Software Components

Design Patterns

Goals

- Introduce the concept of design patterns
  - Explain how it arose from the field of architecture and anthropology
- Discuss why design patterns are important and what advantages they provide
- Present an example of one design pattern

- In 1995, a book was published by the “Gang of Four” called Design Patterns
  - It applied the concept of patterns (discussed next) to software design and described 23 of them
    - The authors did not invent these patterns
      - Instead, they included patterns they found in at least 3 “real” software systems.
- Since that time lots of DP books have been published
  - and more patterns have been cataloged
  - although many pattern authors abandoned the criteria of having to find the pattern in 3 shipping systems
- Unfortunately, many people feel like they should become experts in OO A&D before they learn about patterns
  - our book takes a different stance: learning about design patterns will help you become an expert in OO A&D

- Design Patterns have their intellectual roots in the discipline of cultural
  - Within a culture, individuals will agree on what is considered good design
    - "Cultures make judgements on good design that transcend individual beliefs"
  - Patterns (structures and relationships that appear over and over again in many different well designed objects) provide an objective basis for judging design

- Design patterns in software design traces its intellectual roots to work performed in the 1970s by an architect named Christopher Alexander
  - His 1979 book called “The Timeless Way of Building” that asks the question “Is quality objective?”
    - in particular, “What makes us know when an architectural design is good? Is there an objective basis for such a judgement?”
  - His answer was “yes” that it was possible to objectively define “high quality” or “beautiful” buildings
- He studied the problem of identifying what makes a good architectural design by observing all sorts of built structures
  - buildings, towns, streets, homes, community centers, etc.
- When he found an example of a high quality design, he would compare that object to other objects of high quality and look for commonalties
  - especially if both objects were used to solve the same type of problem

- By studying high quality structures that solve similar problems, he could discover similarities between the designs and these similarities where what he called patterns
  - “Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”
  - The pattern provides an approach that can be used to achieve a high quality solution to its problem

- Alexander identified four elements to describe a pattern
  - The name of the pattern
  - The purpose of the pattern: what problem it solves
  - How to solve the problem
  - The constraints we have to consider in our solution
- He also felt that multiple patterns applied together can help to solve complex architectural problems

- Alexander identified four elements to describe a pattern
  - The name of the pattern
  - The purpose of the pattern: what problem it solves
  - How to solve the problem
  - The constraints we have to consider in our solution
- He also felt that multiple patterns applied together can help to solve complex architectural problems

- Work on design patterns got started when people asked
  - Are there problems in software that occur all the time that can be solved in somewhat the same manner?
  - Was it possible to design software in terms of patterns?
- Many people felt the answer to these questions was “yes” and this initial work influenced the creation of the Design Patterns book by the Gang of Four
  - It catalogued 23 patterns: successful solutions to common problems that occur in software design
- Design patterns, then, assert that the quality of software systems can be measured objectively
  - What is present in a good quality design (X’s) that is not present in a poor quality design?
  - What is present in a poor quality design (Y’s) that is not present in a good quality design?
- We would then want to maximize the X’s while minimizing the Y’s in our own designs

- “Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context”

[Gamma, Helm, Johnson, Vlissides 1995]

- **Name**
- **Intent**: The purpose of the pattern
- **Problem**: What problem does it solve?
- **Solution**: The approach to take to solve the problem
- **Participants**: The entities involved in the pattern
- **Consequences**: The effect the pattern has on your system
- **Structure**: Class Diagram
- **Implementation**: Example ways to implement the pattern
- **sample code**
- **related patterns**

- Patterns let us
  - reuse solutions that have worked in the past; why waste time reinventing the wheel?
  - have a shared vocabulary around software design
    - they allow you to tell a fellow software engineer “I used a Strategy pattern here to allow the algorithm used to compute this calculation to be customizable”
      - You don’t have to waste time explaining what you mean since you both know the Strategy pattern
- Design patterns provide you **not with code reuse** but with **experience reuse**
  - Knowing concepts such as abstraction, inheritance and polymorphism will NOT make you a good designer, unless you use those concepts to create flexible designs that are maintainable and that can cope with change
- Design patterns can show you how to apply those concepts to achieve those goals

- Design Patterns give you a higher-level perspective on
  - the problems that come up in OO A&D work
  - the process of design itself
  - the use of object orientation to solve problems
- You'll be able to think more abstractly and not get bogged down in implementation details too early in the process

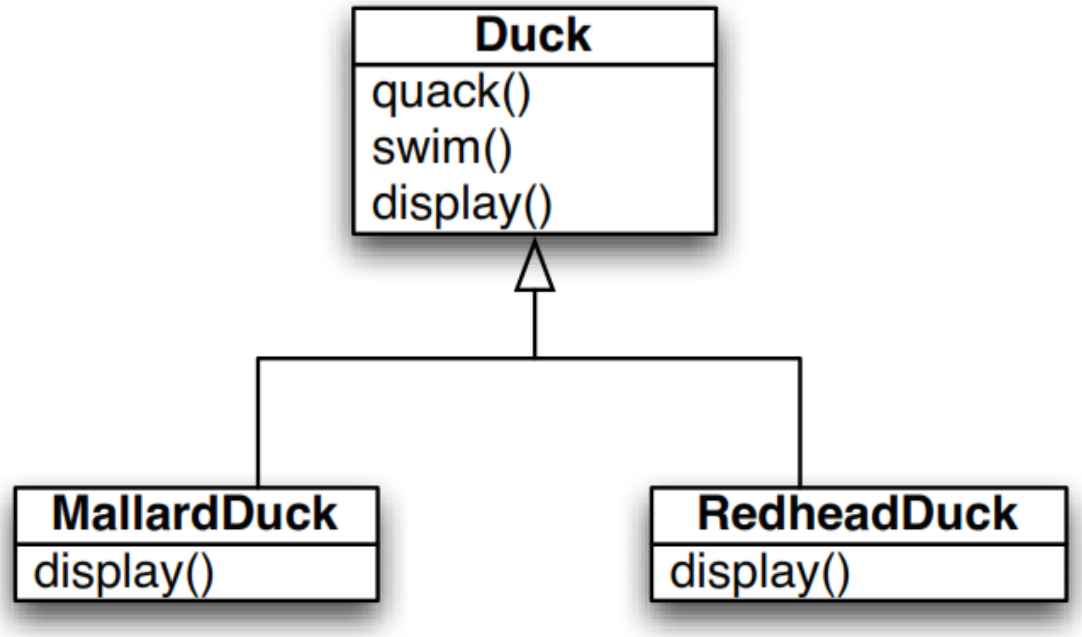
- Improved Motivation of Individual Learning in Team Environments
  - Junior developers see that the design patterns discussed by more senior developers are valuable and are motivated to learn them
- Improved maintainability
  - Many design patterns make systems easy to extend, leading to increased maintainability
- Design patterns lead to a deeper understanding of core OO principles
- They reinforce useful design heuristics such as
  - code to an interface
  - favor delegation over inheritance
  - find what varies and encapsulate it
- Since they favor delegation, they help you avoid the creation of large inheritance hierarchies, reducing complexity

Purpose		
Creational	**Structural**	**Behavioral**
Abstract Factory Builder Factory Method Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Template Method Visitor

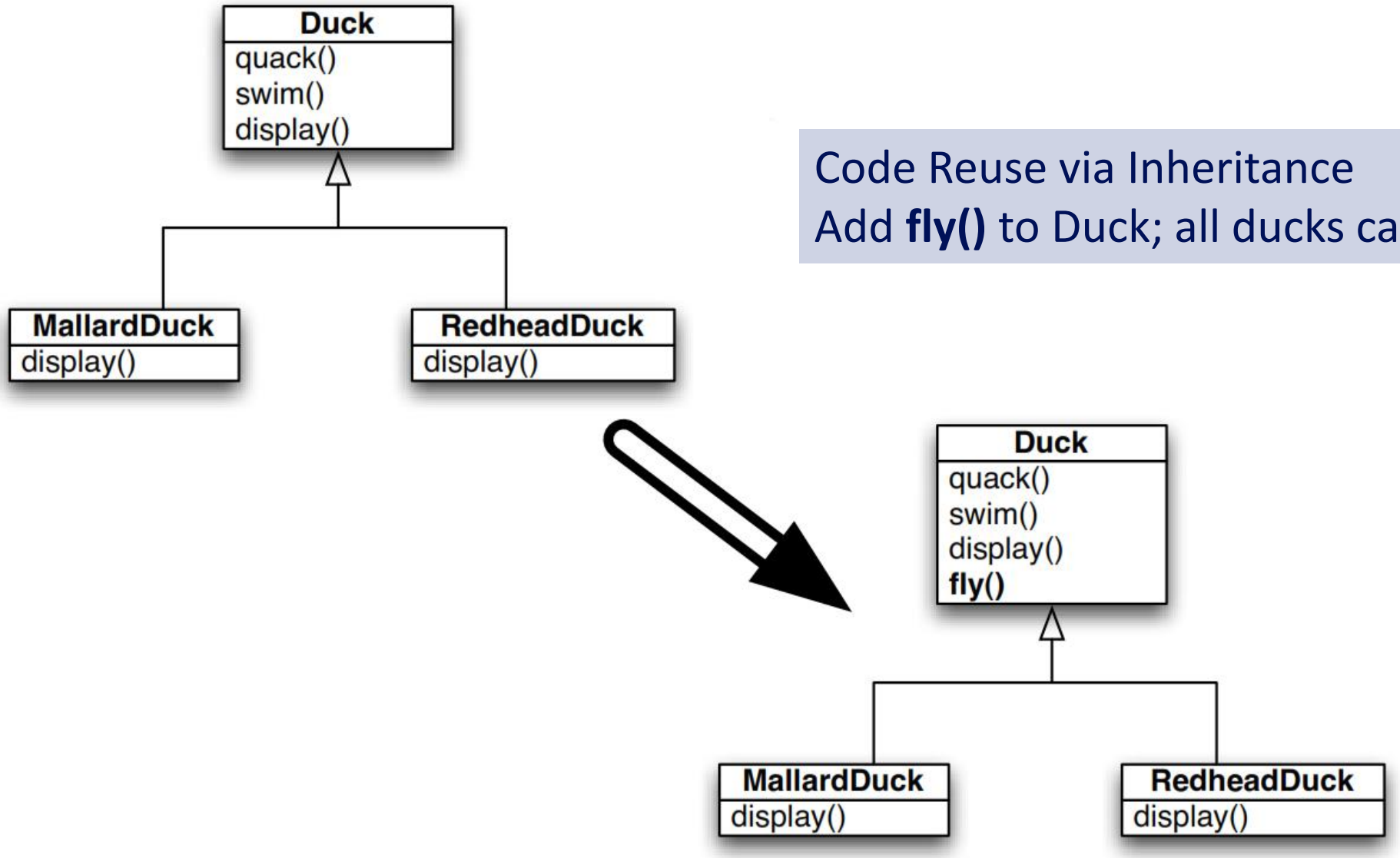
Purpose		
Creational	**Structural**	**Behavioral**
Abstract Factory Builder Factory Method Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Template Method Visitor

- Patterns We will be talking about in detail; you should read about the others in the book or online.

- SimUDuck: a “duck pond simulator” that can show a wide variety of duck species swimming and quacking
  - Initial State

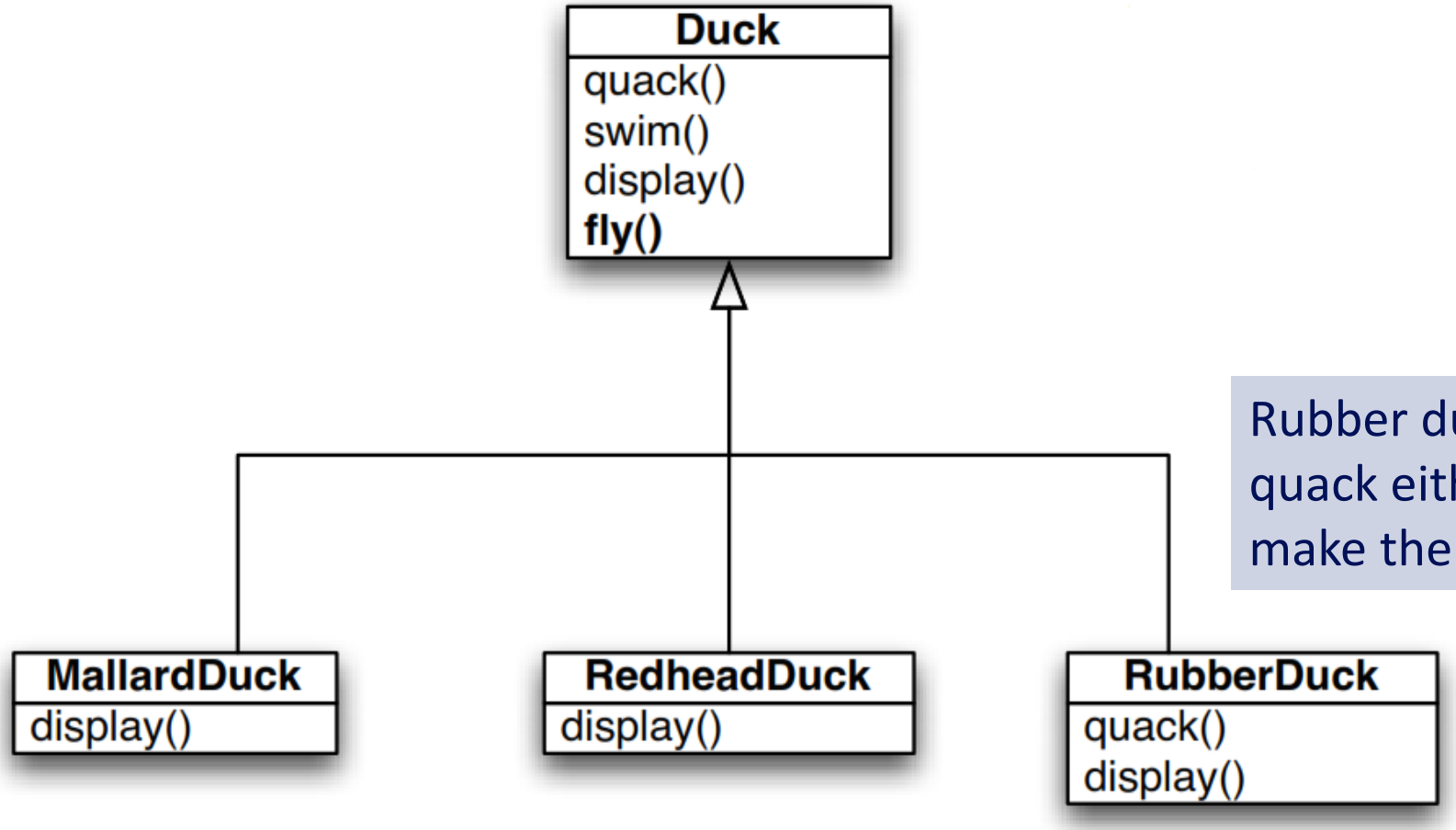


- But a request has arrived to allow ducks to also fly. (We need to stay ahead of the competition!)



Code Reuse via Inheritance
Add **fly()** to Duck; all ducks can now fly

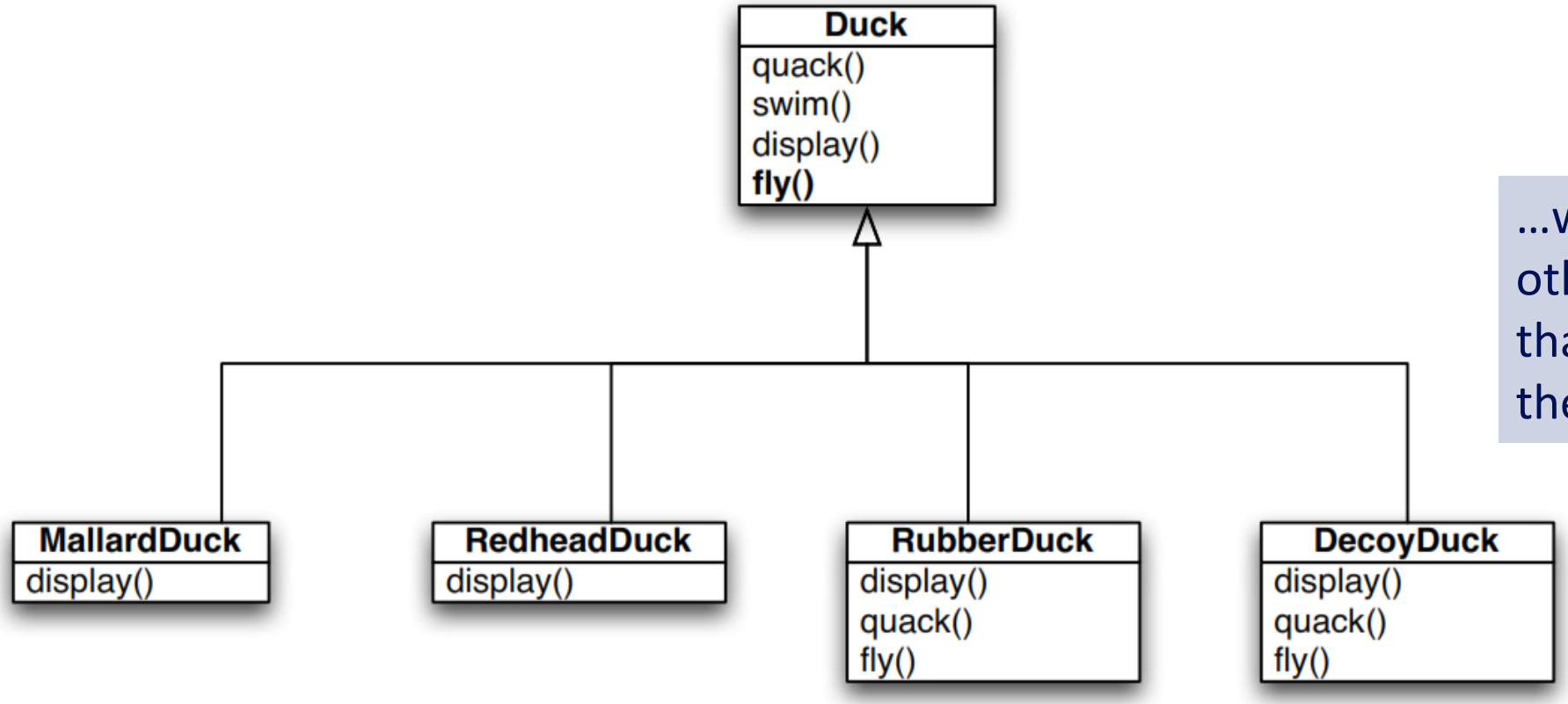
Whoops!



Rubber ducks do not fly! They don't quack either, so we override quack() to make them squeak

We could override fly() in RubberDuck to make it do nothing, but that's less than ideal, especially…

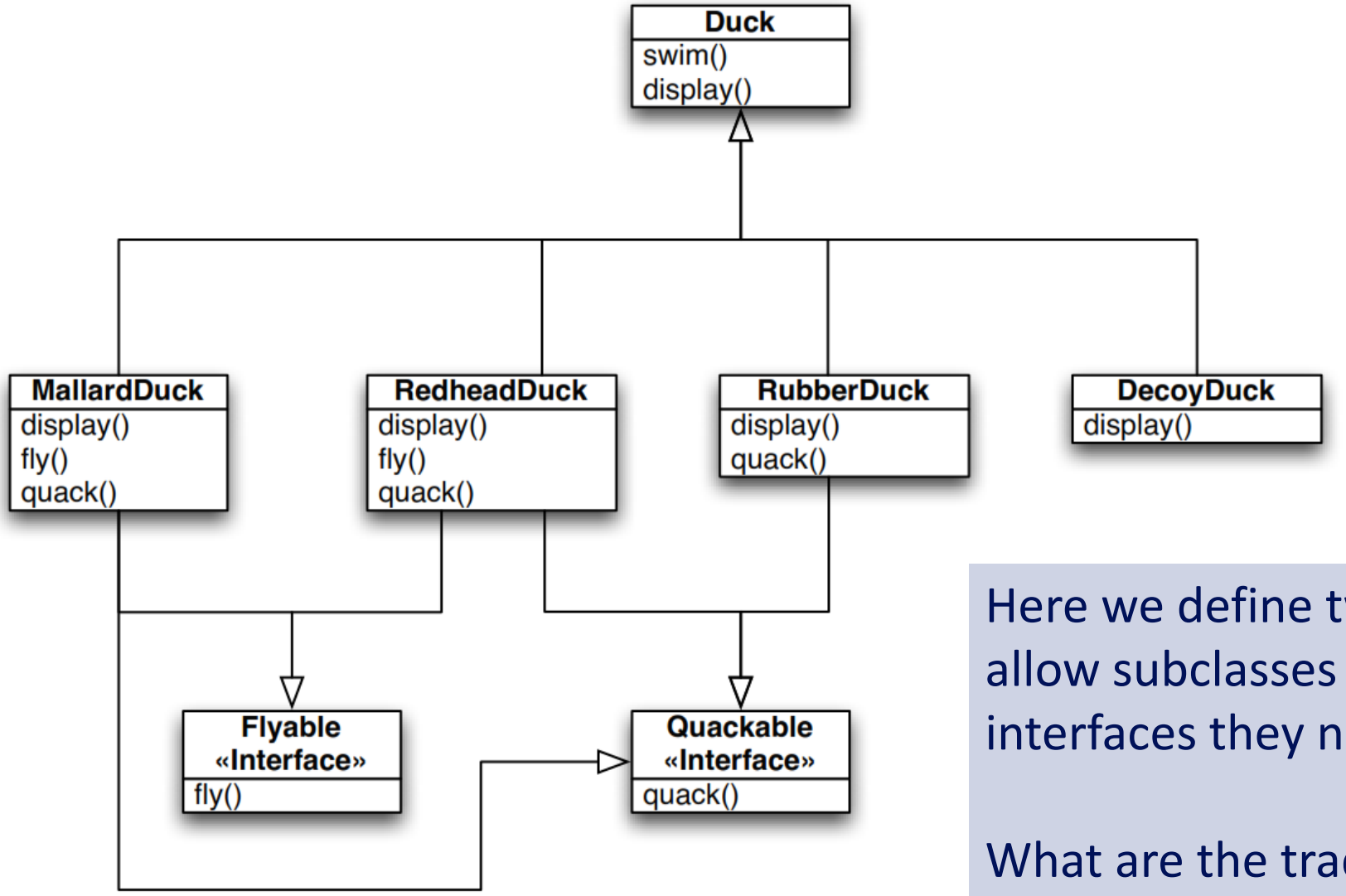
Double Whoops!



…when we might find other Duck subclasses that would have to do the same thing!

What was supposed to be a good instance of reuse via inheritance has turned into a maintenance headache!

What about an Interface?



Here we define two interfaces and allow subclasses to implement the interfaces they need.

What are the trade-offs?

- With inheritance, we get
  - Code reuse, only one fly() and quack() method vs. multiple (pro)
  - Common behavior in root class, not so common after all (con)
- With interfaces, we get
  - Specificity: only those subclasses that need a fly() method get it (pro)
  - No code re-use: since interfaces only define signatures (con)
- Use of abstract base class over an interface? Could do it, but only in languages that support multiple inheritance
  - In this approach, you implement Flyable and Quackable as abstract base classes and then have Duck subclasses use multiple inheritance

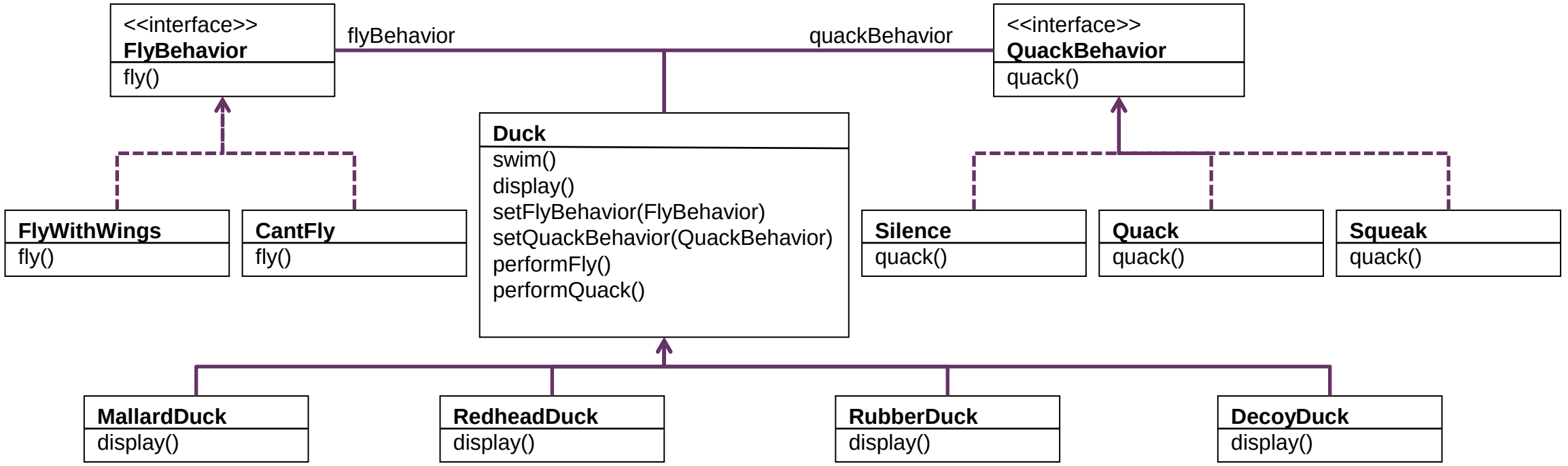
- Encapsulate What Varies
  - For this particular problem, the “what varies” is the behaviors between Duck subclasses
    - We need to pull out behaviors that vary across the subclasses and put them in their own classes (i.e., encapsulate them)
  - The result: fewer unintended consequences from code changes (such as when we added fly() to Duck) and more flexible code

- Take any behavior that varies across Duck subclasses and pull them out of Duck
  - Duck will no longer have fly() and quack() methods directly
  - Create two sets of classes, one that implements fly behaviors and one that implements quack behaviors
- Code to an Interface
  - We'll make use of the "code to an interface" principle and make sure that each member of the two sets implements a particular interface
    - For QuackBehavior, we'll have Quack, Squeak, Silence
    - For FlyBehavior, we'll have FlyWithWings, CantFly, FlyWhenThrown, …
- Additional Benefits
  - Other classes can gain access to these behaviors and we can add additional behaviors without impacting other classes

- We are overloading the word "interface" when we say "code to an interface"
  - We can implement "code to an interface" by defining a Java interface and then have various classes implement that interface
  - Or, we can "code to a supertype" and instead define an abstract base class which classes can access via inheritance
- When we say "code to an interface" it implies that the object that is using the interface will have a variable whose type is the supertype (whether it is an interface or an abstract base class) and thus
  - can point at any implementation of that supertype
  - and is shielded from their specific class names
    - A Duck will point to a fly behavior with a variable of type FlyBehavior NOT FlyWithWings; the code will be more loosely coupled as a result

- To take advantage of these new behaviors, we must modify Duck to delegate its flying and quacking behaviors to these other classes
  - rather than implementing this behavior internally
- We'll add two attributes that store the desired behavior and we'll rename fly() and quack() to performFly() and performQuack()
  - this last step is meant to address the issue of it not making sense for a DecoyDuck to have methods like fly() and quack() directly as part of its interface
    - Instead, it inherits these methods and plugs-in CantFly and Silence behaviors to make sure that it does the right things if those methods are invoked
- This is an instance of the principle "Favor delegation over inheritance"

New Class Diagram



- FlyBehavior and QuackBehavior define a set of behaviors that provide behavior to Duck.
- Duck delegates to each set of behaviors and can switch among them dynamically, if needed.
- While each subclass now has a performFly() and performQuack() method, at least the user interface is uniform and those methods can point to null behaviors when required.

FlyBehavior.java

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

QuackBehavior.java

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

FlyBehavior.java

```
public class FlyWithWings implements FlyBehavior {

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

}
```

QuackBehavior.java

```
public class Squeak implements QuackBehavior {

    public void quack() {
        System.out.println("Squeak");
    }

}
```

Duck.java

FlyBehavior.java

```
public abstract class Duck {

    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;

    public Duck() {}

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

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

    abstract void display();
```

FlyBehavior.java (continue)

```
    public void performFly() {
        flyBehavior.fly();
    }

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

    public void swim() {
        System.out.println("All ducks float,
                                even decoys!");
    }
}
```

Note: “code to interface”, delegation, encapsulation, and ability to change behaviors dynamically

RubberDuck.java

FlyBehavior.java

```
public class RubberDuck extends Duck {

    public RubberDuck() {
        flyBehavior = new CantFly();
        quackBehavior = new Squeak();
    }

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

DuckSimulator.java (I)

FlyBehavior.java

```
public class DuckSimulator {

    public static void main(String[] args) {
        List<Duck> ducks = new LinkedList<Duck>();

        Duck myDuck = new RubberDuck();

        ducks.add(new MallardDuck());
        ducks.add(new DecoyDuck());
        ducks.add(new RedheadDuck());
        ducks.add(myDuck);

        processDucks(ducks);

        // change my ducks' behavior dynamically to make a rubber
        // duck that can fly rocket powered and speak
        myDuck.setFlyBehavior(new FlyRocketPowered());
        myDuck.setQuackBehavior(new Speak());

        processDucks(ducks);
    }
}
```

Note: all variables are of type Duck, not the specific subtypes; “code to interface” in action

Note: here we see the power of delegation. We can change behaviors at run-time

DuckSimulator.java (II)

FlyBehavior.java (continue)

```
import java.util.*;

public class DuckSimulator {

    public static void processDucks(List<Duck> ducks) {
        for (Duck duck : ducks) {
            System.out.println("--------------------");
            System.out.println("Name: " + duck.getClass().getName());
            duck.display();
            duck.performQuack();
            duck.performFly();
            duck.swim();
        }

        System.out.println("Done processing ducks\n");
    }

}
```

Because of **abstraction** and **polymorphism**, processDucks() consists of nice, clean, robust, and extensible code!

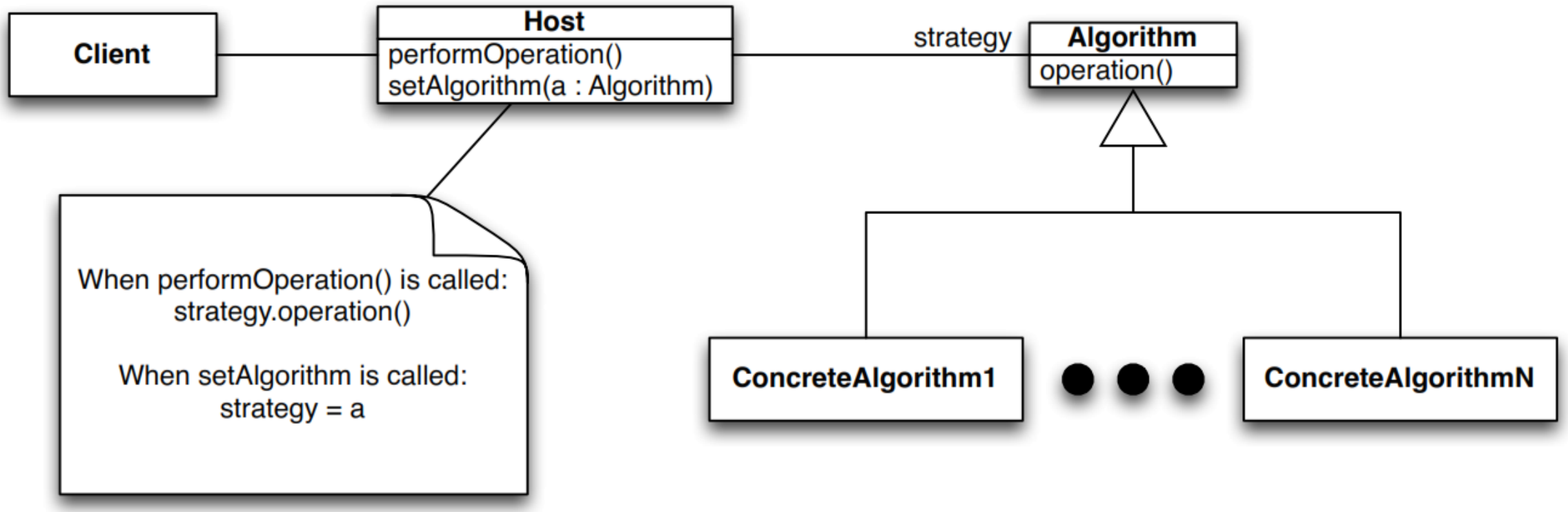
Not Completely Decoupled

- Is DuckSimulator completely decoupled from the Duck subclasses?
  - All of its variables are of type Duck
- No!
  - The subclasses are still coded into DuckSimulator
    - Duck myDuck = new RubberDuck();
- This is a type of coupling…
  - Fortunately, we can eliminate this type of coupling if needed, using a pattern called Factory.
    - We'll see Factory in action later

Strategy Pattern

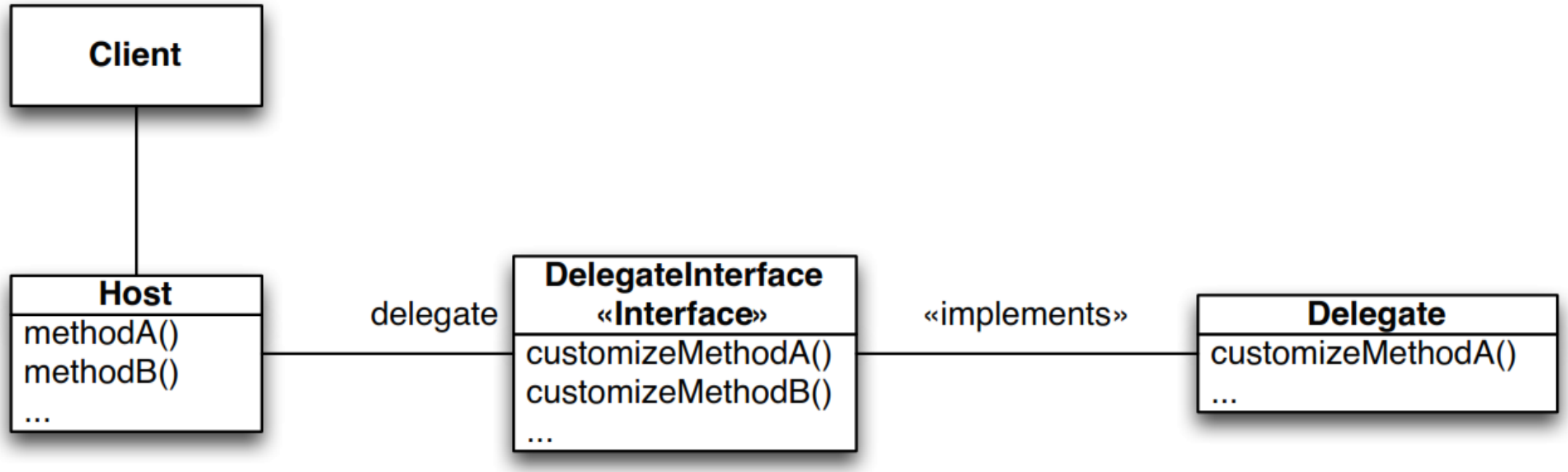
- The solution that we applied to this design problem is known as the **Strategy Design Pattern**
  - It features the following design concepts/principles:
    - **Encapsulate what varies**
    - **Code to an Interface**
    - **Delegation**
    - **Favor Delegation over Inheritance**
  - Definition: The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it

Structure of Strategy



- Algorithm is pulled out of Host. Client only makes use of the public interface of Algorithm and is not tied to concrete subclasses.
- Client can change its behavior by switching among the various concrete algorithms

- Purpose of Delegate:
  - Allow an object's behavior to be customized without forcing a developer to create a subclass that overrides default behavior
- Structure
  - Host object; Delegate Interface; Delegate object; Client
  - Client invokes method on Host; Host checks to see if Delegate handles this method; if so, it routes the call to the Delegate; if not, it provides default behavior for the method

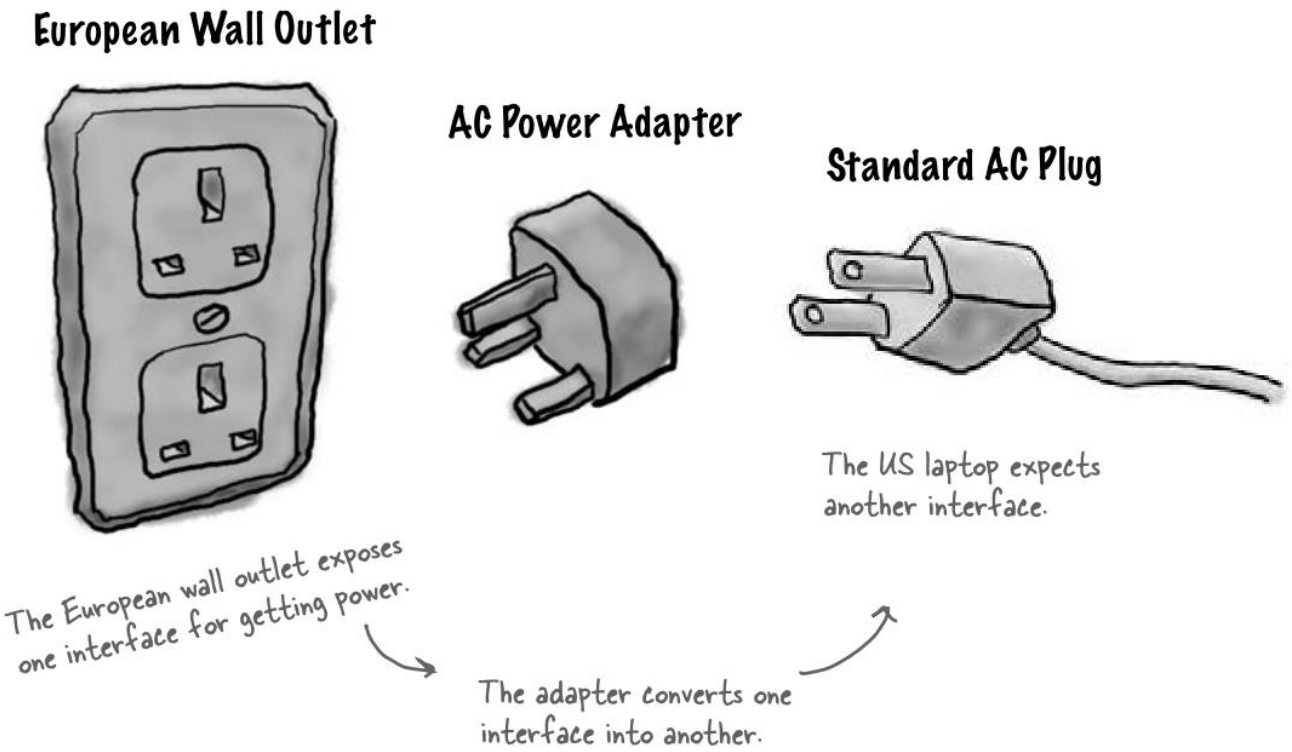


- Here, if Client invokes methodA() on Host, the Delegate's customizeMethodA() will be invoked at some point to help customize Host's behavior for methodA();

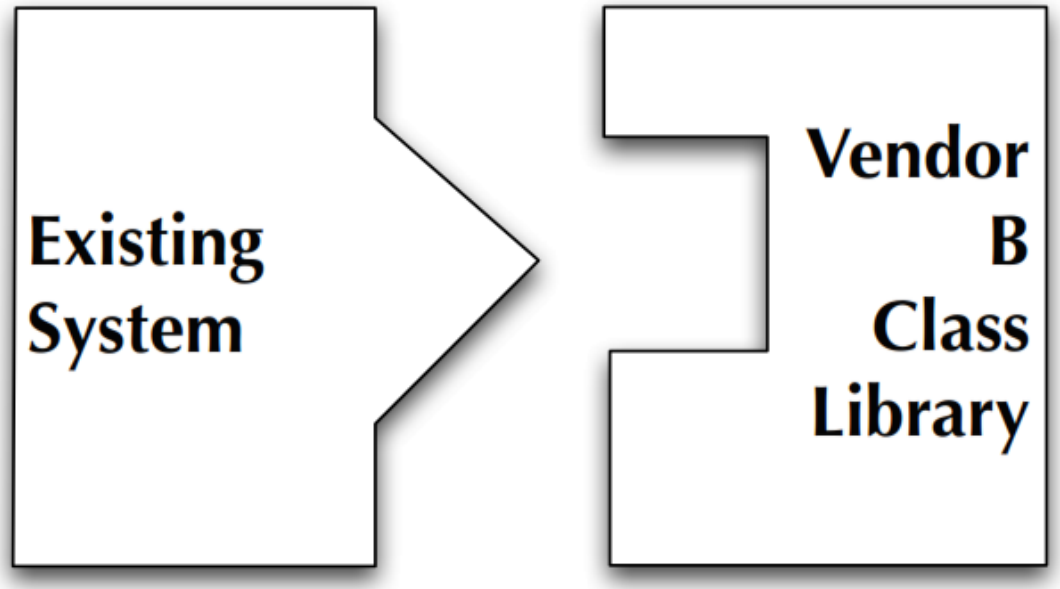
Adapter Pattern

Adapters in the Real World

- Our next pattern provides steps for converting an incompatible interface with an existing system into a different interface that is compatible
  - Real world example: AC power adapters
  - Electronic products made for the USA cannot be used directly with outlets found in most other parts of the world
    - To use these products outside the US, you need an AC power adapter

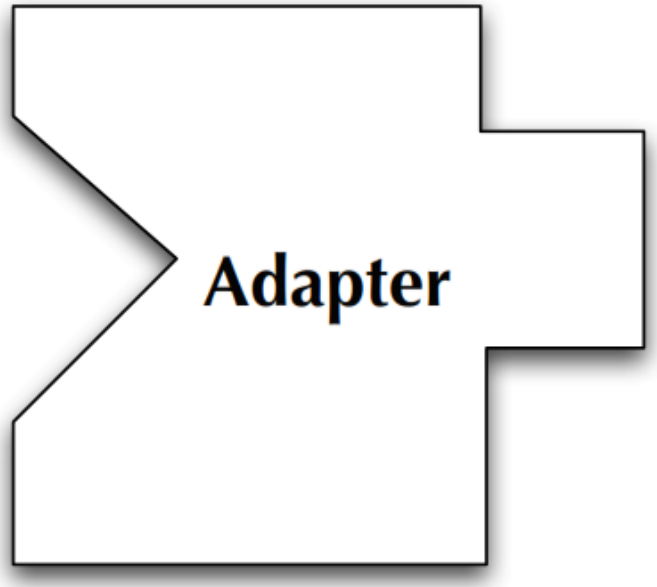


- Pre-condition: You are maintaining an existing system that makes use of a third-party class library from vendor A
- Stimulus: Vendor A goes belly up and corporate policy does not allow you to make use of an unsupported class library
- Response: Vendor B provides a similar class library but its interface is completely different from the interface provided by vendor A
- Assumptions: You don’t want to change your code, and you can’t change vendor B’s code
- Solution?: Write new code that adapts vendor B’s interface to the interface expected by your original code

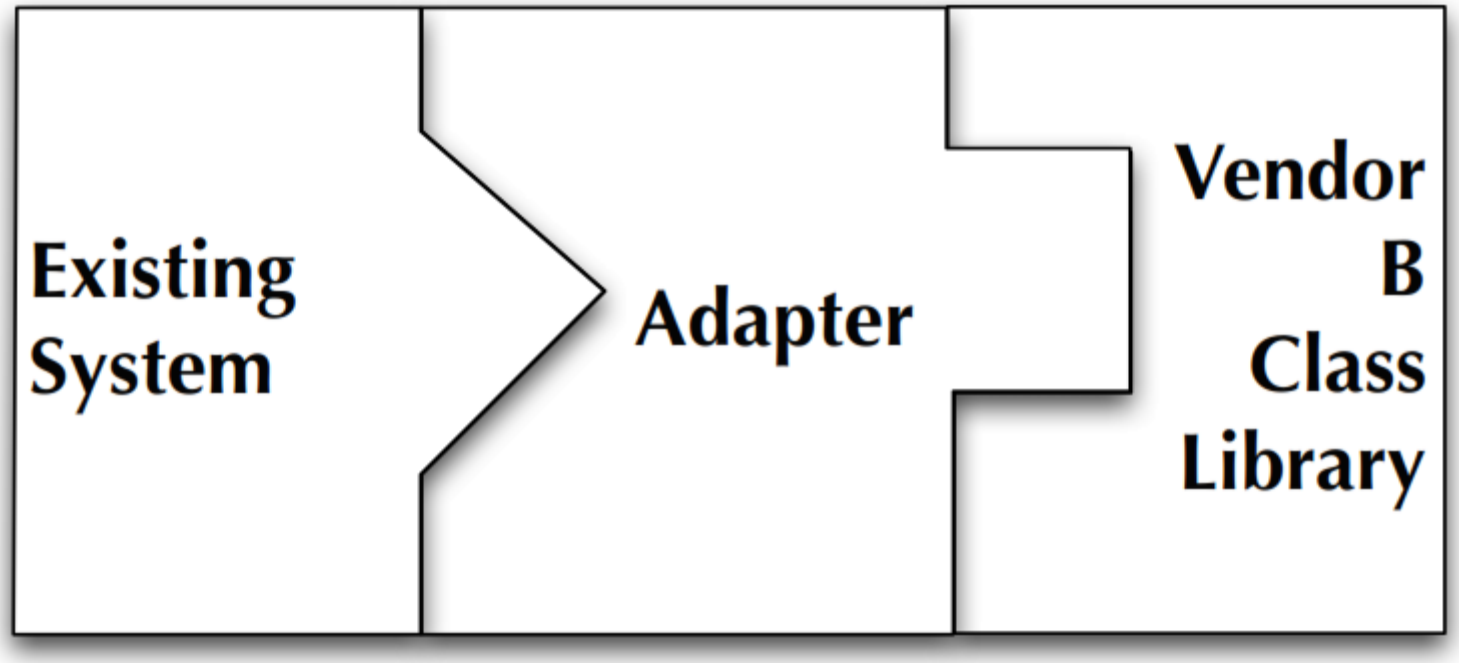


Interface Mismatch
Need Adapter

Create Adapter



And then...



…plug it in

Benefit: Existing system and new vendor library do not change — new code is isolated within the adapter

Example: A Turkey Amongst Ducks! (I)

- If it walks like a duck and quacks like a duck, then it must be a duck!

Or…

- It **might** be a **turkey wrapped with a duck adapter!**

- A slightly different duck model

Duck.java

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

MallarDuck.java

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

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

- An interloper wants to invade the simulator

Turkey.java

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

WildTurkey.java

```
public class WildTurkey implements Turkey {
  public void gobble() {
    System.out.println("Gobble Gobble");
  }

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

But the duck simulator doesn't know how to handle turkeys, only ducks!

- Solution: Write an adapter that makes a turkey look like a duck

TurkeyAdapter.java

```
public class TurkeyAdapter implements Duck {

  private Turkey turkey;

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

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

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

}
```

1. Adapter implements target interface (Duck)

2. Adaptee (turkey) is passed via constructor and stored internally

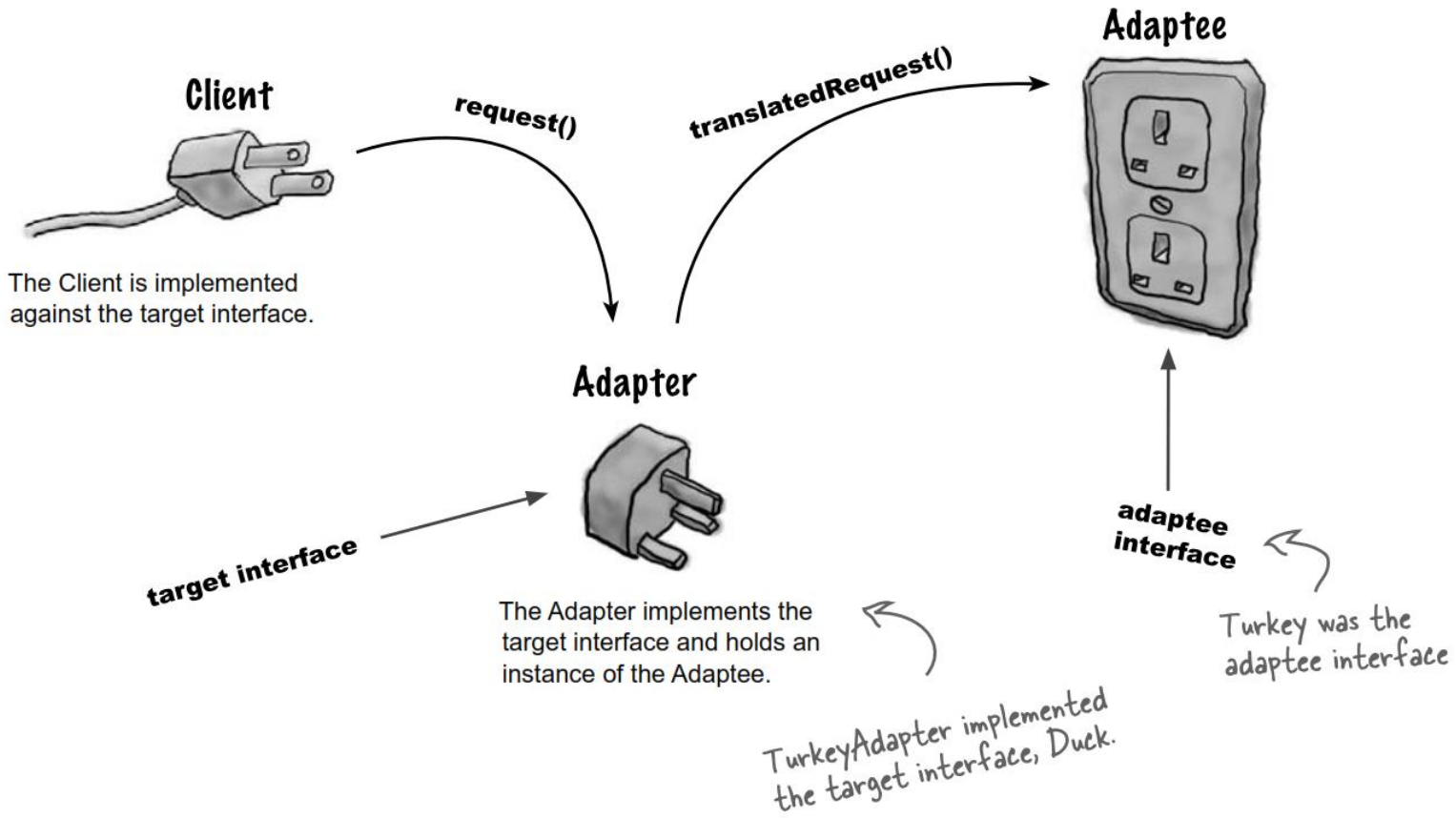
3. Calls by client code are delegated to the appropriate methods in the adaptee

4. Adapter is full-fledged class, could contains additional vars and methods to get its job done; can be used polymorphically as a Duck

DuckSimulator.java

```
import java.util.LinkedList;
import java.util.List;

public class DuckSimulator {

    public static void main(String[] args) {
        MallardDuck mallardDuck = new MallardDuck();

        WildTurkey wildTurkey = new WildTurkey();
        Duck turkeyAdapter = new TurkeyAdapter(wildTurkey);

        List<Duck> ducks = new LinkedList<Duck>();
        ducks.add(mallardDuck);
        ducks.add(turkeyAdapter);

        for (Duck duck : ducks) {
            duck.quack();
            duck.fly();
        }
    }
}
```

The Adapter Pattern explained



Here's how the Client uses the Adapter

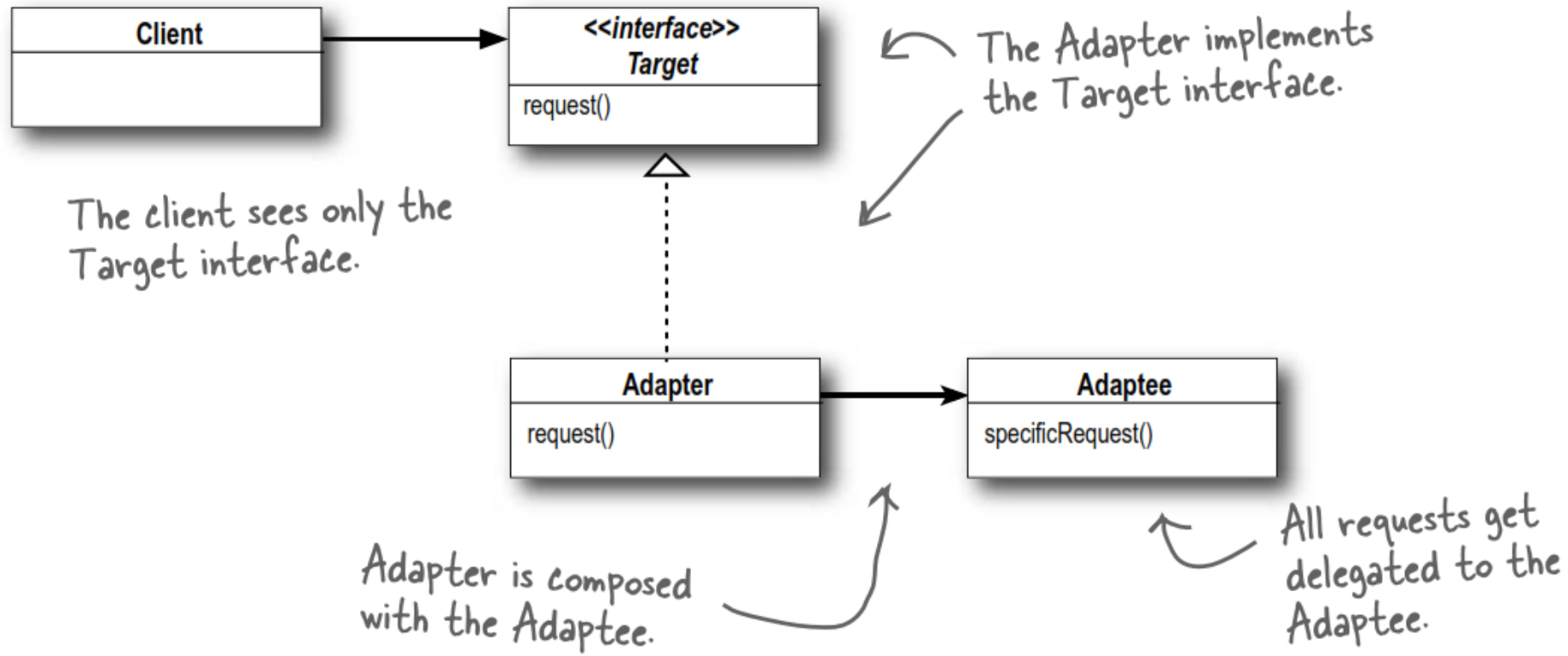
1. The client makes a request to the adapter by calling a method on it using the target interface.

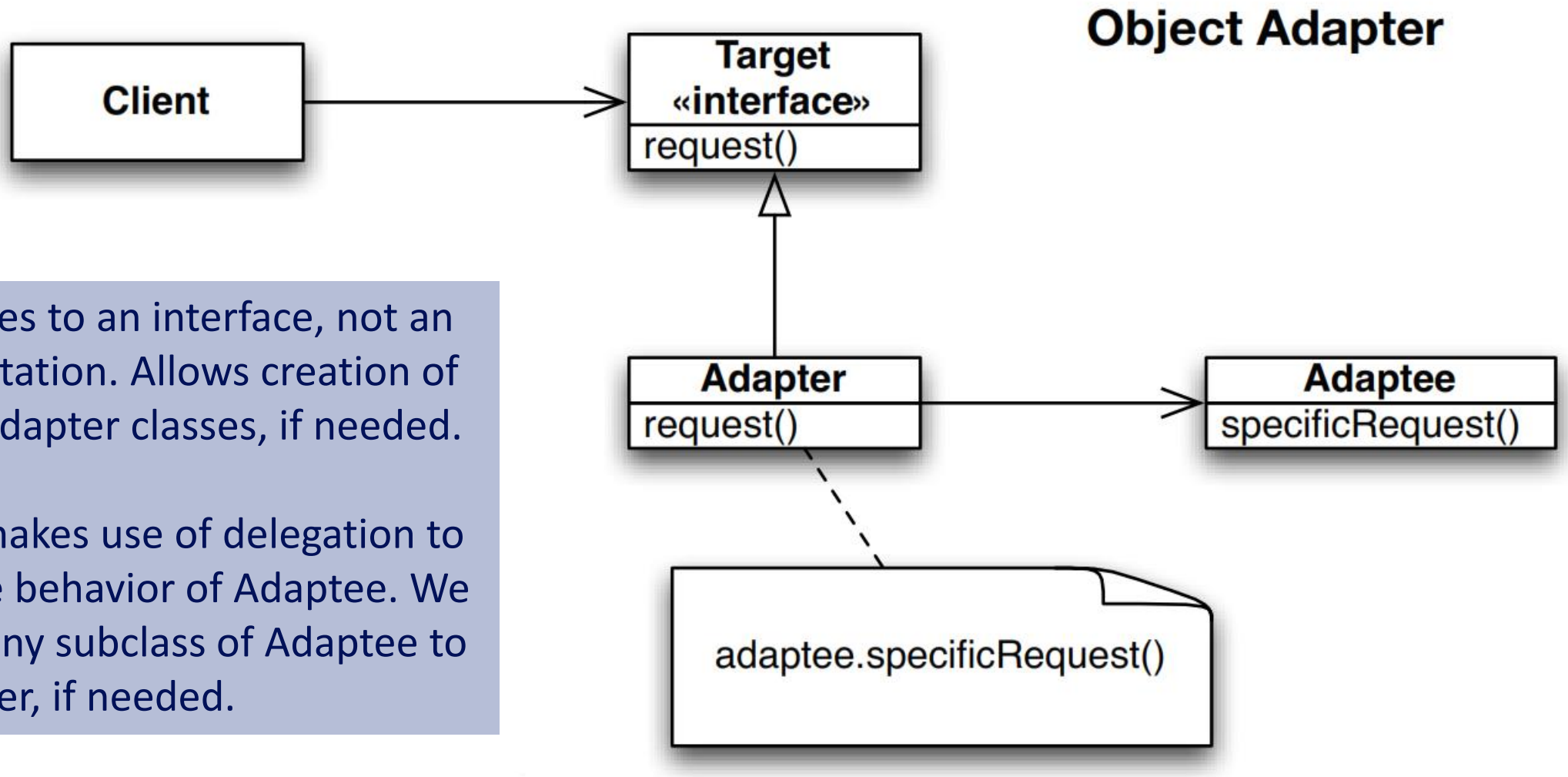
2. The adapter translates the request into one or more calls on the adaptee using the adaptee interface.

3. The client receives the results of the call and never knows there is an adapter doing the translation.

Note that the Client and Adaptee are decoupled – neither knows about the other.

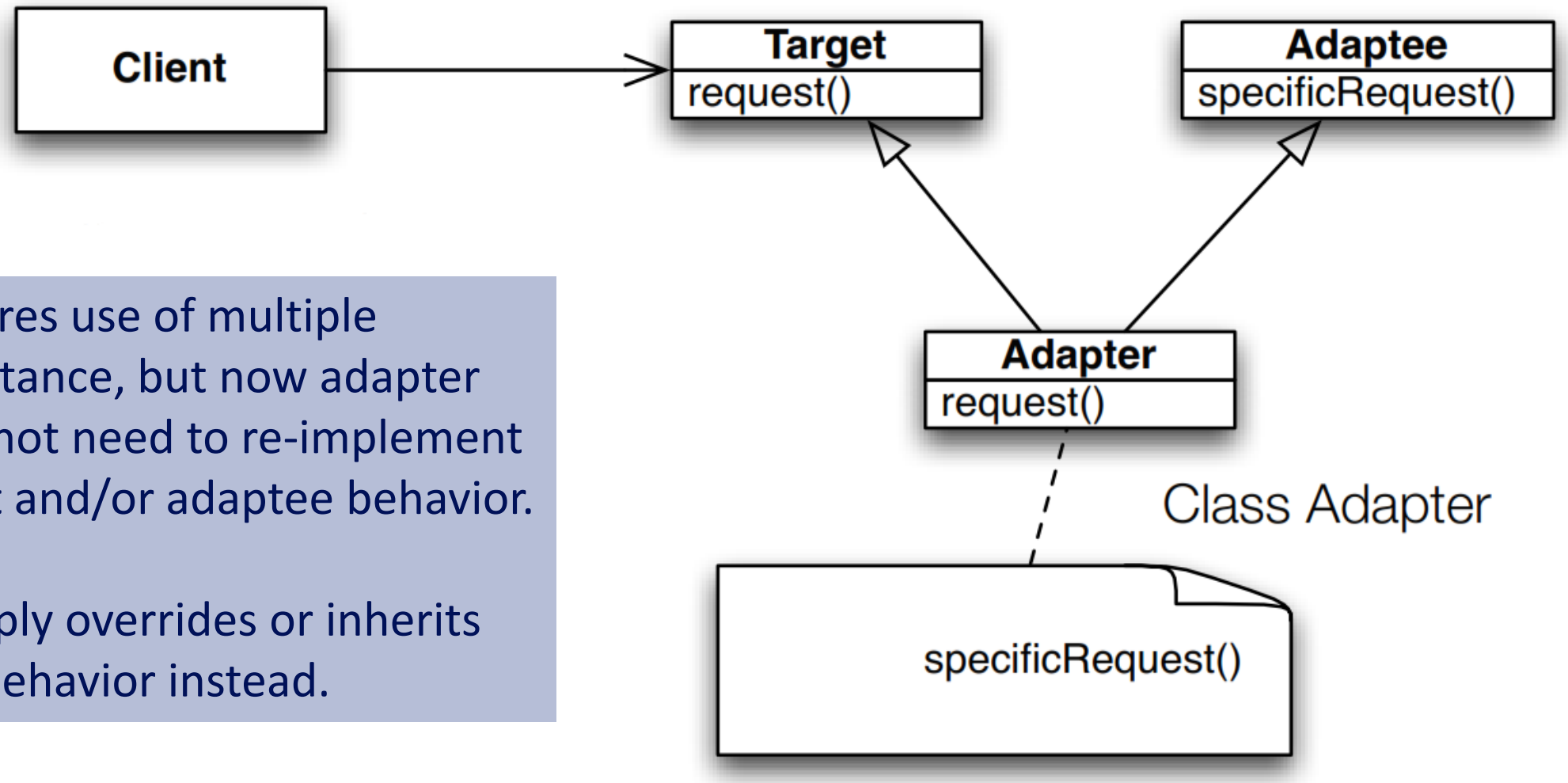
- The Adapter pattern converts the interface of a class into another interface that clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.





1. Client codes to an interface, not an implementation. Allows creation of multiple adapter classes, if needed.

2. Adapter makes use of delegation to access the behavior of Adaptee. We can pass any subclass of Adaptee to the Adapter, if needed.



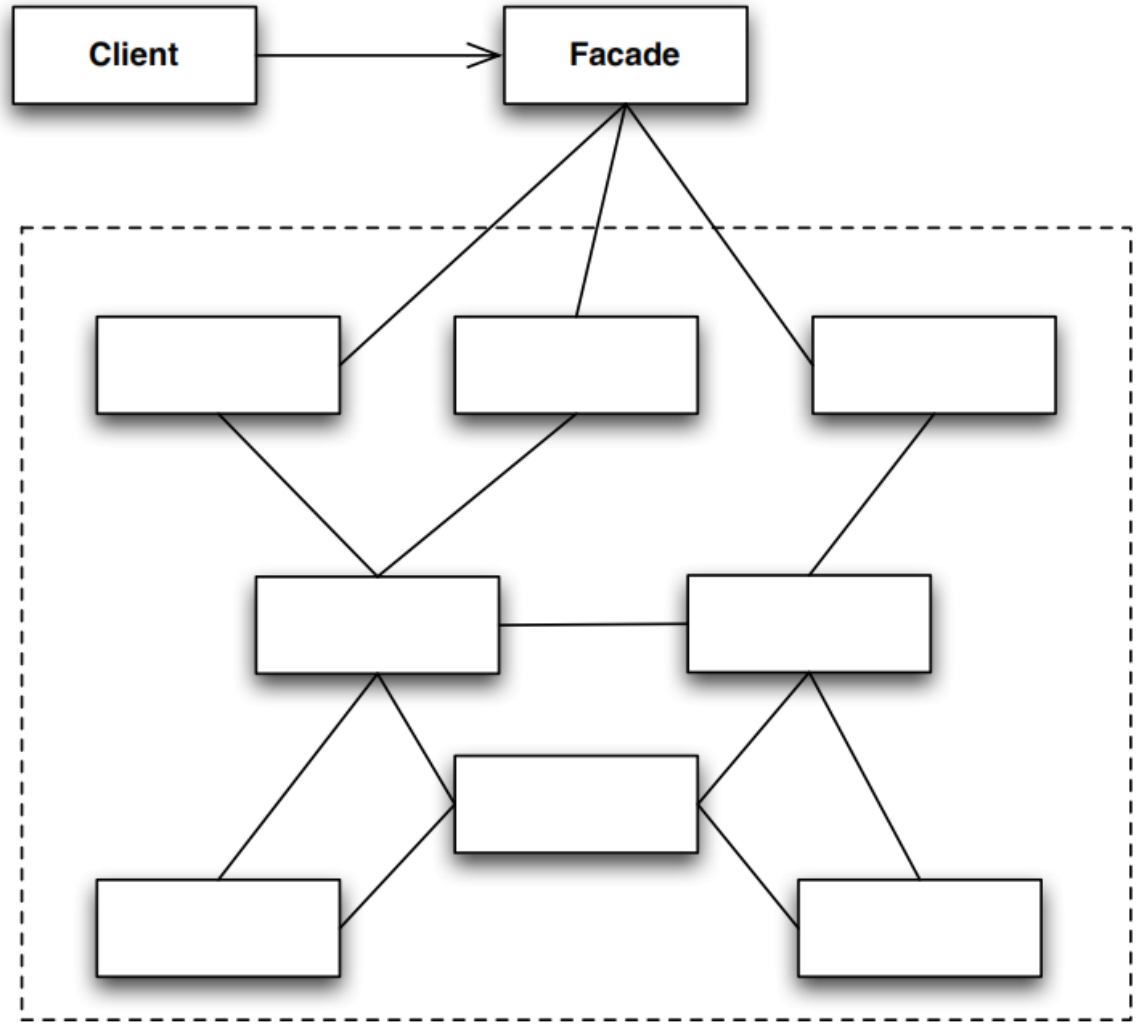
1. Requires use of multiple inheritance, but now adapter does not need to re-implement target and/or adaptee behavior.

2. It simply overrides or inherits that behavior instead.

- Before Java's new collection classes, iteration over a collection occurred via java.util.Enumeration
  - hasMoreElements() : boolean
  - nextElement() : Object
- With the collection classes, iteration was moved to a new interface: java.util.Iterator
  - hasNext(): boolean
  - next(): Object
  - remove(): void
- There's a lot of code out there that makes use of the Enumeration interface
  - New code can still make use of that code by creating an adapter that converts from the Enumeration interface to the Iteration interface

Facade Pattern

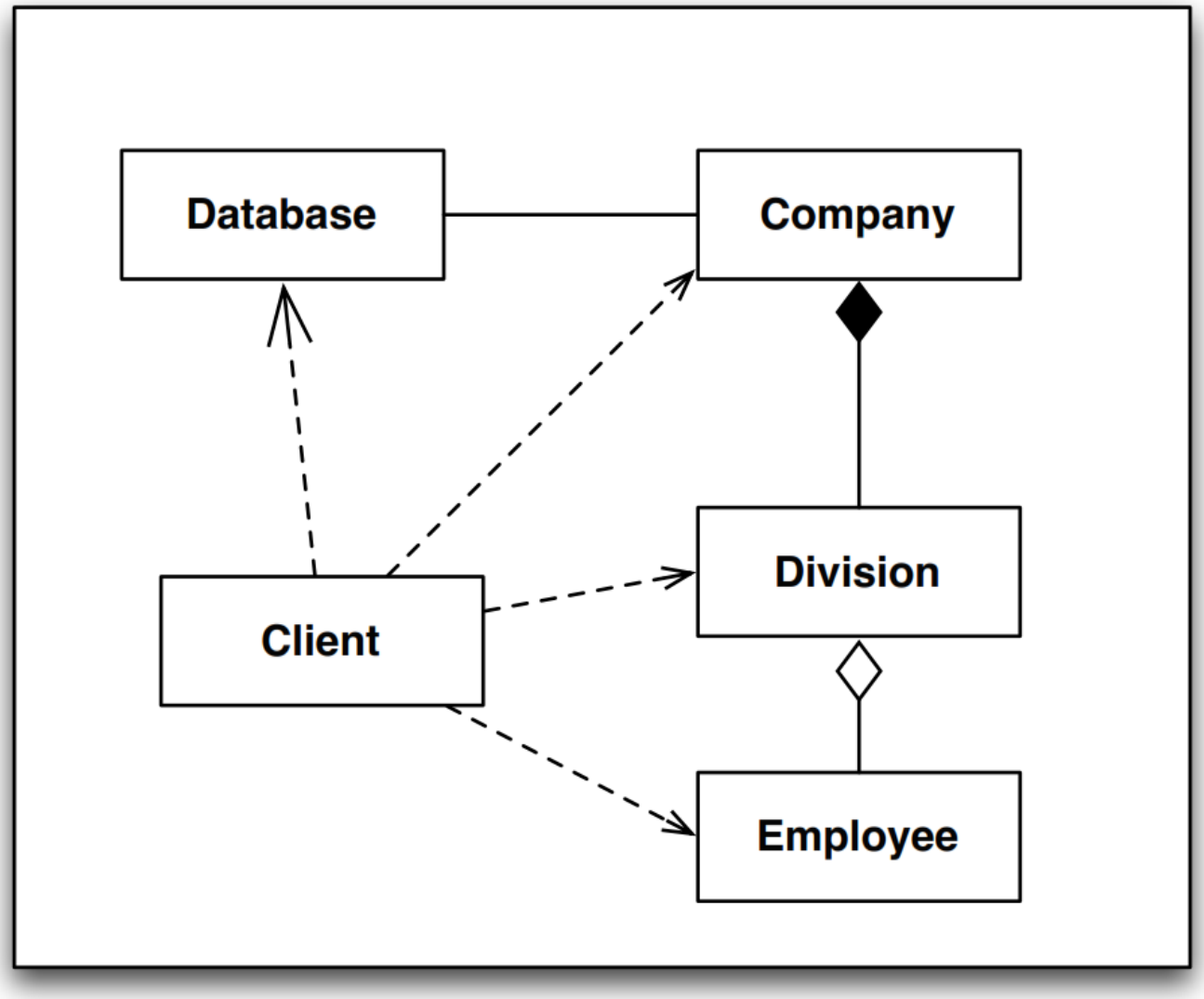
- “Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.”
  - Design Patterns, Gang of Four, 1995
- There can be significant benefit in wrapping a complex subsystem with a simplified interface
  - If you don’t need the advanced functionality or fine-grained control of the former, the latter makes life easy


Client
Facade

Façade (II)

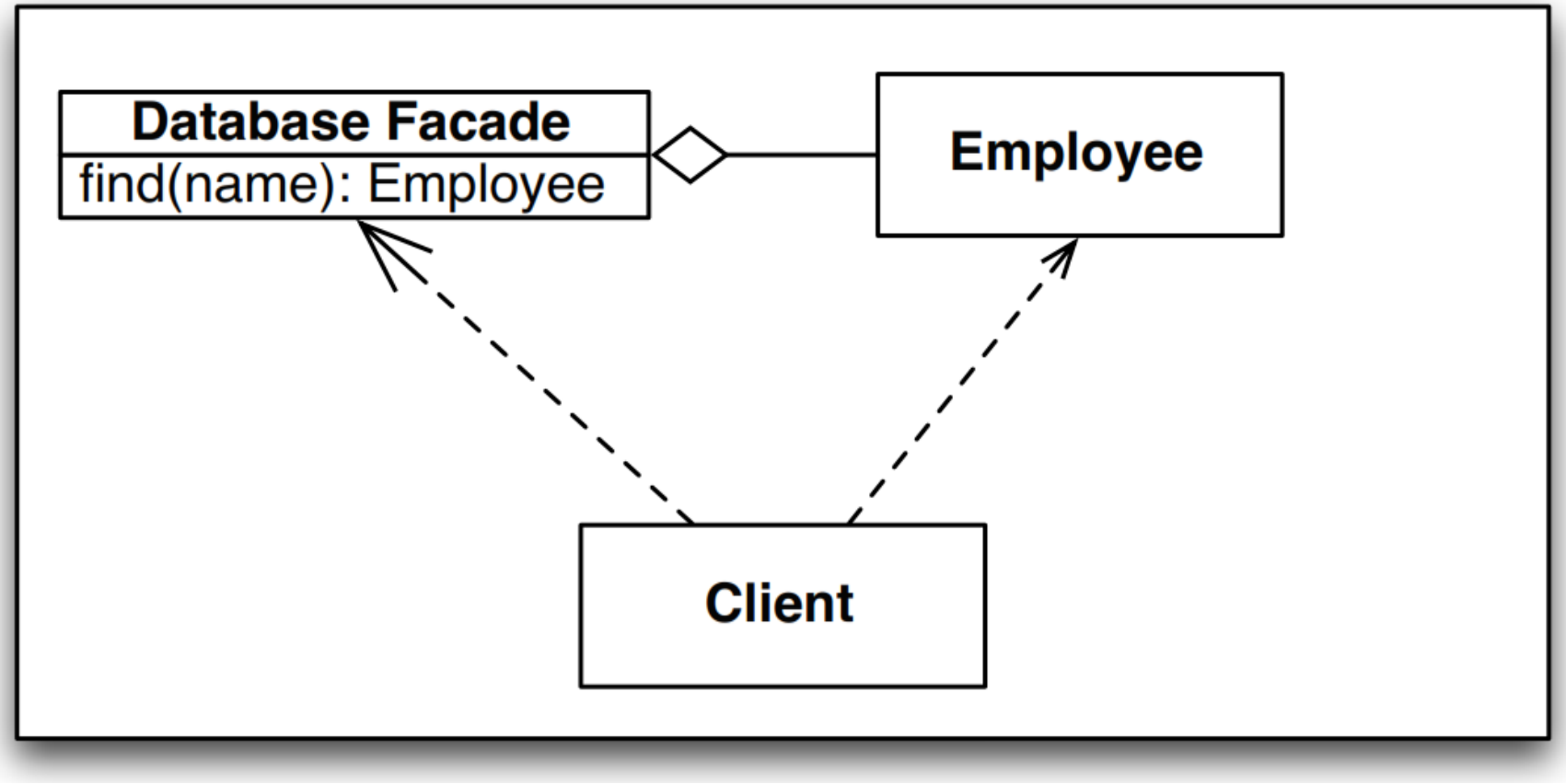
- Façade works best when you are accessing a subset of the system's functionality
  - You can add new features by adding them to the Façade (not the subsystem); you still get a simpler interface
- Façade not only reduces the number of methods you are dealing with but also the number of classes
  - Imagine having to pull Employees out of Divisions that come from Companies that you pull from a Database
    - A Façade in this situation can fetch Employees directly

Example (Without a Façade)



- Without a Façade, Client contacts the Database to retrieve Company objects. It then retrieves Division objects from them and finally gains access to Employee objects.
- It uses four classes.

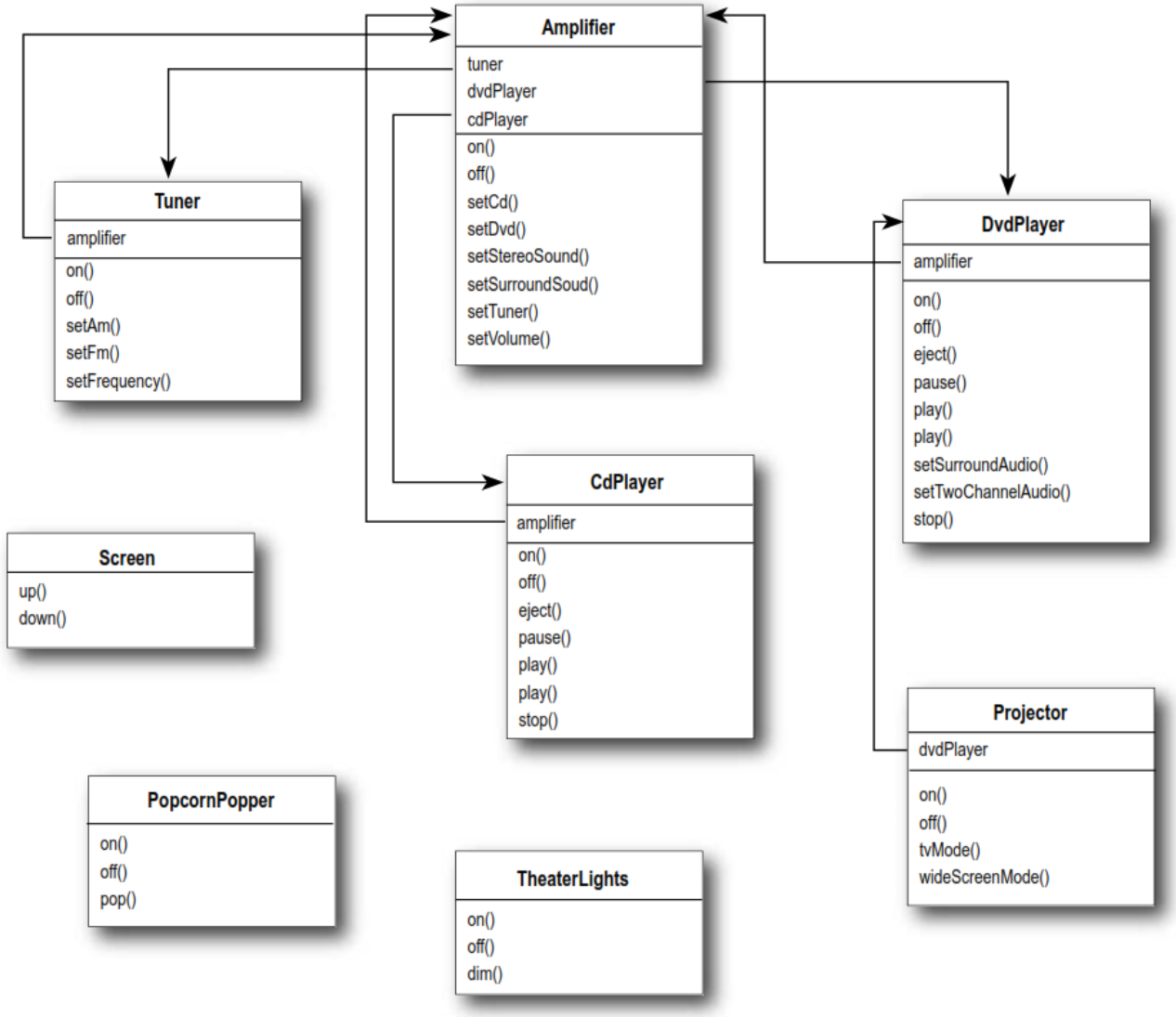
Example (With a Façade)



- With a Façade, the Client is shielded from most of the classes. It uses the Database Façade to retrieve Employee objects directly.

Façade Example (I)

- Imagine a library of classes with a complex interface and/or complex interrelationships
  - Home Theater System
    - Amplifier, DvdPlayer, Projector, CdPlayer, Tuner, Screen, PopcornPopper, and TheatreLights
      - each with its own interface and interclass dependencies



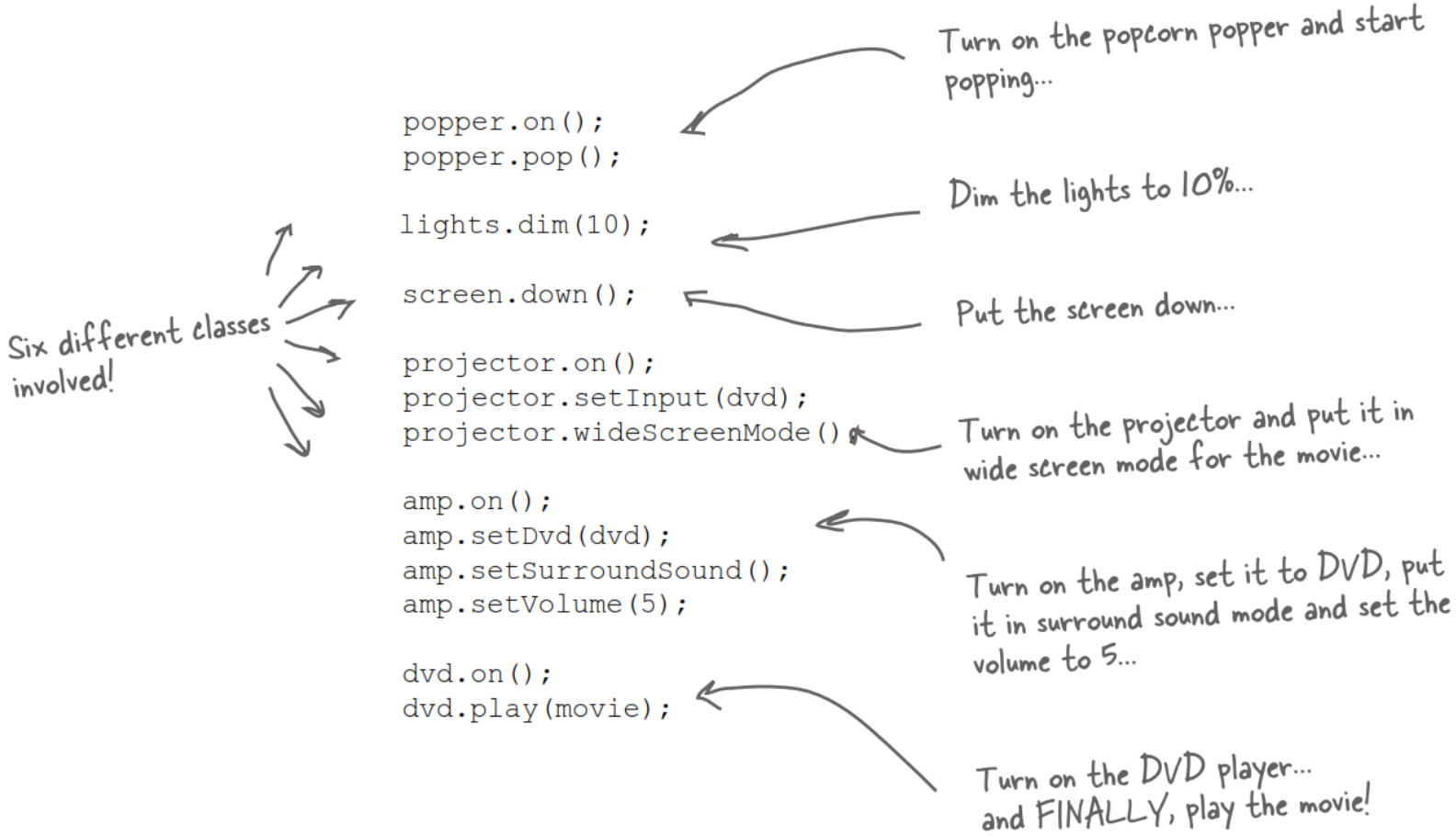
That's a lot of classes, a lot of interactions, and a big set of interfaces to learn and use

Watching a movie (the hard way)

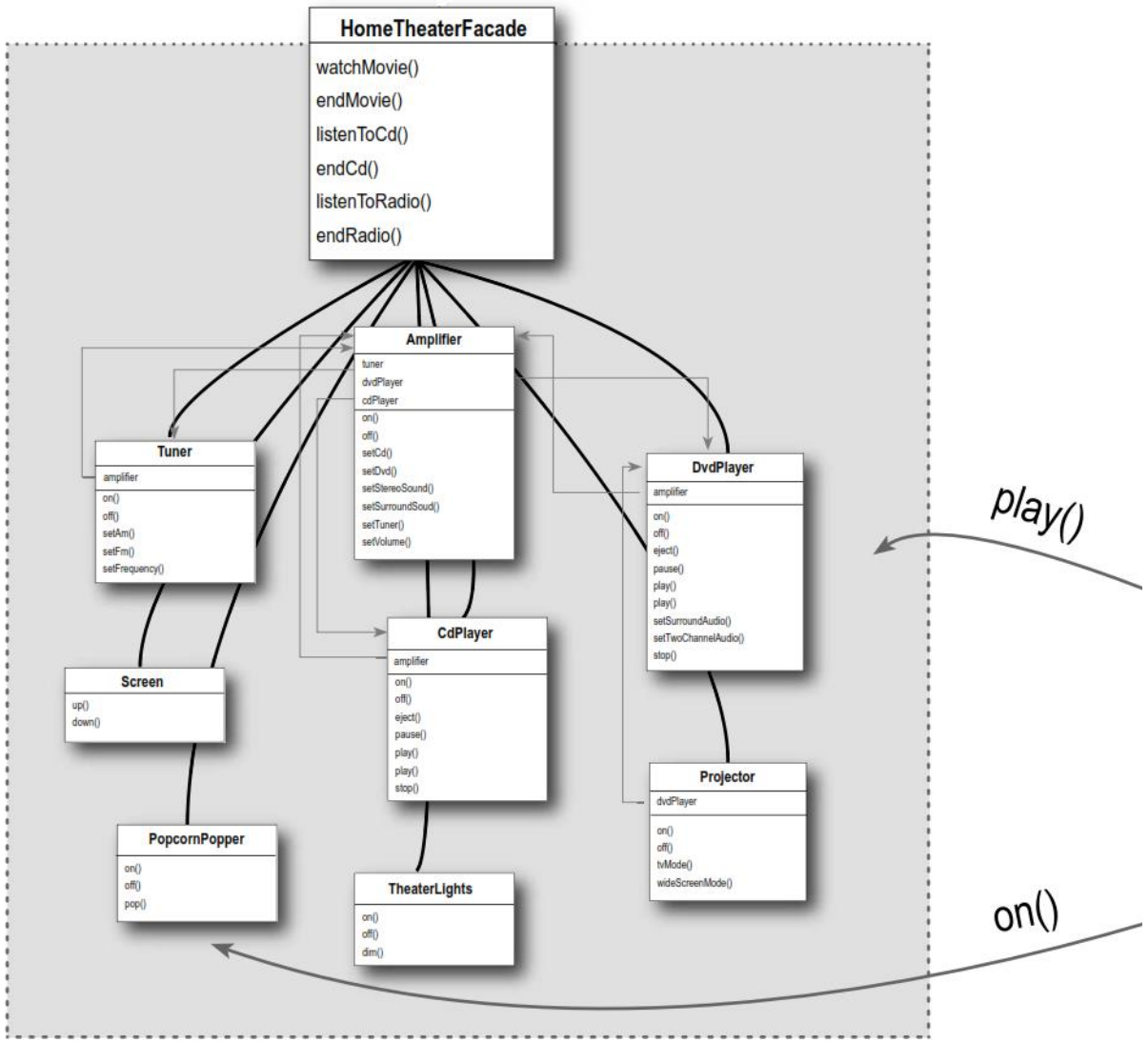
there's just one thing – to watch the movie, you need to perform a few tasks:

1. Turn on the popcorn popper
2. Start the popper popping
3. Dim the lights
4. Put the screen down
5. Turn the projector on
6. Set the projector input to DVD
7. Put the projector on wide-screen mode
8. Turn the sound amplifier on
9. Set the amplifier to DVD input
10. Set the amplifier to surround sound
11. Set the amplifier volume to medium (5)
12. Turn the DVD Player on
13. Start the DVD Player playing

Let's check out those same tasks in terms of the classes and the method calls needed to perform them:



Façade Example (IV)




```
public class HomeTheaterFacade {
    Amplifier amp;
    Tuner tuner;
    DvdPlayer dvd;
    CdPlayer cd;
    Projector projector;
    TheaterLights lights;
    Screen screen;
    PopcornPopper popper;

    public HomeTheaterFacade(Amplifier amp,
                             Tuner tuner,
                             DvdPlayer dvd,
                             CdPlayer cd,
                             Projector projector,
                             Screen screen,
                             TheaterLights lights,
                             PopcornPopper popper) {

        this.amp = amp;
        this.tuner = tuner;
        this.dvd = dvd;
        this.cd = cd;
        this.projector = projector;
        this.screen = screen;
        this.lights = lights;
        this.popper = popper;
    }

    // other methods here

}
```

Here's the composition; these are all the components of the subsystem we are going to use.

The facade is passed a reference to each component of the subsystem in its constructor. The facade then assigns each to the corresponding instance variable.

We're just about to fill these in...

Time to watch a movie (the easy way)

It's SHOWTIME!

```
public class HomeTheaterTestDrive {
    public static void main(String[] args) {
        // instantiate components here

        HomeTheaterFacade homeTheater =
                new HomeTheaterFacade(amp, tuner, dvd, cd,
                        projector, screen, lights, popper);

        homeTheater.watchMovie("Raiders of the Lost Ark");
        homeTheater.endMovie();
    }
}
```

Here we're creating the components right in the test drive. Normally the client is given a facade, it doesn't have to construct one itself.

First you instantiate the Facade with all the components in the subsystem.

Use the simplified interface to first start the movie up, and then shut it down.

- For this example, we can place high level methods…
  - like “watch movie”, “reset system”, “play cd”
- … in a façade object and encode all of the steps for each high level service in the façade;
- Client code is simplified and dependencies are reduced
  - A façade **not only simplifies an interface**, it **decouples a client from a subsystem of components**
- Indeed, Façade lets us **encapsulate subsystems**, hiding them from the rest of the system

Façade vs. Adapter Comparison (I)

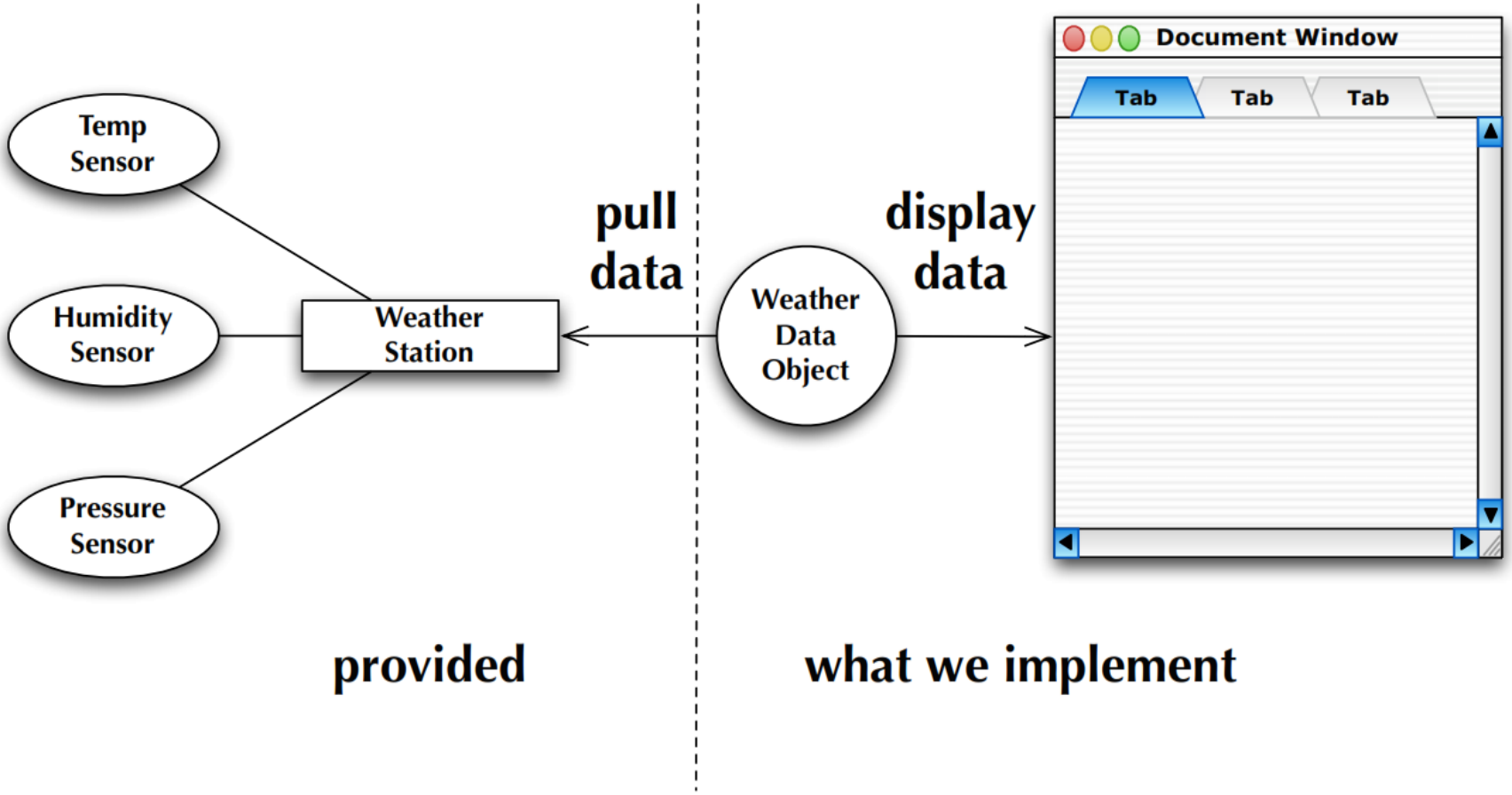
- To many people, two patterns Adapter and Façade appear to be similar
  - They both act as wrappers of a preexisting class
  - They both take an interface that we don’t want and convert it to an interface that we can use
- With Façade, the intent is to simplify the existing interface
- With Adapter, we have a target interface that we are converting to
  - In addition, we often want the adapter to plug into an existing framework and behave polymorphically

Façade vs. Adapter Comparison (II)

- Superficial difference
  - Façade hides many classes
  - Adapter hides only one
- But
  - a Façade can simplify a single, very complex object
  - an adapter can wrap multiple objects at once in order to access all the functionality it needs
- The key is simplify (façade) vs. convert (adapter)

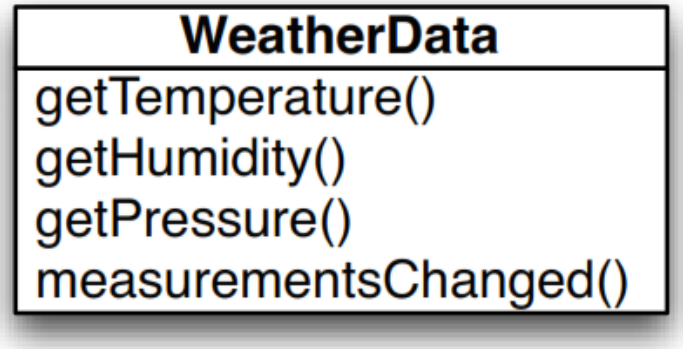
Observer Pattern

- Don’t miss out when something interesting (in your system) happens!
  - The observer pattern allows objects to keep other objects informed about events occurring within a software system (or across multiple systems)
  - It’s dynamic in that an object can choose to receive or not receive notifications at run-time
  - Observer happens to be one of the most heavily used patterns in the Java Development Kit



- We need to pull information from the station and then generate “current conditions, weather stats, and a weather forecast”.

WeatherData Skeleton



- We receive a partial implementation of the WeatherData class from our client.
- They provide three getter methods for the sensor values and an empty measurementsChanged() method that is guaranteed to be called whenever a sensor provides a new value
- We need to pass these values to our three displays… so that's simple!

TurkeyAdapter.java

```
...
  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);

  }
...
```

Problems?

1. **The number and type of displays may vary.**

   These three displays are hard coded with no easy way to update them.

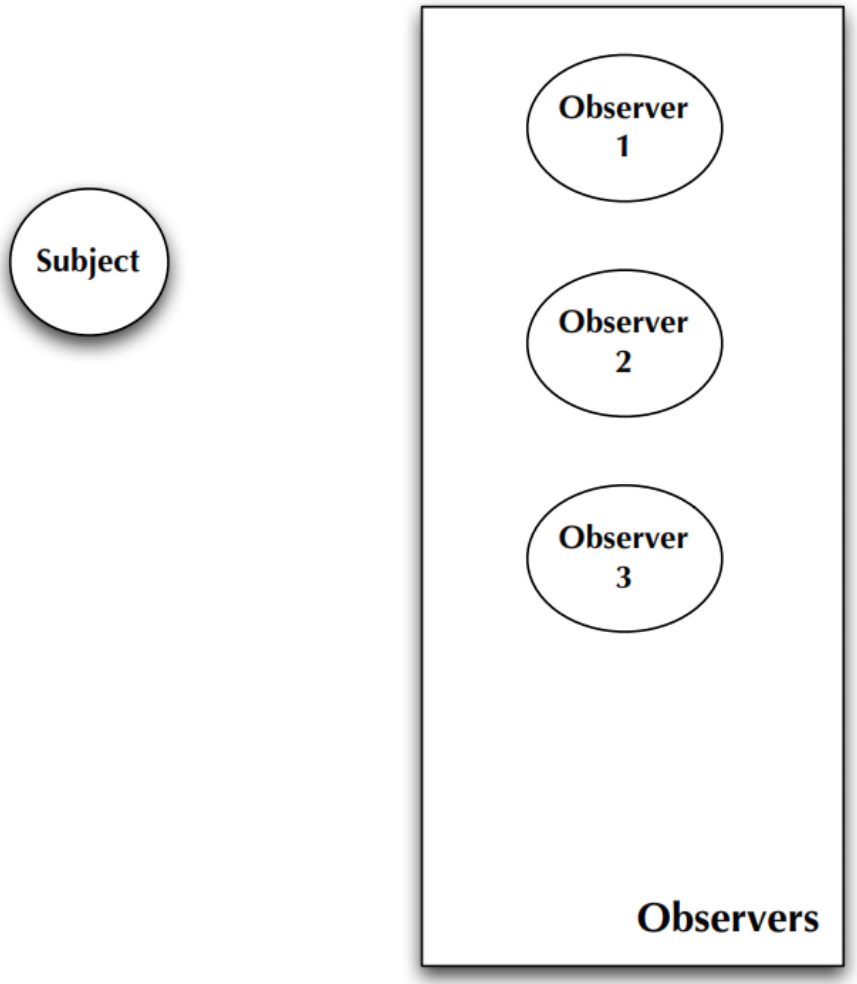
2. **Coding to implementations, not an interface!**

   Each implementation has adopted the same interface, so this will make translation easy!

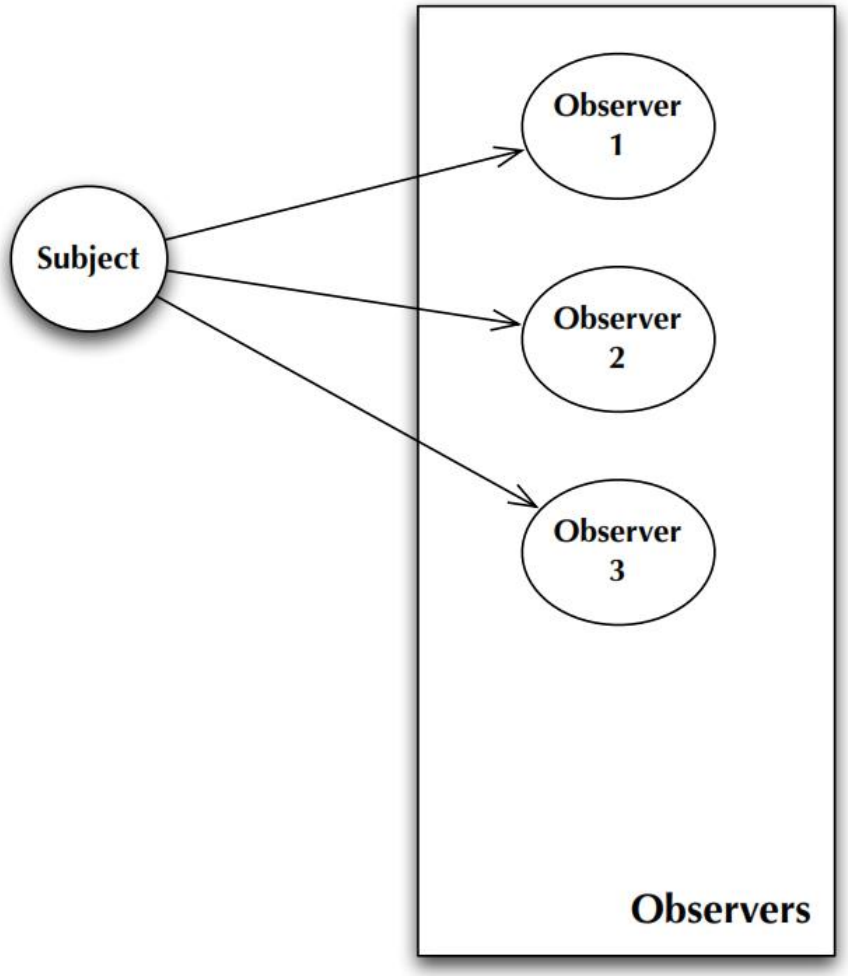
Observer Pattern

- This situation can benefit from use of the observer pattern
  - This pattern is similar to subscribing to a hard copy newspaper
    - A newspaper comes into existence and starts publishing editions
    - You become interested in the newspaper and subscribe to it
    - Any time an edition becomes available, you are notified (by the fact that it is delivered to you)
    - When you don't want the paper anymore, you unsubscribe
    - The newspaper's current set of subscribers can change at any time
  - Observer is just like this but we call the publisher the "subject" and we refer to subscribers as "observers"

Observer in Action (I)

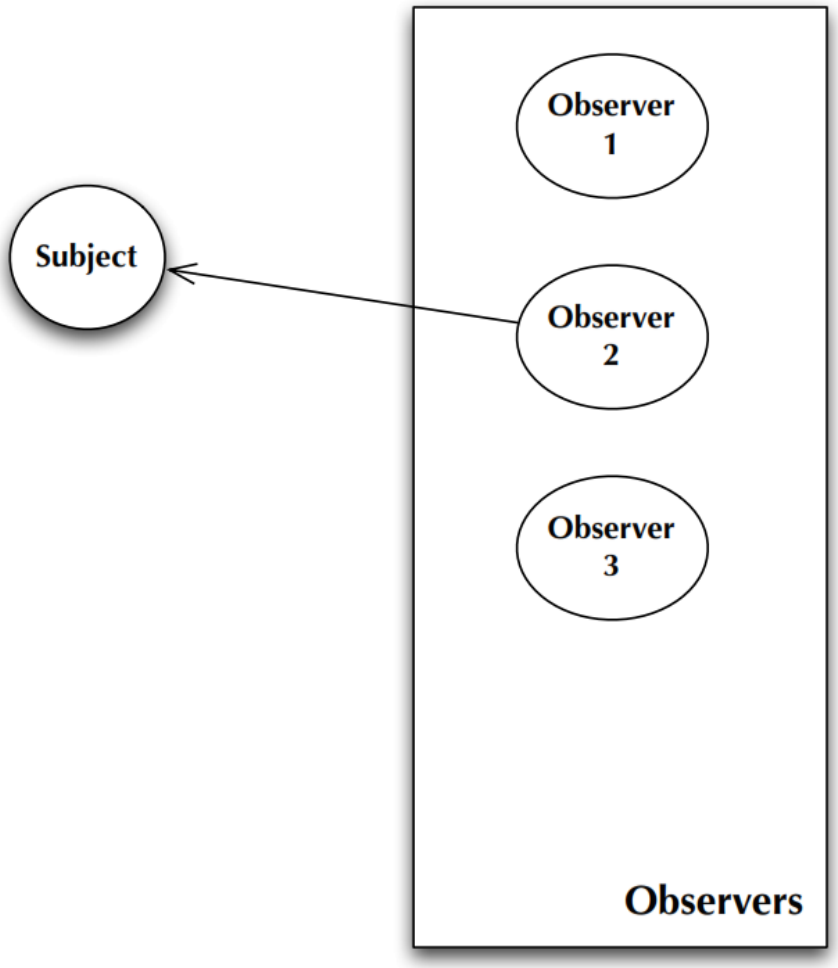


Subject maintains a list of observers



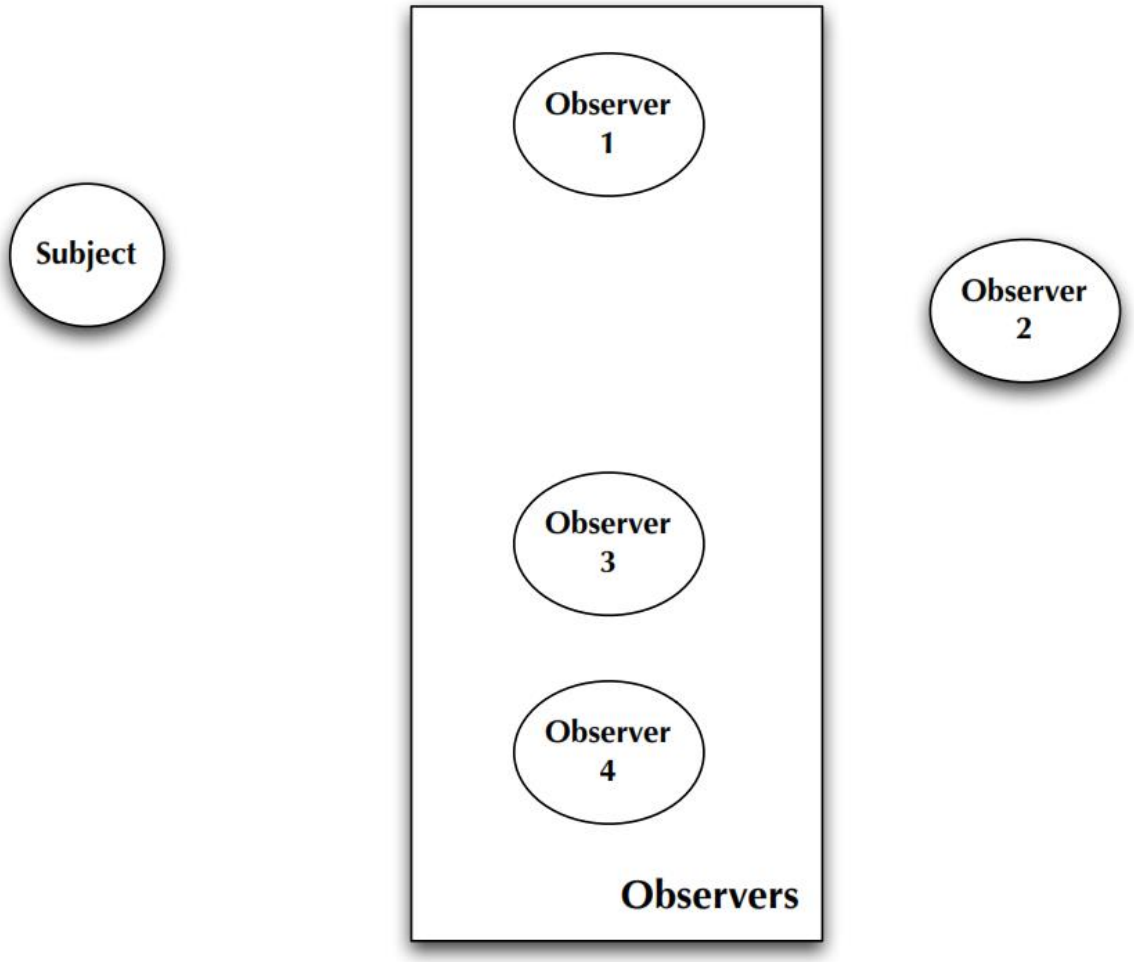
If the Subject changes, it notifies its observers

Observer in Action (III)

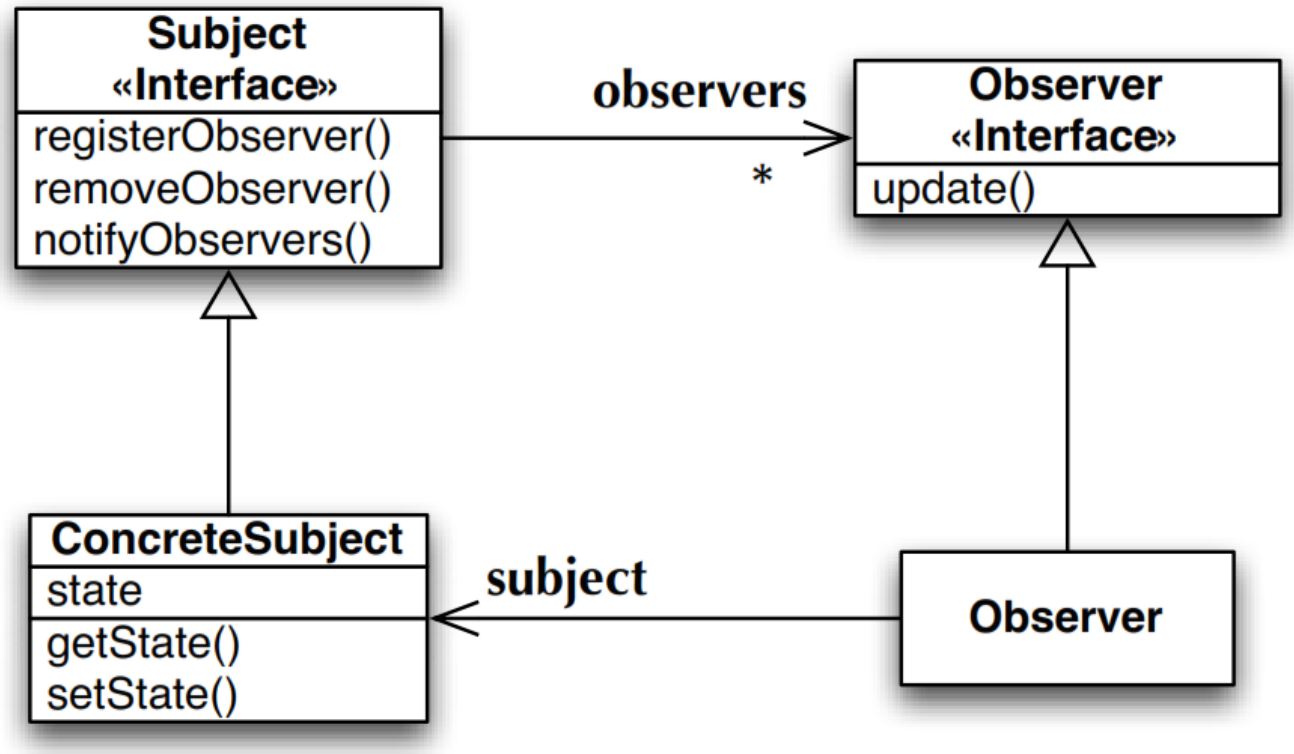


If needed, an observer may query its subject for more information

Observer in Action (IV)



At any point, an observer may join or leave the set of observers



- The Observer Pattern defines a one-to-many dependency between a set of objects, such that when one object (the subject) changes all of its dependents (observers) are notified and updated automatically

- Observer affords a loosely coupled interaction between subject and observer
  - This means they can interact with very little knowledge about each other
- Consider
  - The subject only knows that observers implement the Observer interface
    - We can add/remove observers of any type at any time
    - We never have to modify subject to add a new type of observer
  - We can reuse subjects and observers in other contexts
    - The interfaces plug-and-play anywhere observer is used
  - Observers may have to know about the ConcreteSubject class if it provides many different state-related methods
    - Otherwise, data can be passed to observers via the update() method

Factory Pattern

- Each time we invoke the “new” command to create a new object, we violate the “Code to an Interface” design principle
- Example

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

- Even though our variable uses an “interface”, this code depends on “DecoyDuck”
- In addition, if you have code that instantiates a particular subtype based on the current state of the program, then the code depends on each concrete class

```
if (hunting) {
    return new DecoyDuck()
} else {
    return new RubberDuck()
}
```

Obvious problems:
needs to be recompiled each time a dependent changes;
add new classes -> change this code
remove existing classes -> change this code

This means that this code violates the open-closed principle and the “encapsulate what varies” design principle

PizzaStore Example

- We have a pizza store program that wants to separate the process of creating a pizza from the process of preparing/ordering a pizza
- Initial code: mixes the two processes

PizzaStore.java

```
public class PizzaStore {

    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;
    }
}
```

Creation
Creation code has all the same problems as the code earlier

Preparation

PizzaStore.java

```
public class PizzaStore {

    private SimplePizzaFactory factory;

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

    public Pizza orderPizza(String type) {

        Pizza pizza = factory.createPizza(type);

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

        return pizza;
    }
}
```

- A simple way to encapsulate this code is to put it in a separate class
  - That new class depends on the concrete classes, but those dependencies no longer impact the preparation code

SimplePizzaFactory.java

```
public class SimplePizzaFactory {

    public Pizza createPizza(String type) {
        if (type.equals("cheese")) {
            return new CheesePizza();
        } else if (type.equals("greek")) {
            return new GreekPizza();
        } else if (type.equals("pepperoni")) {
            return new PepperoniPizza();
        }
    }
}
```

Pizza.java

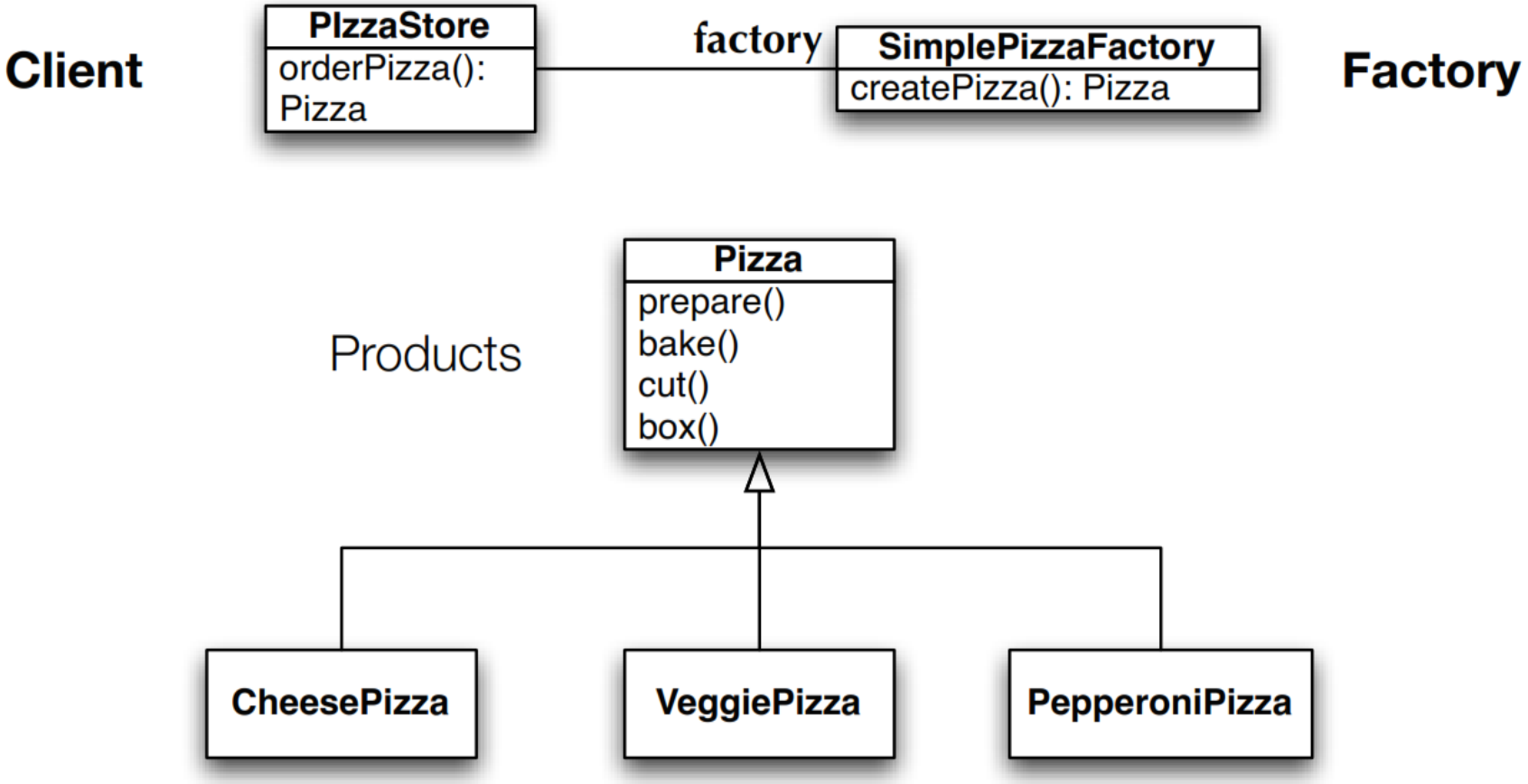
```
import java.util.ArrayList;
import java.util.List;

public abstract class Pizza {

    String name;
    String dough;
    String sauce;
    List<String> toppings = new ArrayList<String>();

    public String getName() {
        return name;
    }

    public void prepare() {
        System.out.println("Preparing " + name);
    }

    public void bake() {
        System.out.println("Baking " + name);
    }
```

Pizza.java (countinue)

```
    public void cut() {
        System.out.println("Cutting " + name);
    }

    public void box() {
       System.out.println("Boxing " + name);
    }

    public String toString() {
        // code to display pizza name and ingredients
        StringBuffer display = new StringBuffer();
        display.append("---- " + name + " ----\n");
        display.append(dough + "\n");
        display.append(sauce + "\n");

        for (int i = 0; i < toppings.size(); i++) {
            display.append((String )toppings.get(i) + "\n");
        }

        return display.toString();
    }
}
```

PizzaTestDrive.java

```
public class PizzaTestDrive {

    public static void main(String[] args) {
        System.out.println("");
        SimplePizzaFactory factory = new SimplePizzaFactory();
        PizzaStore store = new PizzaStore(factory);

        Pizza pizza = store.orderPizza("cheese");
        System.out.println("We ordered a " + pizza.getName() + "\n");

        pizza = store.orderPizza("veggie");
        System.out.println("We ordered a " + pizza.getName() + "\n");
    }
}
```



While this is nice, it is not as flexible as it can be: to increase flexibility we need to look at two design patterns: Factory Method and Abstract Factory

- To demonstrate the FactoryMethod pattern, the pizza store example evolves
  - to include the notion of different franchises
  - that exist in different parts of the country (California, New York, Chicago)
- Each franchise needs its own factory to match the proclivities of the locals
  - However, we want to retain the preparation process that has made PizzaStore such a great success
- The Factory Method design pattern allows you to do this by
  - placing abstract “code to an interface” code in a superclass
  - placing object creation code in a subclass
- PizzaStore becomes an abstract class with an abstract createPizza() method
- We then create subclasses that override createPizza() for each region

PizzaStore.java

```
public abstract class PizzaStore {

    abstract Pizza createPizza(String item);

    public Pizza orderPizza(String type) {
        Pizza pizza = createPizza(type);
        System.out.println("--- Making a " +
                           pizza.getName() + " ---");

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

        return pizza;
    }
}
```

Factory Method

This class is a (very simple) OO framework. The framework provides one service: “order pizza.”

The framework invokes the createPizza factory method to create a pizza that it can then prepare using a well-defined, consistent process.

A “client” of the framework will subclass this class and provide an implementation of the createPizza method.

Any dependencies on concrete “product” classes are encapsulated in the subclass.

New York Pizza Store

NYPizzaStore.java

```
public class NYPizzaStore extends PizzaStore {

    Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new NYStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new NYStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new NYStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new NYStylePepperoniPizza();
        }
        return null;
    }
}
```

- Nice and simple. If you want a NY-style pizza, you create an instance of this class and call orderPizza() passing in the type. The subclass makes sure that the pizza is created using the correct style.
- If you need a different style, create a new subclass.

ChicagoPizzaStore.java

```
public class ChicagoPizzaStore extends PizzaStore {

    Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new ChicagoStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new ChicagoStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new ChicagoStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new ChicagoStylePepperoniPizza();
        }

        return null;
    }
}
```

A couple of the concrete product classes

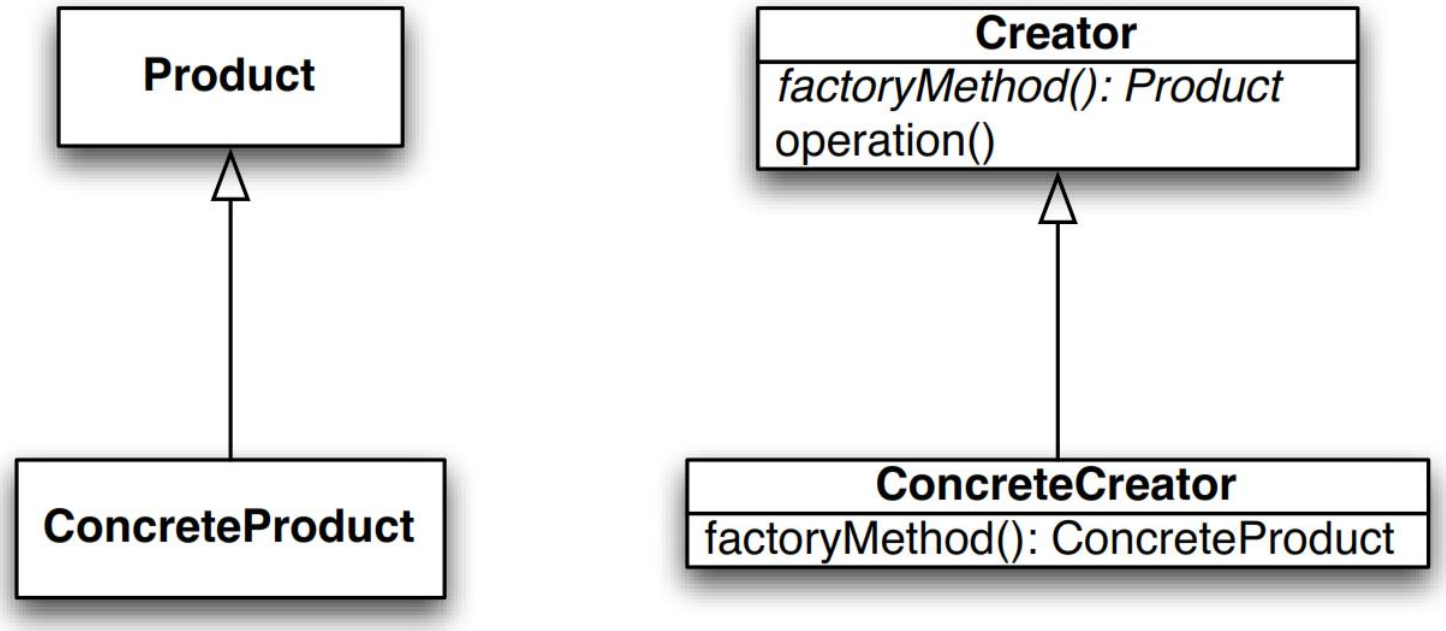
NYStylePepperoniPizza.java

```
public class NYStylePepperoniPizza extends Pizza {

    public NYStylePepperoniPizza() {
        name = "NY Style Pepperoni Pizza";
        dough = "Thin Crust Dough";
        sauce = "Marinara Sauce";

        toppings.add("Grated Reggiano Cheese");
        toppings.add("Sliced Pepperoni");
        toppings.add("Garlic");
        toppings.add("Onion");
        toppings.add("Mushrooms");
        toppings.add("Red Pepper");
    }
}
```

ChicagoStylePepperoniPizza.java

```
public class ChicagoStylePepperoniPizza extends Pizza {

    public ChicagoStylePepperoniPizza() {
        name = "Chicago Style Pepperoni Pizza";
        dough = "Extra Thick Crust Dough";
        sauce = "Plum Tomato Sauce";

        toppings.add("Shredded Mozzarella Cheese");
        toppings.add("Black Olives");
        toppings.add("Spinach");
        toppings.add("Eggplant");
        toppings.add("Sliced Pepperoni");
    }

    void cut() {
        System.out.println("Cutting the pizza
                                into square slices");
    }
}
```

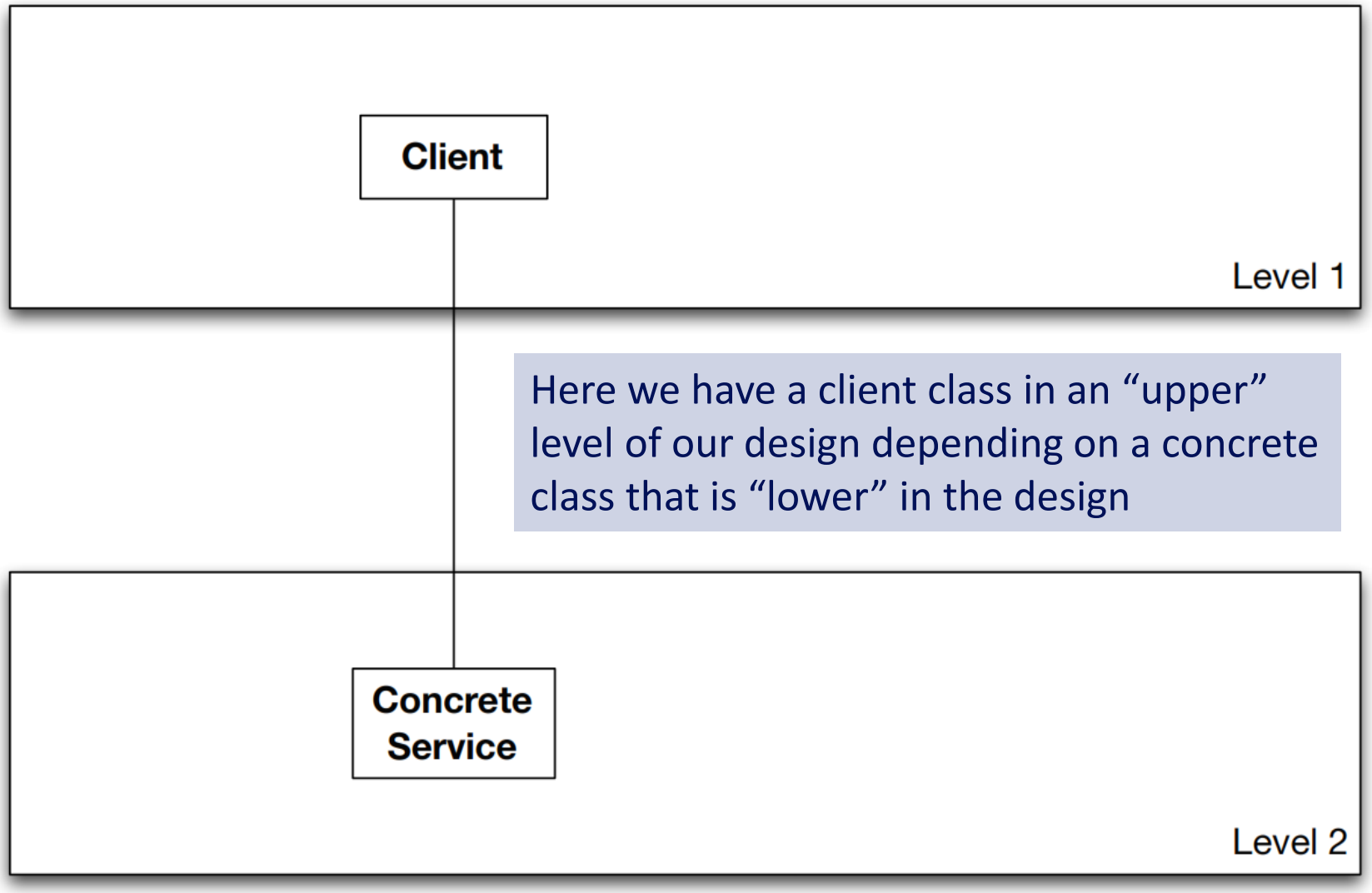
The Factory Method design pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.



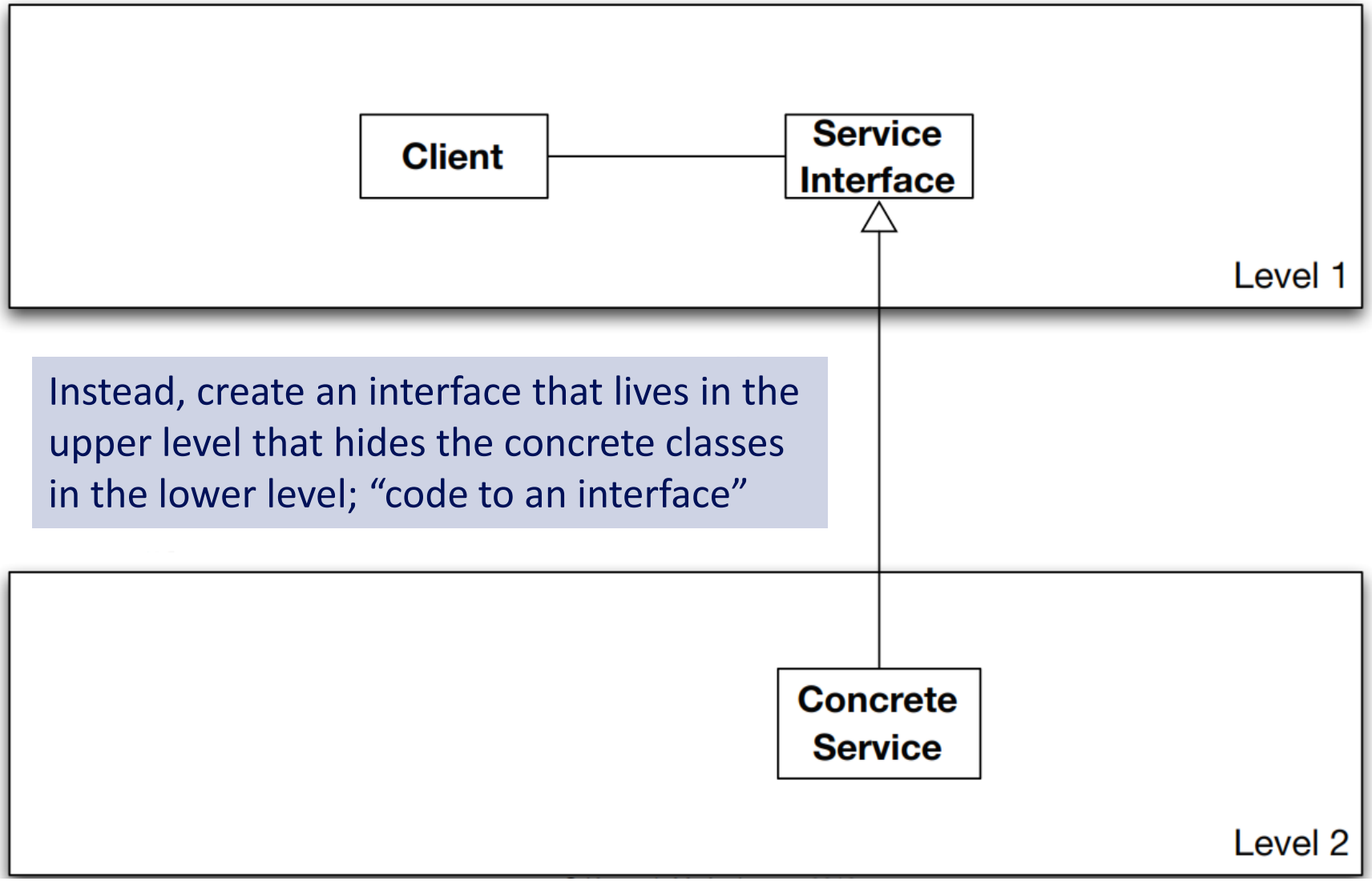
Factory Method leads to the creation of parallel class hierarchies; ConcreateCreators produce instances of ConcreteProducts that are operated on by Creators via the Product interface

- Factory Method is one way of following the dependency inversion principle
  - “Depend upon abstractions. Do not depend upon concrete classes.”
- Normally “high-level” classes depend on “low-level” classes;
  - Instead, they BOTH should depend on an abstract interface

Dependency Inversion Principle: Pictorially



Dependency Inversion Principle: Pictorially



Instead, create an interface that lives in the upper level that hides the concrete classes in the lower level; “code to an interface”

- Factory Method is one way of following the dependency inversion principle
  - "Depend upon abstractions. Do not depend upon concrete classes."
- Normally "high-level" classes depend on "low-level" classes;
  - Instead, they BOTH should depend on an abstract interface
- DependentPizzaStore depends on eight concrete Pizza subclasses
  - PizzaStore, however, depends on the Pizza interface
    - as do the Pizza subclasses
- In this design, PizzaStore (the high-level class) no longer depends on the Pizza subclasses (the low level classes); they both depend on the abstraction "Pizza".

- To achieve the dependency inversion principle in your own designs, follow these GUIDELINES
  - No variable should hold a reference to a concrete class
  - No class should derive from a concrete class
  - No method should override an implemented method of its base classes
- These are “guidelines” because if you were to blindly follow these instructions, you would never produce a system that could be compiled or executed
  - Instead use them as instructions to help optimize your design
- And remember, not only should low-level classes depend on abstractions, but high-level classes should to… this is the very embodiment of “code to an interface”

- The factory method approach to the pizza store is a big success, allowing our company to create multiple franchises across the country quickly and easily
  - But (bad news): we have learned that some of the franchises
    - while following our procedures (the abstract code in PizzaStore forces them to)
    - are skimping on ingredients in order to lower costs and increase margins
  - Our company's success has always been dependent on the use of fresh, quality ingredients
    - So, something must be done!

- We will alter our design such that a factory is used to supply the ingredients that are needed during the pizza creation process
  - Since different regions use different types of ingredients, we'll create region-specific subclasses of the ingredient factory to ensure that the right ingredients are used
  - But, even with region-specific requirements, since we are supplying the factories, we'll make sure that ingredients that meet our quality standards are used by all franchises
    - They'll have to come up with some other way to lower costs.

PizzaIngredientFactory.java

```
public interface PizzaIngredientFactory {
    public Dough createDough();
    public Sauce createSauce();
    public Cheese createCheese();
    public Veggie[] createVeggies();
    public Pepperoni createPepperoni();
    public Clams createClams();
}
```

Note the introduction of more abstract classes: Dough, Sauce, Cheese, etc.

Second, we Implement a Region-Specific Factory

ChicagoPizzaIngredientFactory.java

```
public class ChicagoPizzaIngredientFactory implements PizzaIngredientFactory {
    public Dough createDough() {
        return new ThickCrustDough();
    }

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

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

    public Veggie[] createVeggies() {
        Veggie veggies[] = {new BlackOlives(), new Spinach(), new Eggplant()};
        return veggies;
    }

    public Pepperoni createPepperoni() {
        return new SlicedPepperoni();
    }

    public Clams createClams() {
        return new FrozenClams();
    }
}
```

- This factory ensures that quality ingredients are used during the pizza creation process…
- … while also taking into account the tastes of people in Chicago
- But how (or where) is this factory used?

Within Pizza Subclasses… (I)

Pizza.java

```
public abstract class Pizza {
    String name;

    Dough dough;
    Sauce sauce;
    Veggie veggies[];
    Cheese cheese;
    Pepperoni pepperoni;
    Clams clam;

    abstract void prepare();

    void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }

    void cut() {
        System.out.println("Cutting the pizza into diagonal slices");
    }

    void box() {
        System.out.println("Place pizza in official PizzaStore box");
    }

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

    String getName() {
        return name;
    }
```

First, alter the Pizza abstract base class to make the prepare method abstract…

Pizza.java (Continue)

```
    public String toString() {
        StringBuffer result = new StringBuffer();
        result.append("---- " + name + " ----\n");
        if (dough != null) {
            result.append(dough);
            result.append("\n");
        }
        if (sauce != null) {
            result.append(sauce);
            result.append("\n");
        }
        if (cheese != null) {
            result.append(cheese);
            result.append("\n");
        }
        if (veggies != null) {
            for (int i = 0; i < veggies.length; i++) {
                result.append(veggies[i]);
                if (i < veggies.length-1) {
                    result.append(", ");
                }
            }
            result.append("\n");
        }
        if (clam != null) {
            result.append(clam);
            result.append("\n");
        }
        if (pepperoni != null) {
            result.append(pepperoni);
            result.append("\n");
        }
        return result.toString();
    }
}
```

ChicagoPizzaIngredientFactory.java

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

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

    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
    }
}
```

Then, update pizza subclasses to make use of the factory!

Note: we no longer need subclasses like NYCheesePizza and ChicagoCheesePizza because the ingredient factory now handles regional differences

ChicagoPizzaStore.java

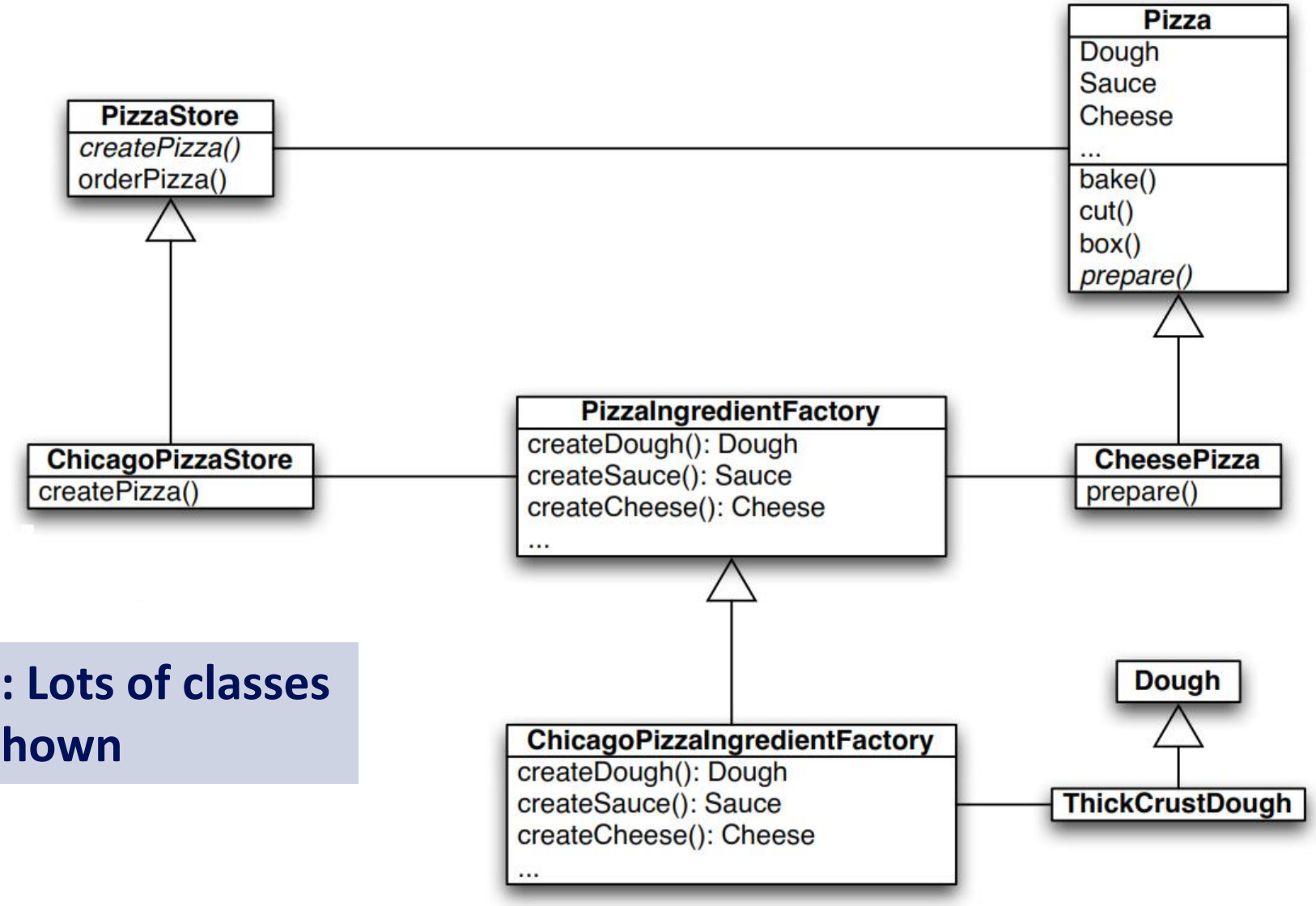
```
public class ChicagoPizzaStore extends PizzaStore {

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

        if (item.equals("cheese")) {
            pizza = new CheesePizza(ingredientFactory);
            pizza.setName("Chicago Style Cheese Pizza");
        } else if (item.equals("veggie")) {
            pizza = new VeggiePizza(ingredientFactory);
            pizza.setName("Chicago Style Veggie Pizza");
        } else if (item.equals("clam")) {
            pizza = new ClamPizza(ingredientFactory);
            pizza.setName("Chicago Style Clam Pizza");
        } else if (item.equals("pepperoni")) {
            pizza = new PepperoniPizza(ingredientFactory);
            pizza.setName("Chicago Style Pepperoni Pizza");
        }
        return pizza;
    }
}
```

We need to update our PizzaStore subclasses to create the appropriate ingredient factory and pass it to each Pizza subclass within the createPizza method

Summary: What did we just do?

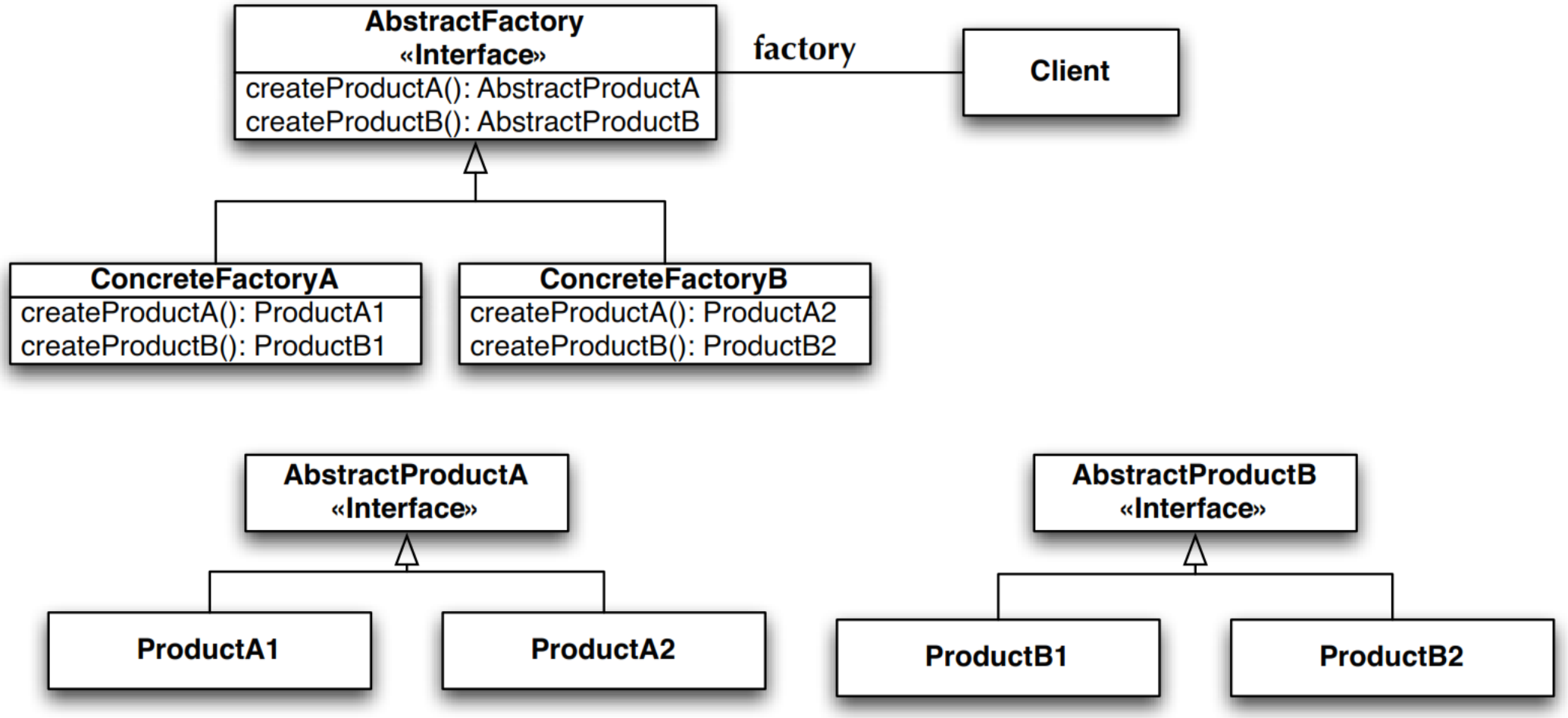
- We created an ingredient factory interface to allow for the creation of a family of ingredients for a particular pizza
- This abstract factory gives us an interface for creating a family of products (e.g., NY pizzas, Chicago pizzas)
  - The factory interface decouples the client code from the actual factory implementations that produce context-specific sets of products
- Our client code (PizzaStore) can then pick the factory appropriate to its region, plug it in, and get the correct style of pizza (Factory Method) with the correct set of ingredients (Abstract Factory)

Class Diagram of Abstract Factory Solution



Note: Lots of classes not shown

- The Abstract Factory design pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes



Abstract Factory Characteristics

- Isolates concrete classes
- Makes exchanging product families easy
- Promotes consistency among products

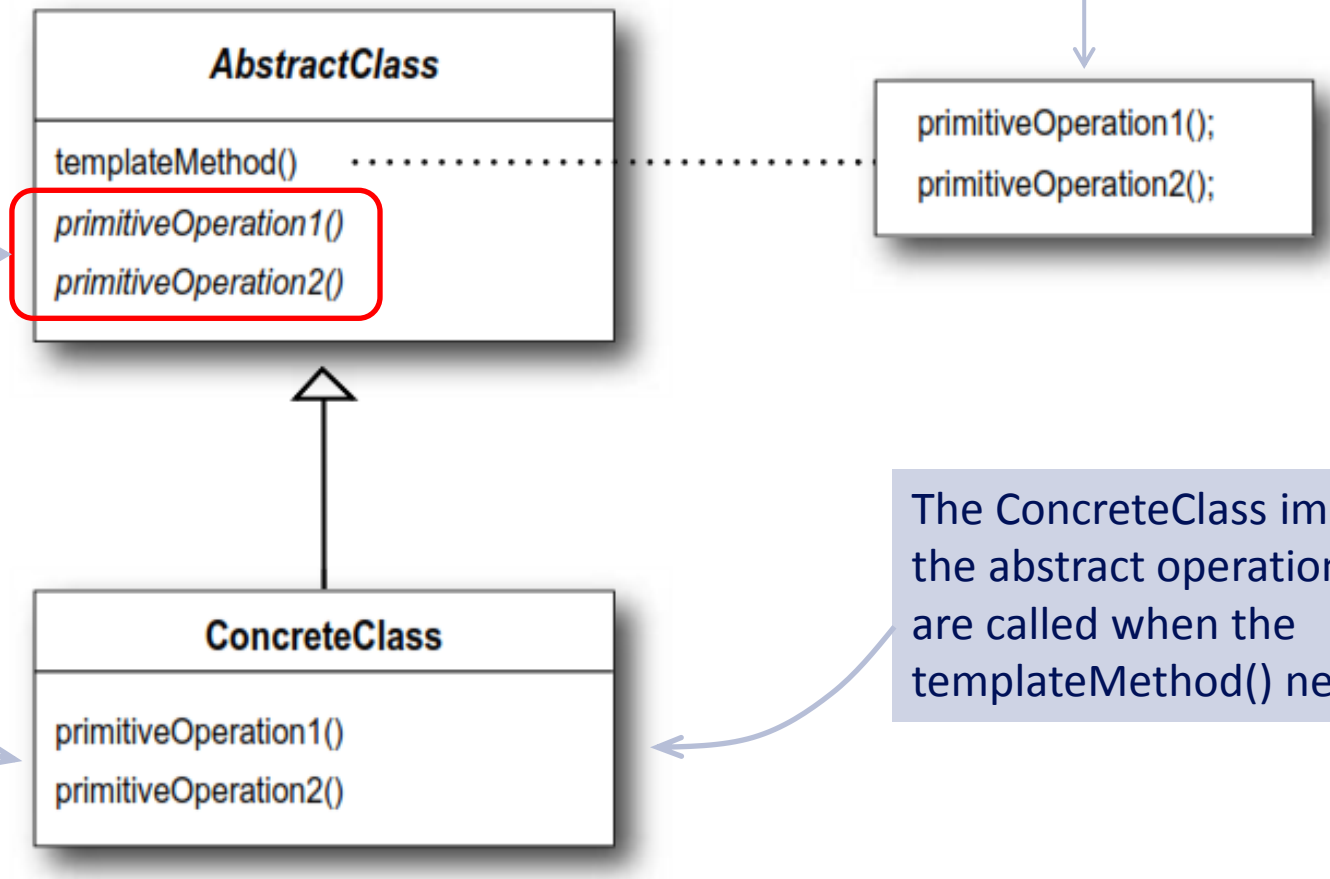
Template Method Pattern

- 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
  - Template Method defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps
    - Makes the algorithm abstract
      - Each step of the algorithm is represented by a method
    - Encapsulates the details of most steps
      - Steps (methods) handled by subclasses are declared abstract
      - Shared steps (concrete methods) are placed in the same class that has the template method, allowing for code re-use among the various subclasses

The AbstractClass contains the template method.

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

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



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.

- Consider an example in which we have recipes for making tea and coffee at a coffee shop
  - Coffee
    - Boil water
    - Brew coffee in boiling water
    - Pour coffee in cup
    - Add sugar and milk
  - Tea
    - Boil water
    - Steep tea in boiling water
    - Pour tea in cup
    - Add lemon

Coffee class for making coffee

Coffee.java

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

    public void boilWater() {
        System.out.println("Boiling water");
    }

    public void brewCoffeeGrinds() {
        System.out.println("Dripping Coffee through filter");
    }

    public void pourInCup() {
        System.out.println("Pouring into cup");
    }

    public void addSugarAndMilk() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

Here's our recipe for coffee, straight out of the training manual.

Each of the steps is implemented as a separate method.

Each of these methods implements one step of the algorithm. There's a method to boil water, brew the coffee, pour the coffee in a cup and add sugar and milk.

Tea class for making tea

Coffee.java

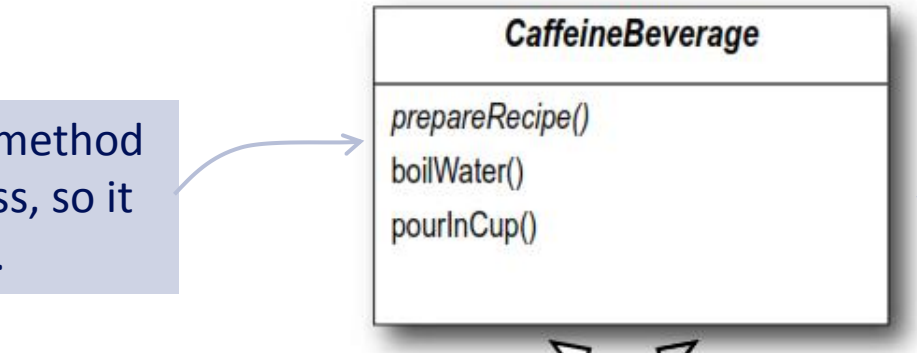
```
public class Tea {
    void prepareRecipe() {
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
    }

    public void boilWater() {
        System.out.println("Boiling water");
    }

    public void steepTeaBag() {
        System.out.println("Steeping the tea");
    }

    public void addLemon() {
        System.out.println("Adding Lemon");
    }

    public void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

This looks very similar to the one we just implemented in Coffee; the second and forth steps are different, but it's basically the same recipe.

These two methods are specialized to Tea.

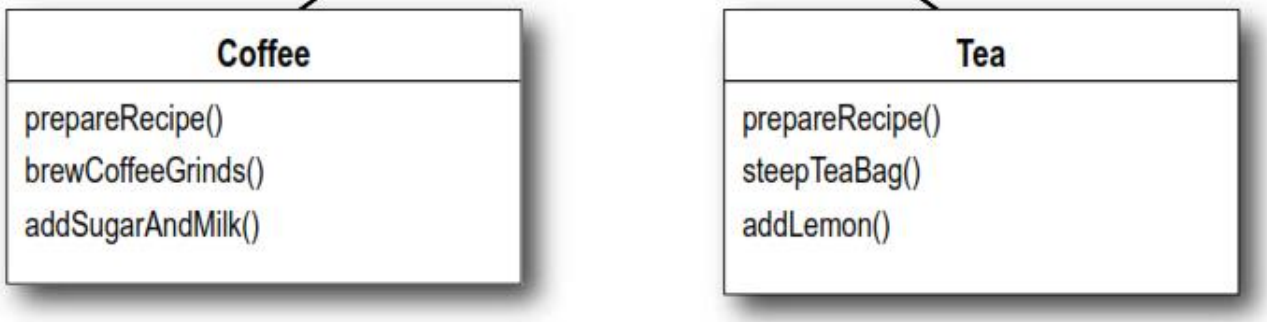
Notice that these two methods are exactly the same as they are in Coffee! So we definitely have some code duplication going on here.

abstract classes Coffee, Tea



The prepareRecipe() method differs in each subclass, so it is defined as abstract.

The boilWater() and pourInCup() methods are shared by both subclasses, so they are defined in the superclass.



Each subclass implements its own recipe.

Each subclass overrides the prepareRecipe() method and implements its own recipe.

The methods specific to Coffee and Tea stay in the subclasses.

- The structure of the algorithms in prepareRecipe() is similar for Tea and Coffee
  - We can improve our code further by making the code in prepareRecipe() more abstract
    - brewCoffeeGrinds() and steepTea() -> brew()
    - addSugarAndMilk() and addLemon() -> addCondiments()
- Now all we need to do is specify this structure in CaffeineBeverage.prepareRecipe() and make sure we do it in such a way so that subclasses can't change the structure
  - By using the word "final"

Tea class for making tea

Coffee.java

```
public abstract class CaffeineBeverage {
    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("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!

brew() and addCondiments() are abstract and must be supplied by subclasses

boilWater() and pourInCup() are specified and shared across all subclasses

Dealing with the Coffee and Tea classes

Coffee.java

```
public class Coffee extends CaffeineBeverage {
    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }

    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

Coffee.java

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

    public void addCondiments() {
        System.out.println("Adding Lemon");
    }
}
```

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.

- Took two separate classes with separate but similar algorithms
- Noticed duplication and eliminated it by adding a superclass
- Made steps of algorithm more abstract and specified its structure in the superclass
  - Thereby eliminating another “implicit” duplication between the two classes
- Revised subclasses to implement the abstract (unspecified) portions of the algorithm… in a way that made sense for them

Singleton Pattern

- Used to ensure that only one instance of a particular class ever gets created and that there is just one (global) way to gain access to that instance
- Let’s derive this pattern by starting with a class that has no restrictions on who can create it

Ball.java

```
public class Ball {
    private String color;

    public Ball(String color) {
        this.color = color;
    }

    public void bounce() {
        System.out.println("Boing!");
    }

    public static void main(String[] args) throws Exception {
        Ball b1 = new Ball("Red");
        Ball b2 = new Ball("Blue");
        b1.bounce();
        b2.bounce();
    }
}
```

As long as a client object “knows about” the name of class Ball, it can create instances of Ball

This is because the constructor is public

We can stop unauthorized creation of Ball instances by making the constructor private

Ball.java

```
public class Ball {
    private String color;

    private Ball(String color) {
        this.color = color;
    }

    public void bounce() {
        System.out.println("Boing!");
    }

    public static void main(String[] args) throws Exception {
        Ball b1 = new Ball("Red");
    }
}
```

Now

```
Ball b1 = new Ball("Red");
```

impossible by any method outside of Ball

Problem: No Point of Access!

- Now that the constructor is private, no class can gain access to instances of Ball
  - But our requirements were that there would be **at least one way** to get access to an instance of Ball
- We need a method to return an instance of Ball
  - But since there is no way to get access to an instance of Ball, the method **CANNOT** be an instance method
    - This means it needs to be a class method, aka a static method

Ball.java

```
public class Ball {
    private String color;

    private Ball(String color) {
        this.color = color;
    }

    public void bounce() {
        System.out.println("Boing!");
    }

    public static Ball getInstance(String color) {
        return new Ball(color);
    }

    public static void main(String[] args) throws Exception {
        // Ball b1 = new Ball("Red");
        Ball b1 = new Ball.getInstance("Blue");
    }
}
```

We are back to this problem where any client can create an instance of Ball;

instead of saying this:

```
Ball b1 = new Ball("Red");
```

they just say:

```
Ball b1 = new Ball.getInstance("Red");
```

Deriving Singleton (IV)

Ball.java

```
public class Ball {
    private static Ball ball;
    private String color;

    private Ball(String color) {
        this.color = color;
    }

    public void bounce() {
        System.out.println("Boing!");
    }

    public static Ball getInstance(String color) {
         return ball;
    }
```

To ensure only one instance is ever created

- Need a static variable to store that instance
  - No instance variables are available in static methods

Now the getInstance() method returns null each time it is called

- Need to check the static variable to see if it is null
  - If so, create an instance
- Otherwise return the single instance

Problem: First Parameter Wins

Ball.java

```
public class Ball {
    private static Ball ball;
    private String color;

    private Ball(String color) {
        this.color = color;
    }

    public void bounce() {
        System.out.println("Boing!");
    }

    public static Ball getInstance(String color) {
        if (ball == null) {
            ball = new Ball(color);
        }
        return ball;
    }
}
```

- The code shows the Singleton pattern
  - private constructor
  - private static variable to store the single instance
  - public static method to gain access to that instance
  - this method creates object if needed; returns it

But this code ignores the fact that a parameter is being passed in;

so if a “Red” ball is created all subsequent requests for a “Green” ball are ignored

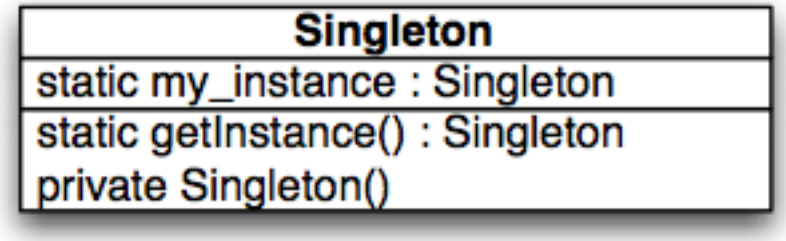
- The solution to the final problem is to change the private static instance variable to a Map

```
private static Map<String, Ball> ballRecord = new HashMap...
```

- Then check if the map contains an instance for a given value of the parameter
  - this ensures that only one ball of a given color is ever created
  - this is a very acceptable variation of the Singleton pattern

Ball.java

```
public static Ball getInstance(String color) {
    if (!ballRecord.containsKey(color)) {
        ballRecord.put(color, new Ball(color));
    }

    return ballRecord.get(color);
}
```



- Singleton involves only a single class (not typically called Singleton). That class is a full-fledged class with other attributes and methods (not shown).
- The class has a static variable that points at a single instance of the class.
- The class has a private constructor (to prevent other code from instantiating the class) and a static method that provides access to the single instance.

Singleton Pattern: Real World Examples

- Centralized manager of resources
  - Window manager
  - File system manager
  - …
- Logger classes
- Factories
  - Especially those that issue IDs
  - Singleton is often combined with Factory Method and Abstract Factory patterns

- Patterns
  - http://en.wikipedia.org/wiki/Software_pattern
  - https://sourcemaking.com/design_patterns
  - http://hillside.net/patterns/patterns-catalog
  - http://c2.com/ppr/

- Interaction design patterns
  - http://en.wikipedia.org/wiki/Interaction_design_pattern

- Anti-Patterns
  - http://en.wikipedia.org/wiki/Anti-pattern#Programming_anti-patterns

Thank you!