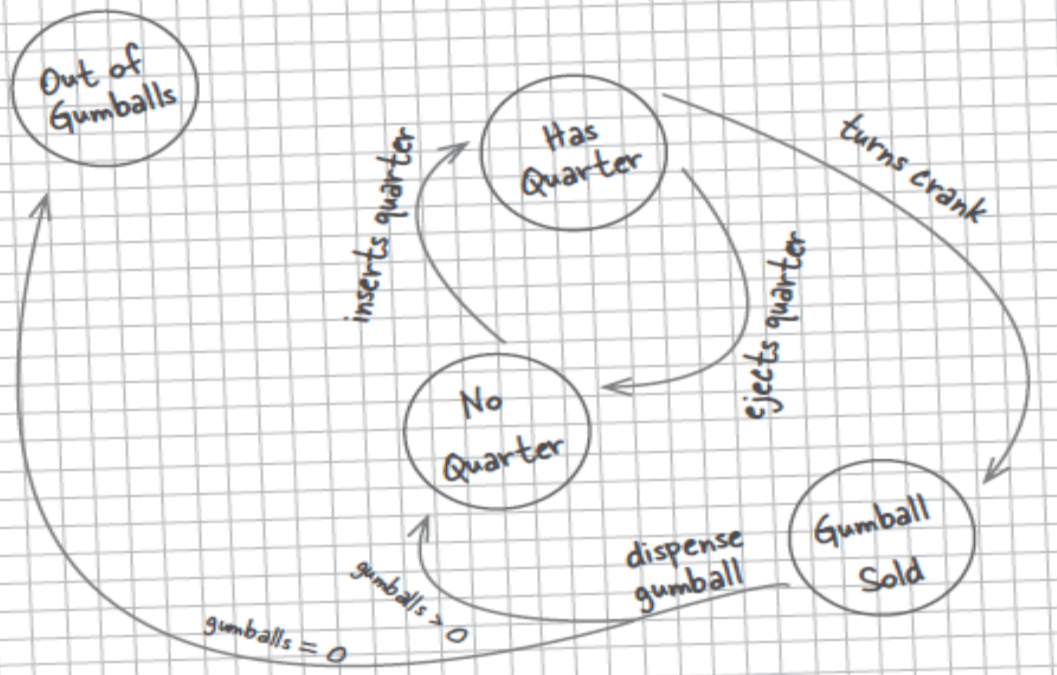
The Gumball Machine

Mighty Gumball, Inc.

Where the Gumball Machine is Never Half Empty

Here's the way we think the gumball machine controller needs to work. We're hoping you can implement this in Java for us! We may be adding more behavior in the future, so you need to keep the design as flexible and maintainable as possible!

– Mighty Gumball Engineers



State machines 101

1. First, gather up your states:

2. Next, create an instance variable to hold the current state, and define values for each of the states:

3. Now we gather up all the actions that can happen in the system:

inserts quarter

turns crank

ejects quarter

dispense

These actions are the gumball machine's interface – the things you can do with it.

Dispense is more of an internal action the machine invokes on itself.

Looking at the diagram, invoking any of these actions causes a state transition.

4 Now we create a class that acts as the state machine. For each action, we create a method that uses conditional statements to determine what behavior is appropriate in each state. For instance, for the insert quarter action, we might write a method like this:

```
public void insertQuarter() {

    if (state == HAS_QUARTER) {

                        ("You can't insert another quarter");

    } else if (state == SOLD_OUT) {

                        ("You can't insert a quarter, the machine is sold out");

    } else if (state == SOLD) {

                        ("Please wait, we're already giving you a gumball");

    } else if (state == NO_QUARTER) {

        state = HAS QUARTER;
                        ("You inserted a quarter");

    }
}
```

Each possible state is checked with a conditional statement...

...and exhibits the appropriate behavior for each possible state...

...but can also transition to other states, just as depicted in the diagram.

Getting The Pieces Together

```
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;

    public GumballMachine(int count) {
        this.count = count;
        if (count > 0) {
            state = NO_QUARTER;
        }
    }
```

Here are the four states; they match the states in Mighty Gumball's state diagram.

Here's the instance variable that is going to keep track of the current state we're in. We start in the SOLD_OUT state.

We have a second instance variable that keeps track of the number of gumballs in the machine.

The constructor takes an initial inventory of gumballs. If the inventory isn't zero, the machine enters state NO_QUARTER, meaning it is waiting for someone to insert a quarter, otherwise it stays in the SOLD_OUT state.

The States

Now we start implementing the actions as methods....

When a quarter is inserted, if....

```
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");
    }
}
```

a quarter is already inserted we tell the customer;

otherwise we accept the quarter and transition to the HAS_QUARTER state.

If the customer just bought a gumball he needs to wait until the transaction is complete before inserting another quarter.

and if the machine is sold out, we reject the quarter.

The States

Now, if the customer tries to remove the quarter...

```
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");
    }
}
```

If there is a quarter, we return it and go back to the NO_QUARTER state.

Otherwise, if there isn't one we can't give it back.

You can't eject if the machine is sold out, it doesn't accept quarters!

If the customer just turned the crank, we can't give a refund; he already has the gumball!

The States

The customer tries to turn the crank...

give a refund; he already has the gumball!

```
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();
    }
}
```

Someone's trying to cheat the machine.

We need a quarter first.

We can't deliver gumballs; there are none.

Success! They get a gumball. Change the state to SOLD and call the machine's dispense() method.

The States

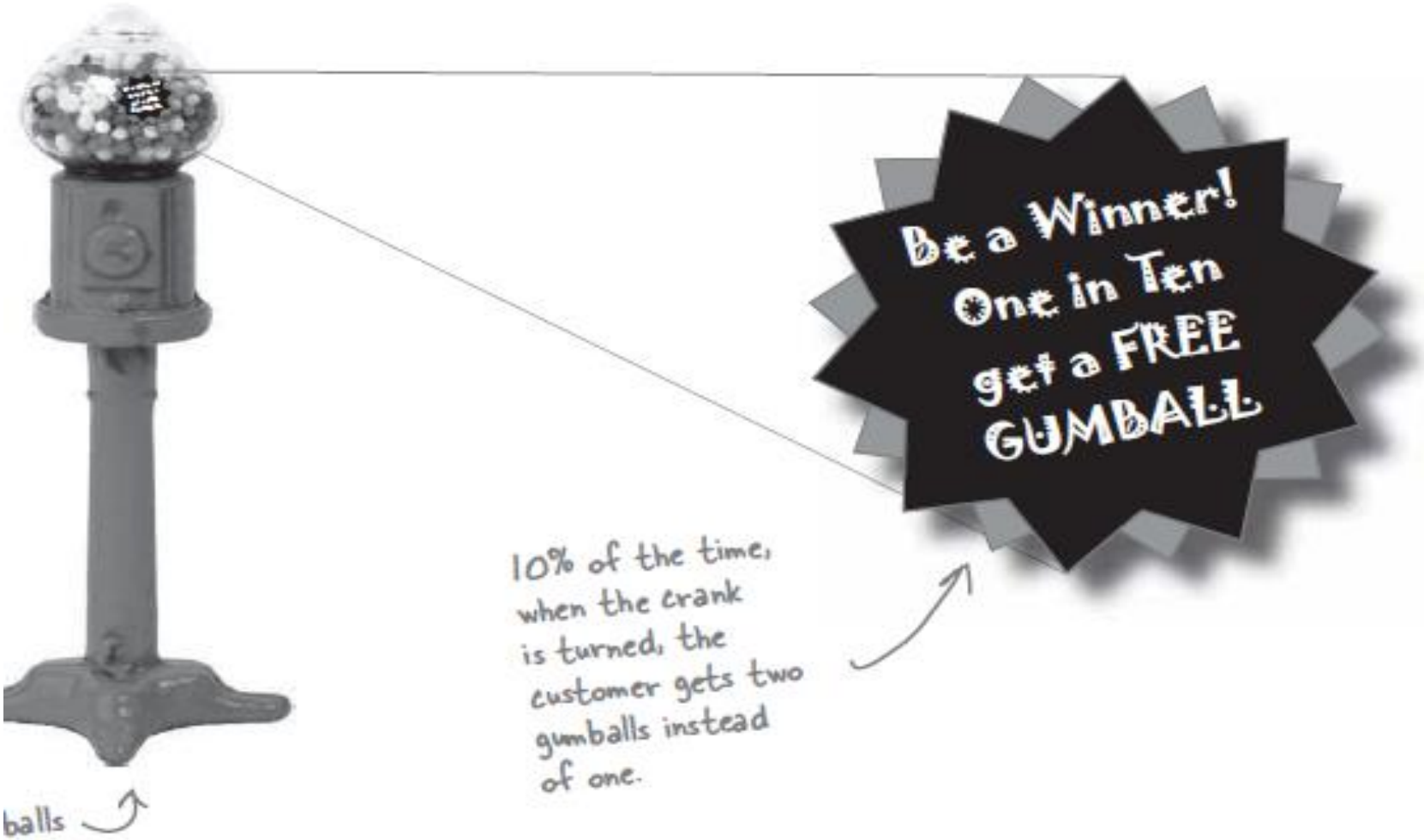
```
public void dispense() {
    if (state == SOLD) {
        System.out.println("A gumball comes rolling out the slot");
        count = count - 1;
        if (count == 0) {
            System.out.println("Oops, out of gumballs!");
            state = SOLD_OUT;
        } else {
            state = NO_QUARTER;
        }
    } 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");
    }
}
```

We're in the SOLD state; give 'em a gumball!

Here's where we handle the "out of gumballs" condition: If this was the last one, we set the machine's state to SOLD_OUT; otherwise, we're back to not having a quarter.

None of these should ever happen, but if they do, we give 'em an error, not a gumball.

Change Request



Changing the Code

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

First, you'd have to add a new WINNER state here. That isn't too bad...

```
public void insertQuarter() {
    // insert quarter code here
}

public void ejectQuarter() {
    // eject quarter code here
}

public void turnCrank() {
    // turn crank code here
}

public void dispense() {
    // dispense code here
}
```

... but then, you'd have to add a new conditional in every single method to handle the WINNER state; that's a lot of code to modify.

turnCrank() will get especially messy, because you'd have to add code to check to see whether you've got a WINNER and then switch to either the WINNER state or the SOLD state.

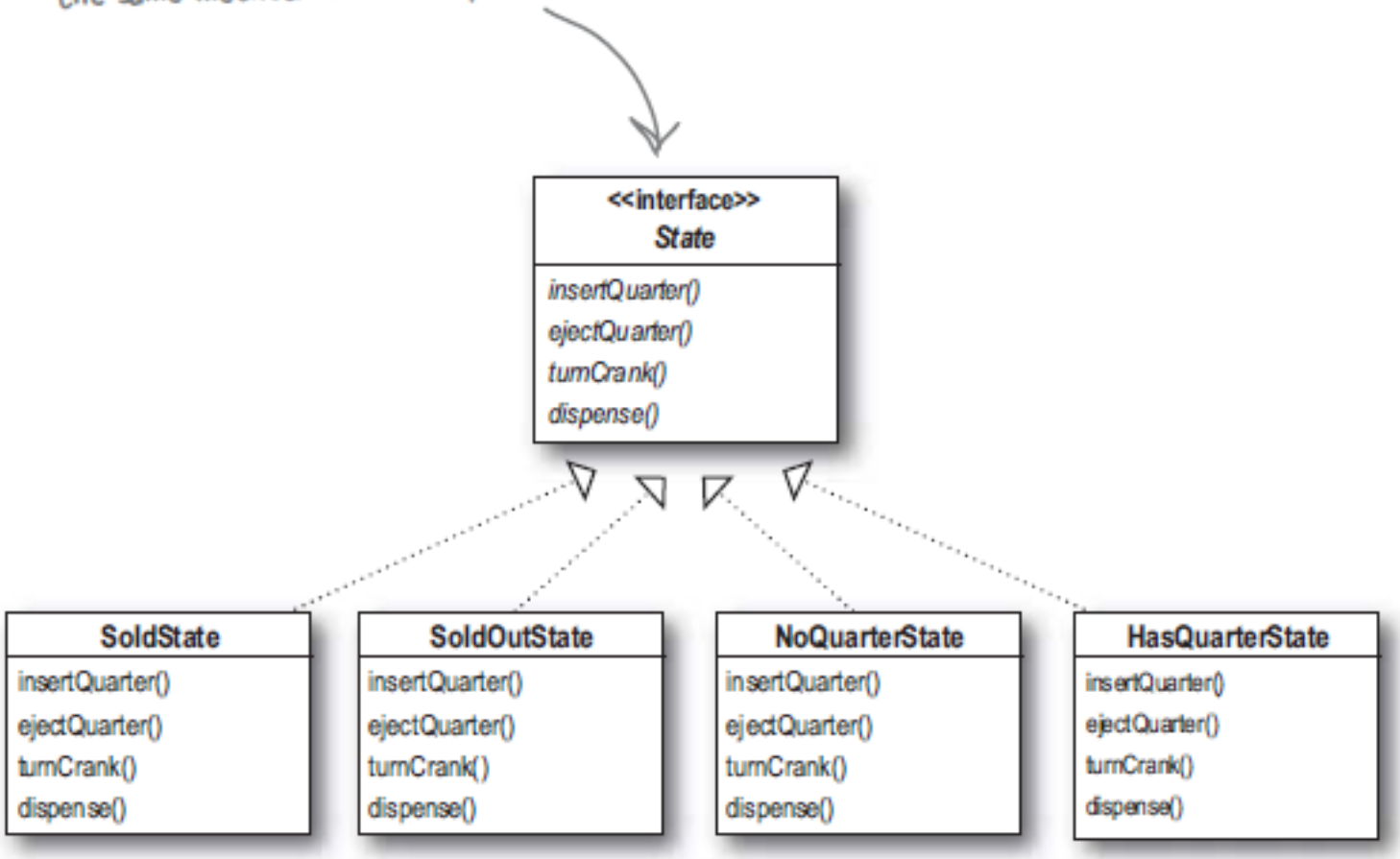
The new design

- Encapsulate what varies
- Put each behavior in its own class

1. **First, we're going to define a State interface that contains a method for every action in the Gumball Machine.**
2. **Then we're going to implement a State class for every state of the machine. These classes will be responsible for the behavior of the machine when it is in the corresponding state.**
3. **Finally, we're going to get rid of all of our conditional code and instead delegate to the state class to do the work for us.**

New Class Diagram

Here's the interface for all states. The methods map directly to actions that could happen to the Gumball Machine (these are the same methods as in the previous code).



No Quarter State

```
public void insertQuarter() {
                        ("You inserted a quarter");
    gumballMachine.setState(gumballMachine.getHasQuarterState());
}
```

```
}
```

GumballMachine class...

```
public class GumballMachine {

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

    State state = soldOutState;
    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;
        }
    }

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

Here are all the States again...

...and the State instance variable.

The count instance variable holds the count of gumballs – initially the machine is empty.

Our constructor takes the initial number of gumballs and stores it in an instance variable.

It also creates the State instances, one of each.

If there are more than 0 gumballs we set the state to the NoQuarterState.

Now for the actions. These are VERY EASY to implement now. We just delegate to the current state.

GumballMachine class...

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

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

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

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

    // More methods here including getters for each State...
}
```

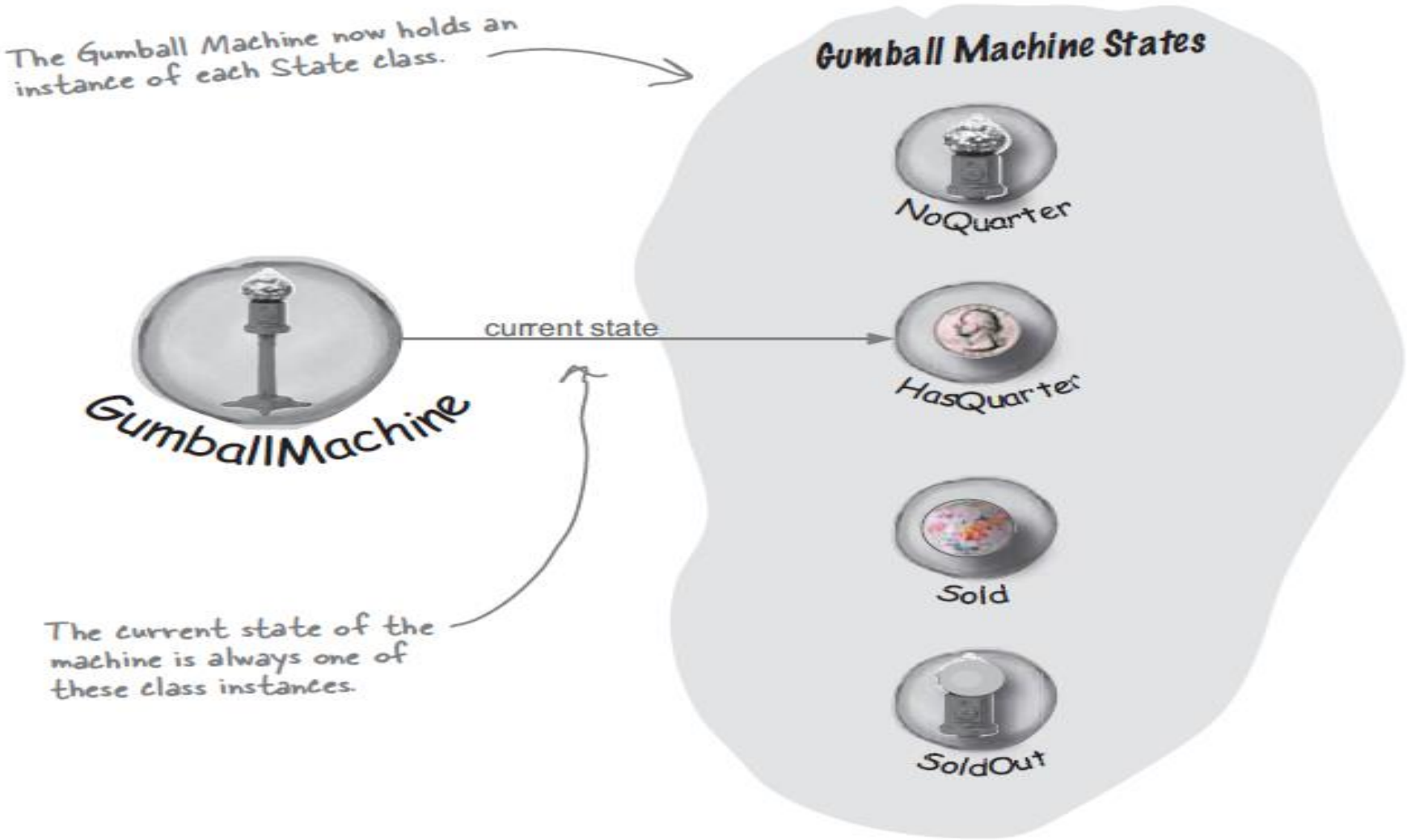
Note that we don't need an action method for dispense() in GumballMachine because it's just an internal action; a user can't ask the machine to dispense directly. But we do call dispense() on the State object from the turnCrank() method.

This method allows other objects (like our State objects) to transition the machine to a different state.

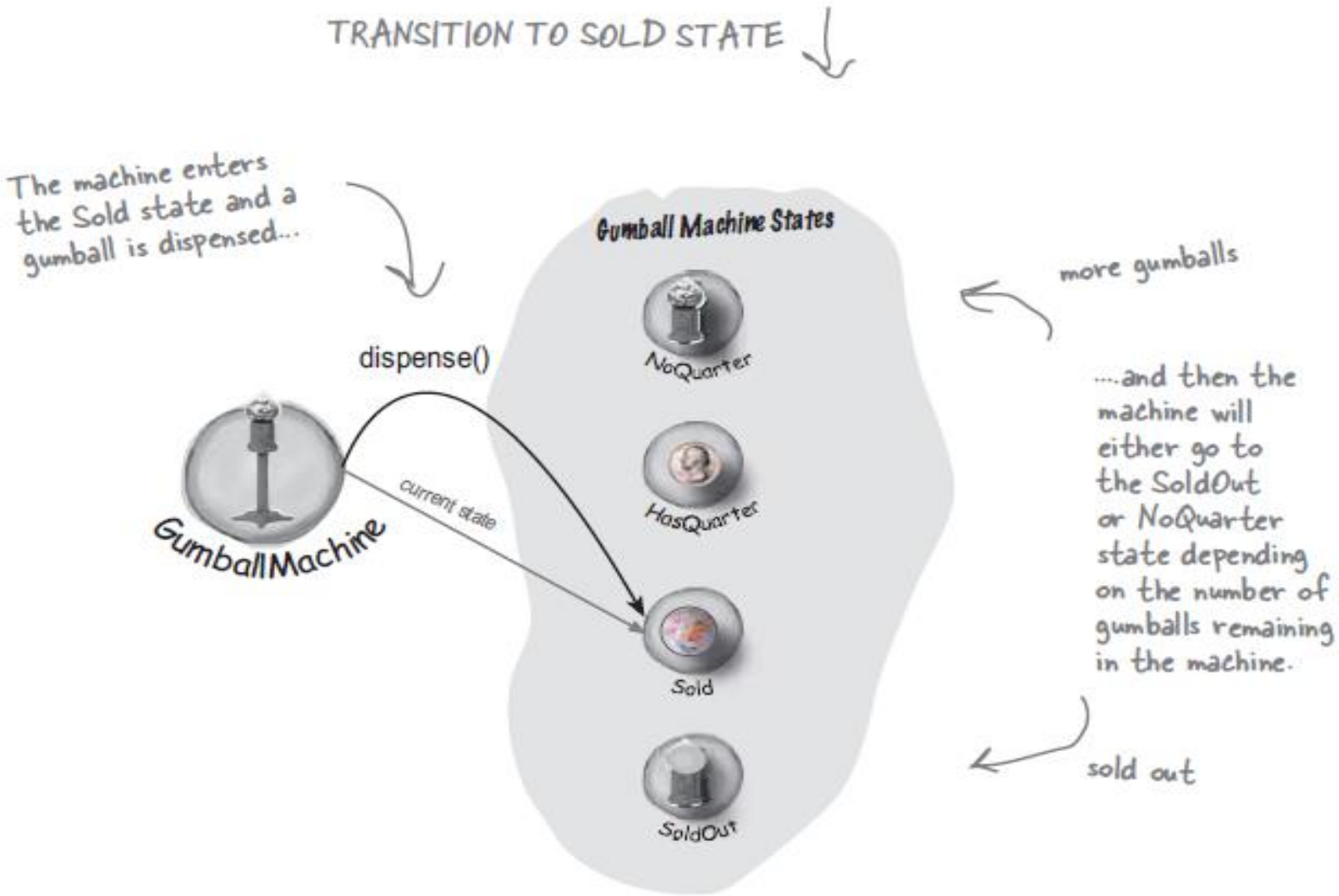
The machine supports a releaseBall() helper method that releases the ball and decrements the count instance variable.

This includes methods like getNoQuarterState() for getting each state object, and getCount() for getting the gumball count

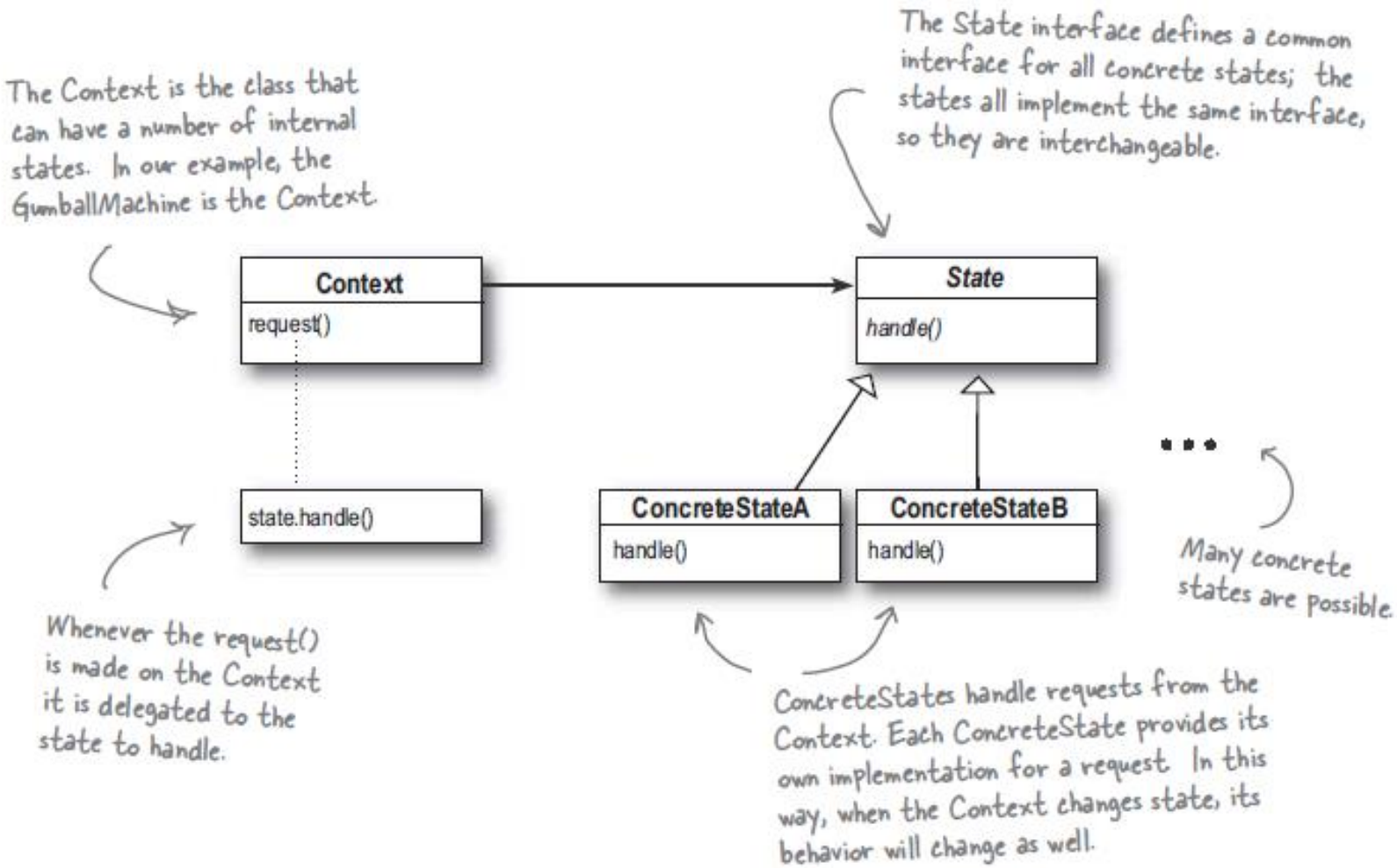
Let's take a look at what we've done so far...



Gumball Machine States



Review UML


The Context is the class that can have a number of internal states. In our example, the GumballMachine is the Context.
The State interface defines a common interface for all concrete states; the states all implement the same interface, so they are interchangeable.
Context
request()
State
handle()
state.handle()
ConcreteStateA
handle()
ConcreteStateB
handle()
...
Many concrete states are possible.
Whenever the request() is made on the Context it is delegated to the state to handle.
ConcreteStates handle requests from the Context. Each ConcreteState provides its own implementation for a request. In this way, when the Context changes state, its behavior will change as well.

State VS Strategy

Strategy pattern: Define a family of algorithms, encapsulate each one and make them interchangeable.

State pattern: Allow an object to alert its behavior when its internal state changes. The object will appear to change its class.

we encapsulate each algorithm into a class in strategy pattern, but we encapsulate each state into a class in state pattern

State VS Strategy

In general, think of the Strategy Pattern as a flexible alternative to subclassing; if you use inheritance to define the behavior of a class, then you're stuck with that behavior even if you need to change it. With Strategy you can change the behavior by composing with a different object.

Think of the State Pattern as an alternative to putting lots of conditionals in your context; by encapsulating the behaviors within state objects, you can simply change the state object in context to change its behavior.

Last Word

The State Pattern allows an object to have many different behaviors that are based on its internal state.

Unlike a procedural state machine, the State Pattern represents state as a full-blown class.

The Context gets its behavior by delegating to the current state object it is composed with.

By encapsulating each state into a class, we localize any changes that will need to be made.

The State and Strategy Patterns have the same class diagram, but they differ in intent.

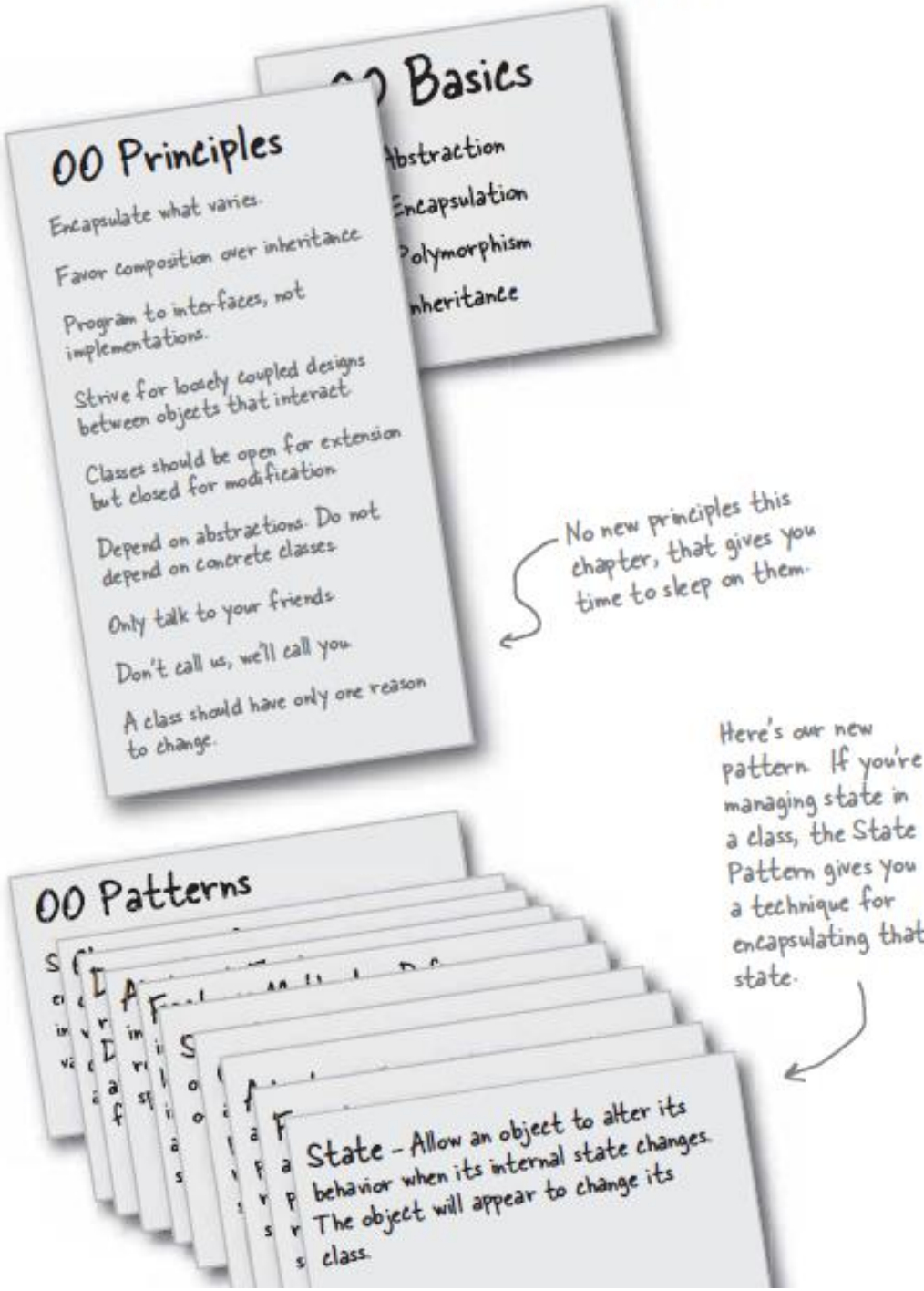
Strategy Pattern typically configures Context classes with a behavior or algorithm.

State Pattern allows a Context to change its behavior as the state of the Context changes.

State transitions can be controlled by the State classes or by the Context classes.

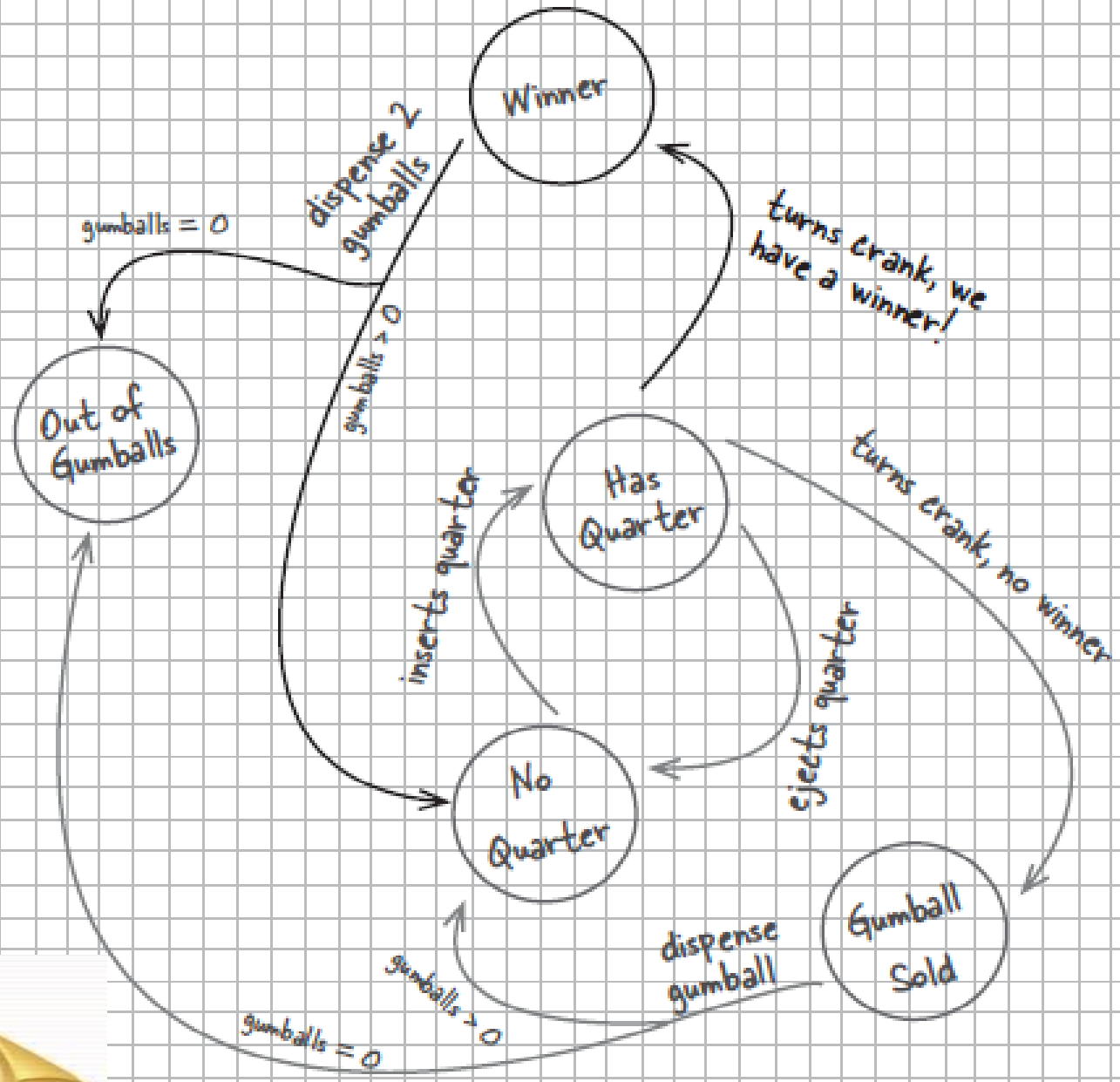
Using the State Pattern will typically result in a greater number of classes in your design.

State classes may be shared among Context instances.



Mighty Gumball, Inc.

Where the Gumball Machine is Never Half Empty

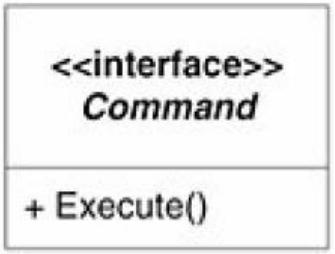


the Command Pattern

- With Command Pattern We are going to encapsulate method invocation.
- The Command pattern creates distance between the client that requests an operation and the object that can perform it. This pattern is particularly versatile. It can support:
  - Sending requests to different receivers
  - Queuing, logging, and rejecting requests
  - Composing higher-level transactions from primitive operations
  - Redo and Undo functionality

The Command Pattern encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.

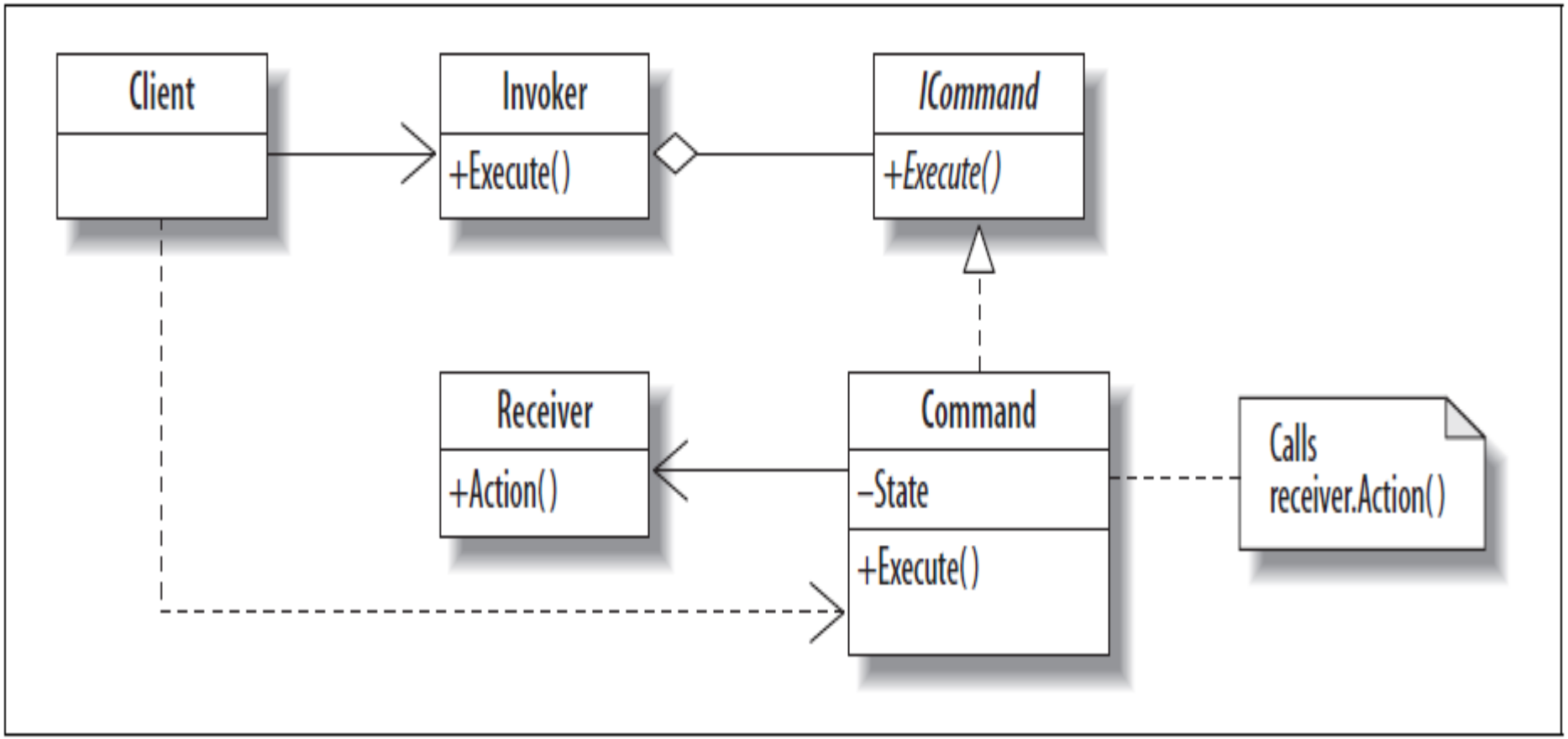
Command Design Pattern




```
public interface Command
{
  void Execute();
}
```

Class Diagram

- the Client issues a call to Execute in the Invoker, and the request goes to the Command and then to Action in the Receiver



Players

- Client

Creates and executes Commands

- ICommand

An interface that specifies the Execute operation

- Invoker

Asks the Command to carry out the Action

- Command

A class that implements the Execute operation by invoking operations on the Receiver

- Receiver

Any class that can perform the required Action

- Action

The operation that needs to be performed

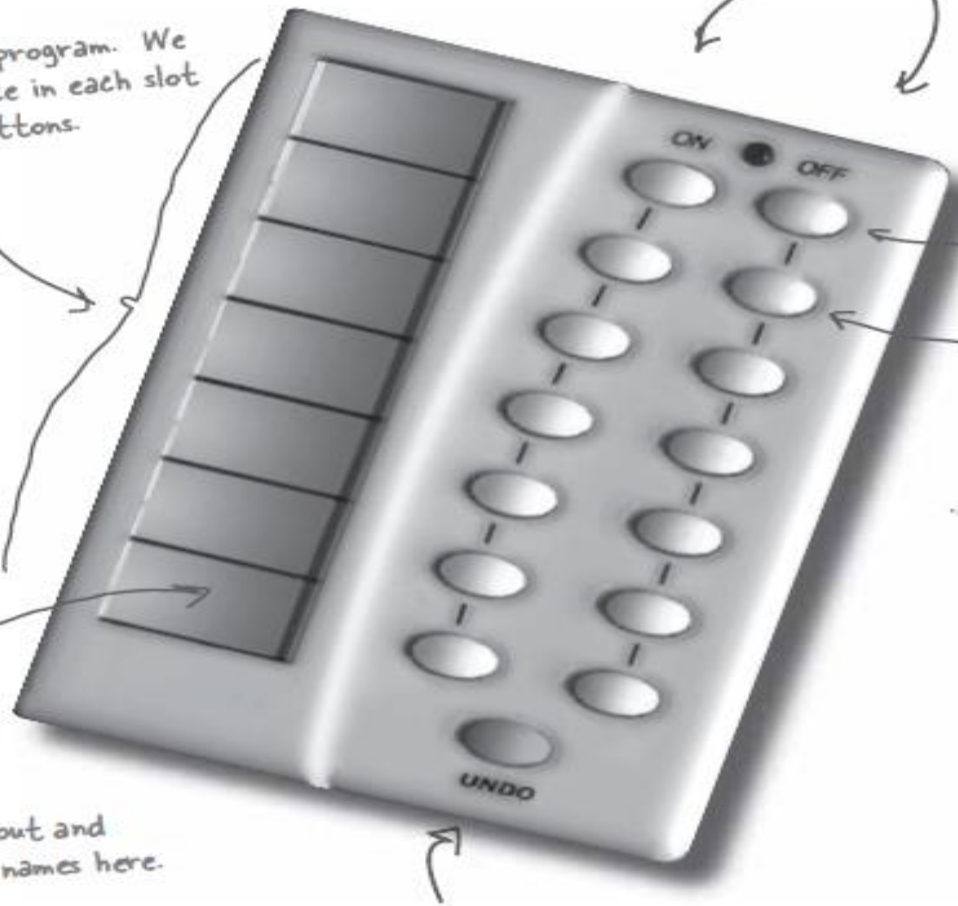
Simple Example

```
abstract class Command
{ protected Receiver receiver;
  public Command(Receiver receiver)
   { this.receiver = receiver}
  public abstract void Execute();
}

class ConcreteCommand : Command
 {
  public ConcreteCommand(Receiver
receiver) :base(receiver) {}
  public override void Execute() {
receiver.Action(); }
 }
```

```
class Receiver
{
 public void Action()
  { Console.WriteLine("Called
Receiver.Action()");        }
 }

class Invoker
{ private Command _command;
  public void
SetCommand(Command command)
  { this._command = command; }
  public void ExecuteCommand()
  { _command.Execute(); }
 }
```

```
invoker.SetCommand(command);

invoker.ExecuteCommand();
```

Free hardware! Let's check out the Remote Control...



There are "on" and "off" buttons for each of the seven slots.

We've got seven slots to program. We can put a different device in each slot and control it via the buttons.

These two buttons are used to control the household device stored in slot one...

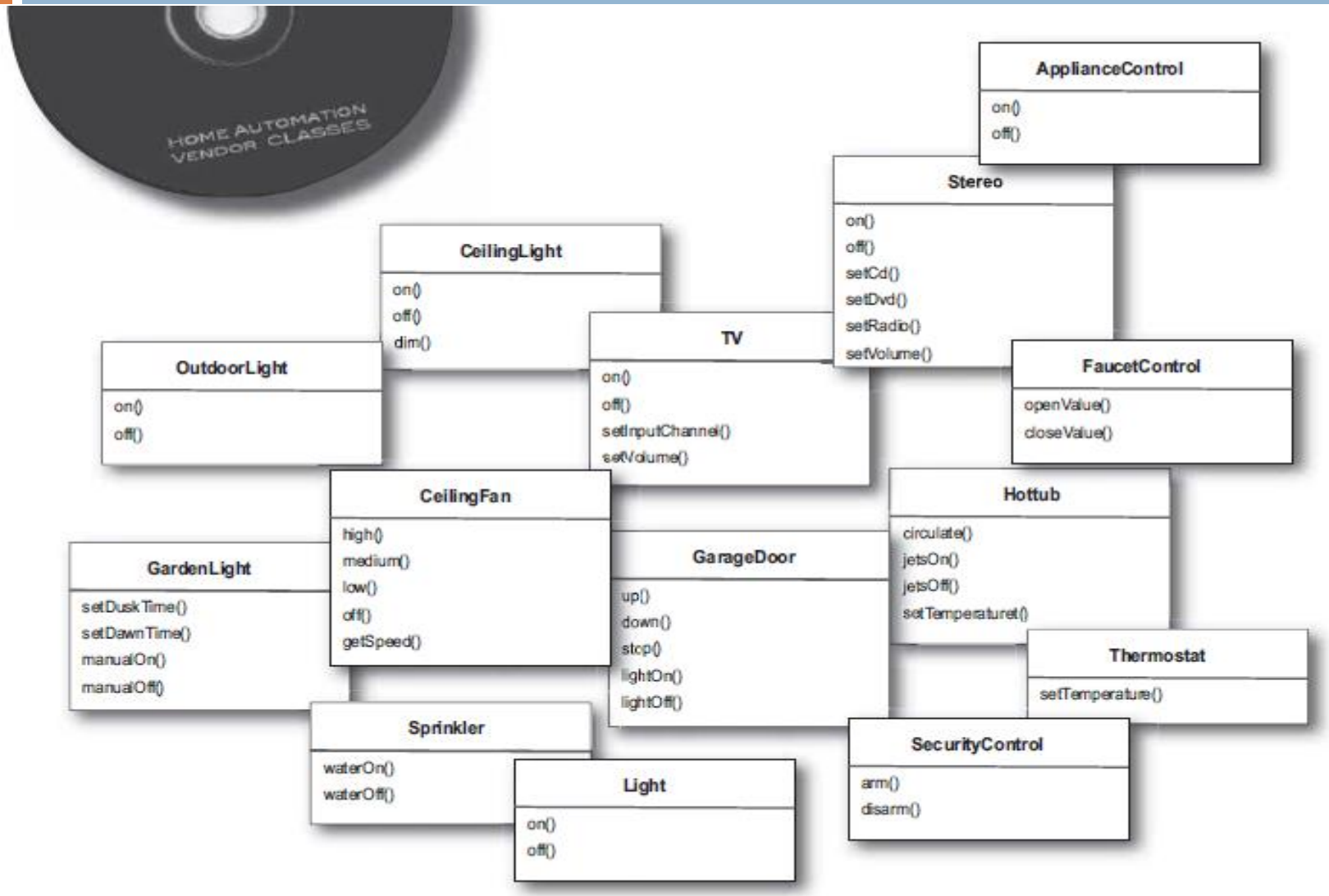
... and these two control the household device stored in slot two...

... and so on.

Get your Sharpie out and write your device names here.

Here's the global "undo" button that undoes the last button pressed.

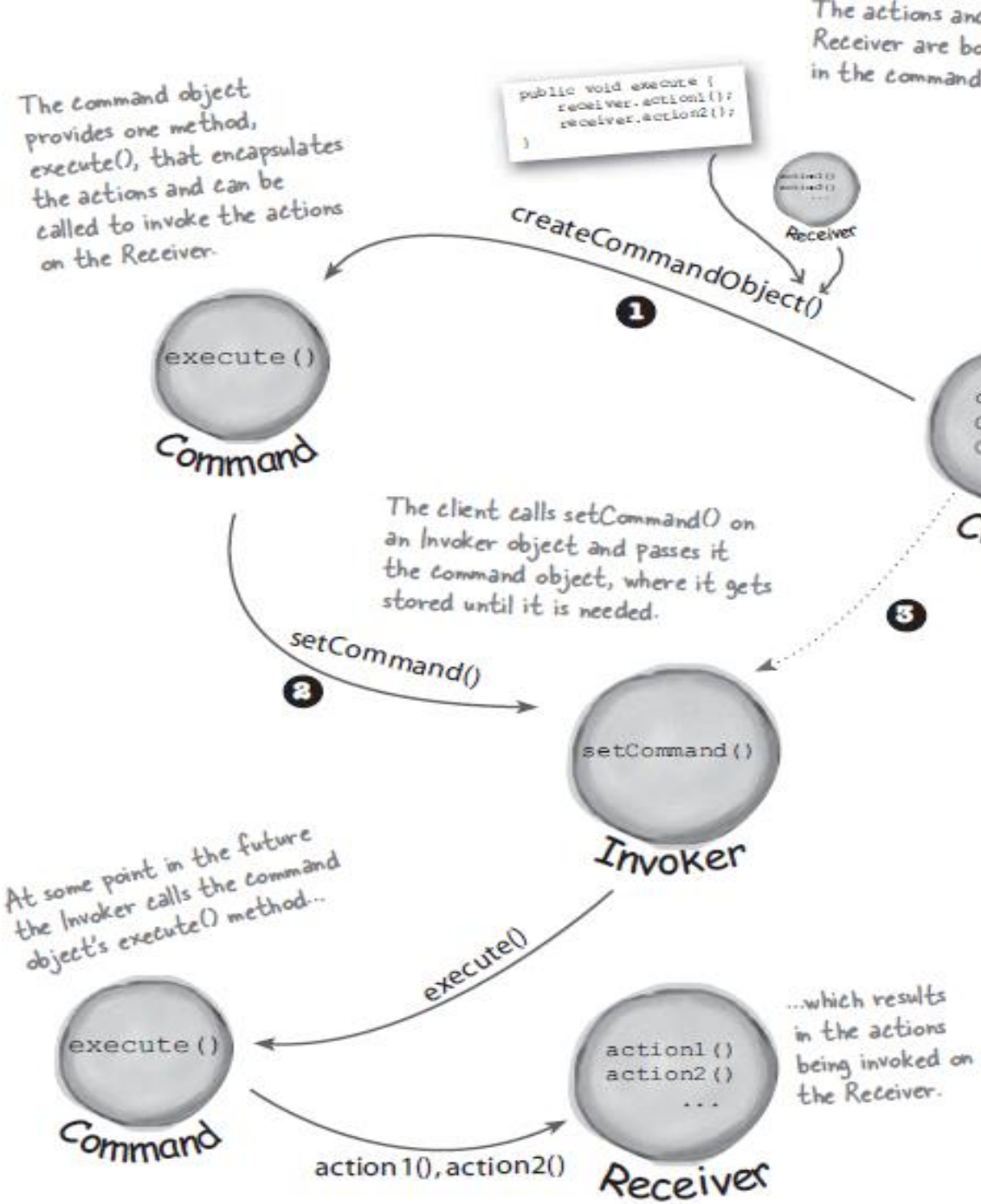
Target Classes



Words of Wisdom

The Command Pattern allows you to decouple the requester of an action from the object that actually performs the action. So, here the requester would be the remote control and the object that performs the action would be an instance of one of your vendor classes.

command object encapsulates a request to do something (like turn on a light) on a specific object (say, the living room light object). So, if we store a command object for each button, when the button is pressed we ask the command object to do some work. The remote doesn't have any idea what the work is, it just has a command object that knows how to talk to the right object to get the work done. So, you see, the remote is decoupled from the light object!



Loading the Invoker

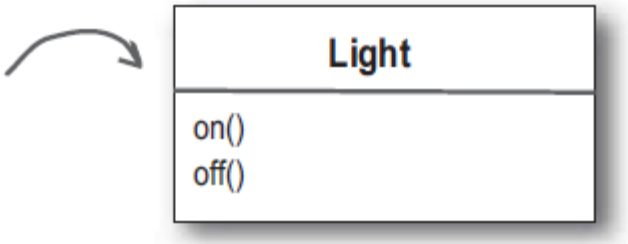
1. The client creates a command object.
2. The client does a setCommand() to store the command object in the invoker.
3. **Later**... the client asks the invoker to execute the command. Note: as you'll see later in the chapter, once the command is loaded into the invoker, it may be used and discarded, or it may remain and be used many times.

Implementing the Command interface

```
public interface Command {
    public void execute();
}
```

Simple. All we need is one method called execute().

Implementing a Command to turn a light on



This is a command, so we need to implement the Command interface.

```
public class LightOnCommand      :      Command {
    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.on();
    }
}
```

The constructor is passed the specific light that this command is going to control – say the living room light – and stashes it in the light instance variable. When execute gets called, this is the light object that is going to be the Receiver of the request.

The execute method calls the on() method on the receiving object, which is the light we are controlling.

Using the command object

```
public class SimpleRemoteControl {
    Command slot;

    public SimpleRemoteControl() {}

    public void setCommand(Command command) {
        slot = command;
    }

    public void buttonWasPressed() {
        slot.execute();
    }
}
```

We have one slot to hold our command, which will control one device.

We have a method for setting the command the slot is going to control. This could be called multiple times if the client of this code wanted to change the behavior of the remote button.

This method is called when the button is pressed. All we do is take the current command bound to the execute() method.

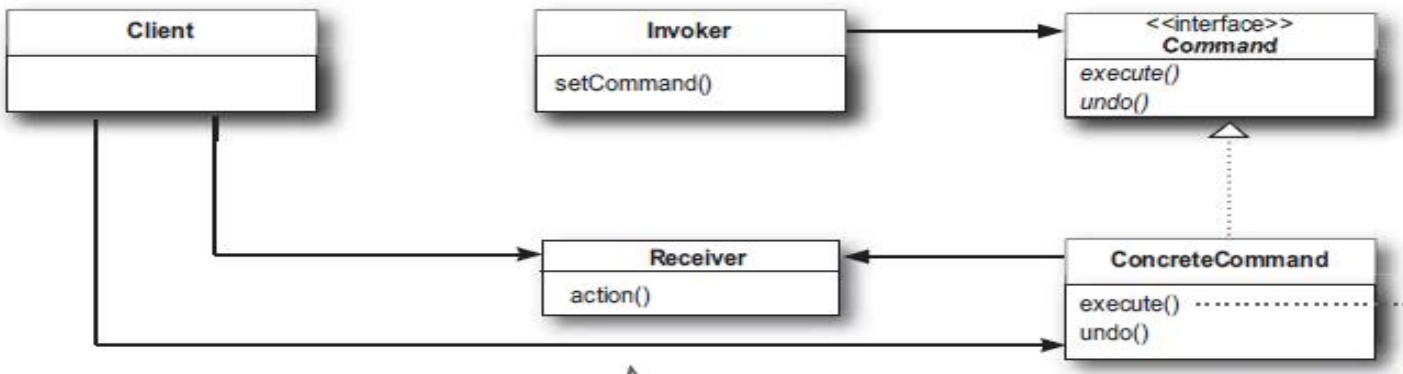
```
SimpleRemoteControl remote = new SimpleRemoteControl();
Light light = new Light();
LightOnCommand lightOn = new LightOnCommand(light);

remote.setCommand(lightOn);
```

The Command Pattern defined

The Client is responsible for creating a ConcreteCommand and setting its Receiver.

The Invoker holds a command and at some point asks the command to carry out a request by calling its execute() method.

Command declares an interface for all commands. As you already know, a command is invoked through its execute() method, which asks a receiver to preform an action. You'll also notice this interface has an undo() method, which we'll cover a bit later in the chapter.



The execute method invokes the action(s) on the receiver needed to fulfill the request.

```
public void execute() {
    receiver.action()
}
```

The Receiver knows how to perform the work needed to carry out the request. Any class can act as a Receiver.

The ConcreteCommand defines a binding between an action and a Receiver. The Invoker makes a request by calling execute() and the ConcreteCommand carries it out by calling one or more actions on the Receiver.

Implementing the Remote Control

```
public class RemoteControl {
    Command[] onCommands;
    Command[] offCommands;

    public RemoteControl() {
        onCommands = new Command[7];
        offCommands = new Command[7];

        Command noCommand = new NoCommand();
        for (int i = 0; i < 7; i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
    }

    public void setCommand(int slot, Command onCommand, Command offCommand) {
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }

    public void onButtonWasPushed(int slot) {
        onCommands[slot].execute();
    }

    public void offButtonWasPushed(int slot) {
        offCommands[slot].execute();
    }
```

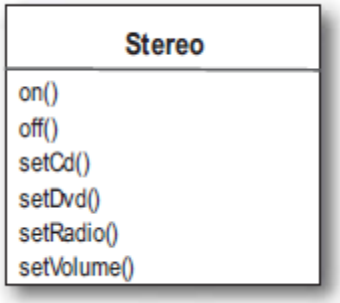
This time around the remote is going to handle seven On and Off commands, which we'll hold in corresponding arrays.

In the constructor all we need to do is instantiate and initialize the on and off arrays.

The setCommand() method takes a slot position and an On and Off command to be stored in that slot. It puts these commands in the on and off arrays for later use.

When an On or Off button is pressed, the hardware takes care of calling the corresponding methods onButtonWasPushed() or offButtonWasPushed().

Another Command




```
public class StereoOnWithCDCommand    :    Command {
    Stereo stereo;

    public StereoOnWithCDCommand(Stereo stereo) {
        this.stereo = stereo;
    }

    public void execute() {
        stereo.on();
        stereo.setCD();
        stereo.setVolume(11);
    }
}
```

Just like the LightOnCommand, we get passed the instance of the stereo we are going to be controlling and we store it in a local instance variable.

To carry out this request, we need to call three methods on the stereo: first, turn it on, then set it to play the CD, and finally set the volume to 11. Why 11? Well, it's better than 10, right?

The Undo

```
public interface Command {
    public void execute();
    public void undo();
}
```

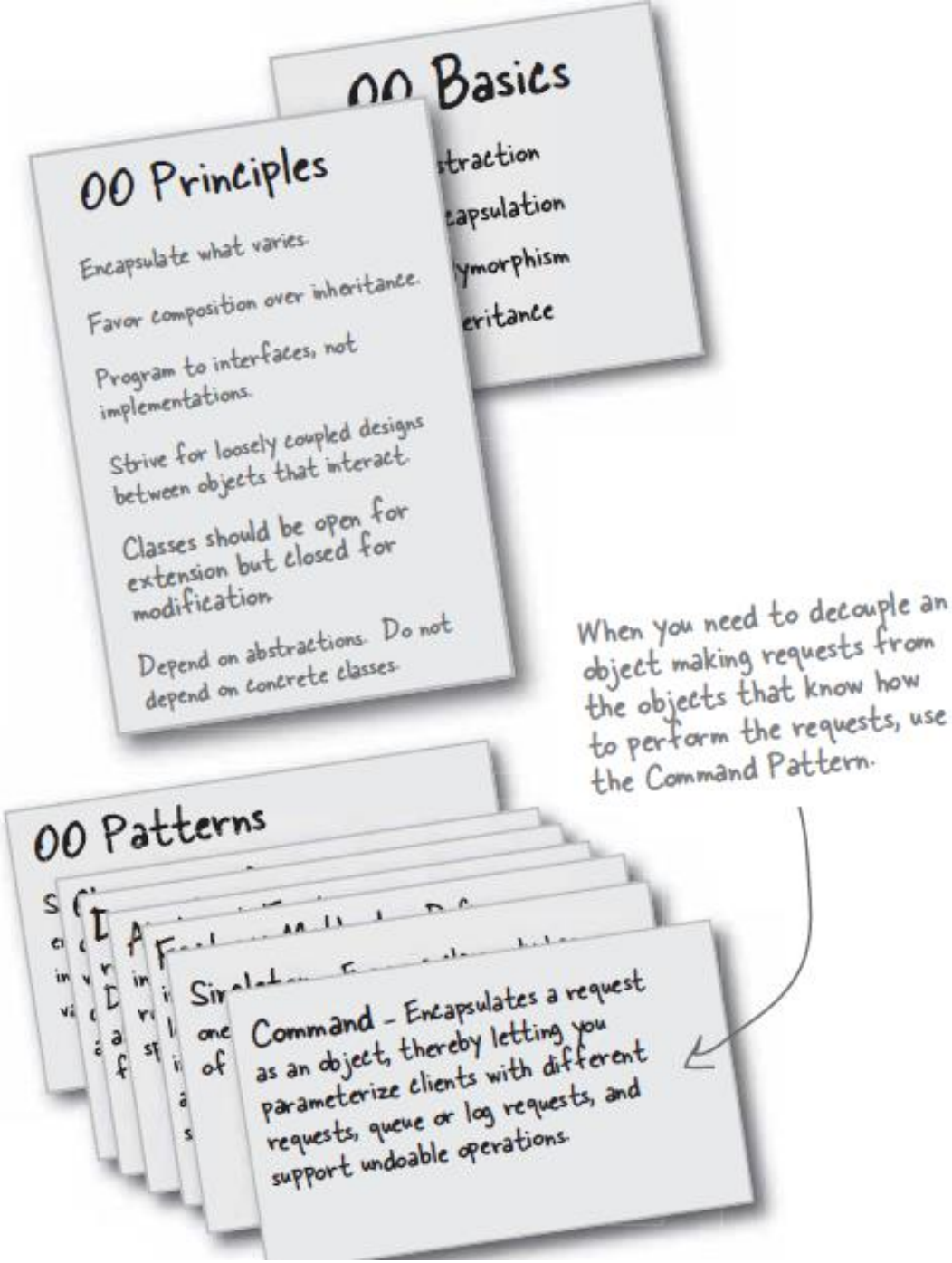
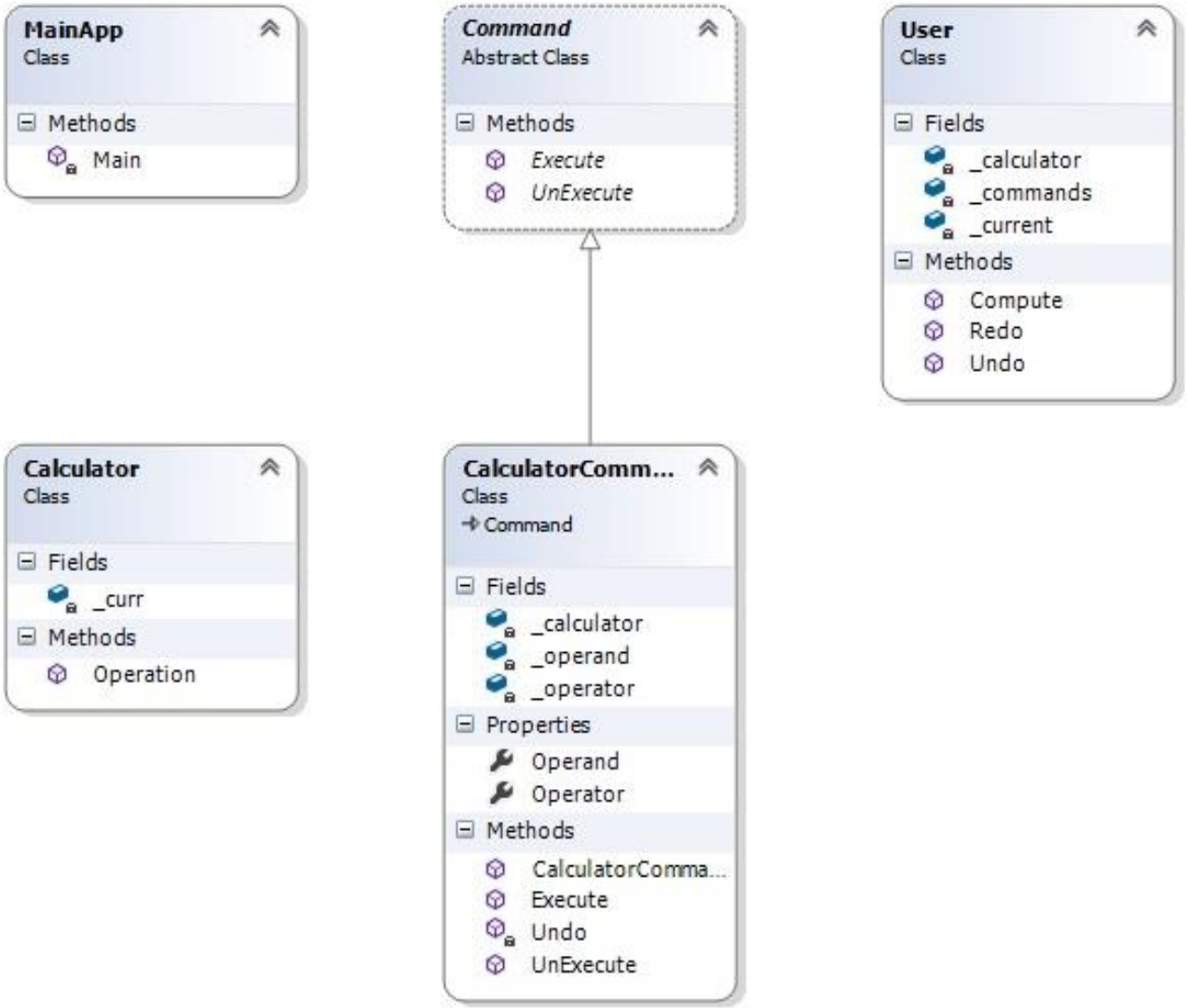
Here's the new undo() method.

```
public class LightOnCommand : Command {
    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.on();
    }

    public void undo() {
        light.off();
    }
}
```

execute() turns the light on, so undo() simply turns the light back off.

Final Word

- The Command Pattern decouples an object, making a request from the one that knows how to perform it.
- A Command object is at the center of this decoupling and encapsulates a receiver with an action (or set of actions) .
- An invoker makes a request of a Command object by calling its execute() method, which invokes those actions on the receiver.
- Invokers can be parameterized with Commands, even dynamically at runtime.
- Commands may support undo by implementing an undo method that restores the object to its previous state before the excute() method was last called.
- Macro Commands are a simple extension of Command that allow multiple commands to be invoked. Likewise, Macro Commands can easily support undo().
- In practice, it is not uncommon for "smart" Command objects to implement the request themselves rather than delegating to a receiver.
- Commands may also be used to implement logging and transactional systems.



GUI Calculator with Do & ReDo