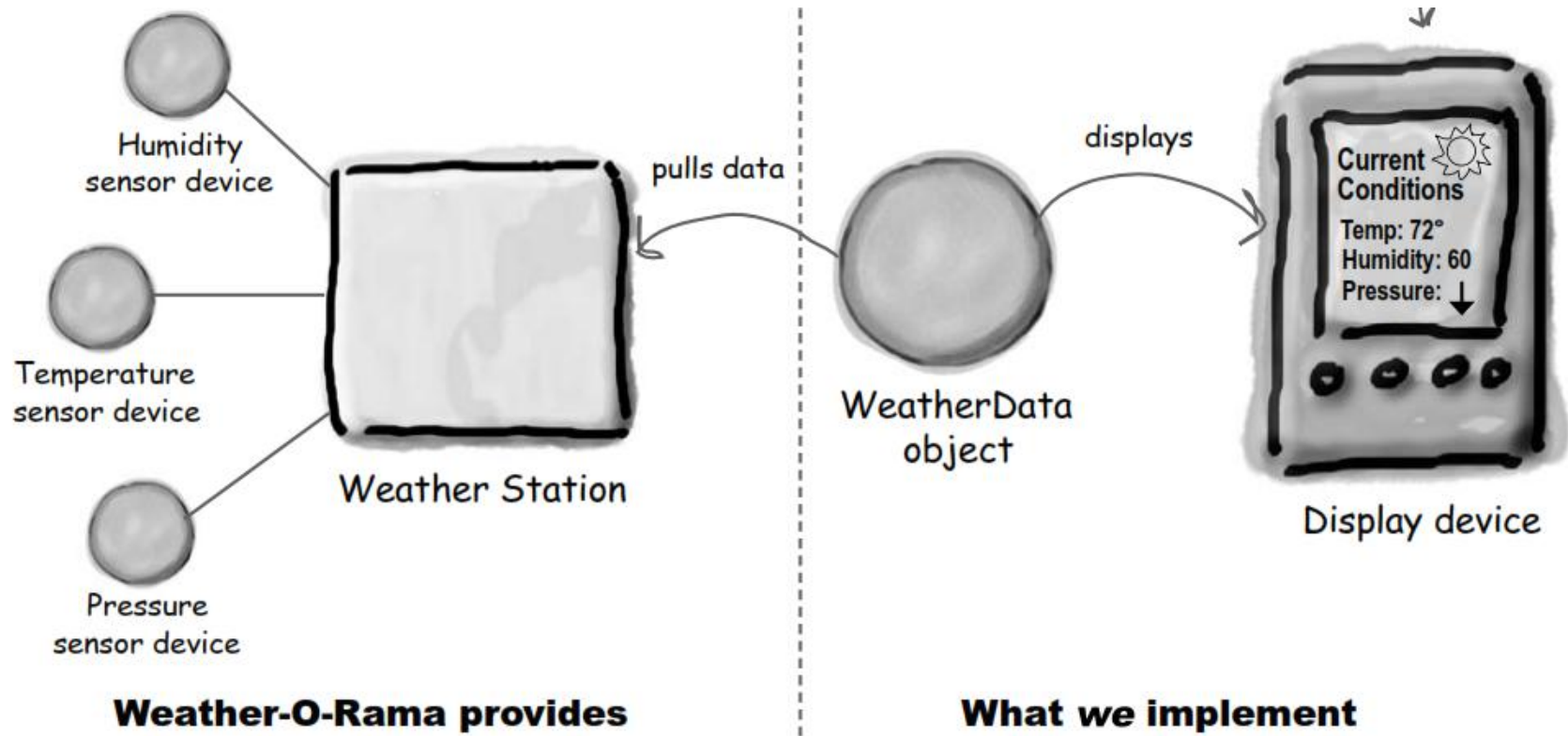# DESIGN PATTERNS II

Yuqun Zhang
CS304

Figures from Head First Design Patterns
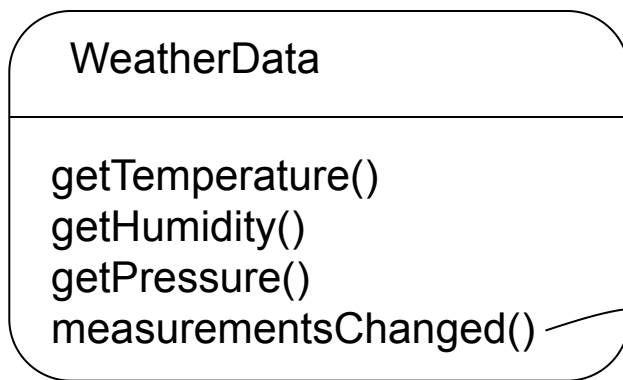
THE OBSERVER PATTERN

A Weather Monitoring Application

Create an app that uses the WeatherData object to update three different displays:
- "current conditions"
- "weather stats"
- "forecast"

What Needs to be Done?

WeatherData

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

Current Conditions
Temp: 72°
Humidity: 60
Pressure: ↓

Update three
different displays

```
/*
* Call this method
* whenever measurements are
* Updated
*/
public void measurementsChanged(){
  // your code goes here
}
```

Problem Specification

- The `WeatherData` class has getters and setters for temperature, humidity, and pressure
- The `measurementsChanged()` method is called anytime new weather data is available
  - We don't know *or care* how.
- We need to implement three different display elements that use the weather data
- The system must be expandable, in case others want to add other display elements later

A First Try

```
public class WeatherData {

    //instance variable declarations

    public void measurementsChanged(){
                        mperature();
                        etHumidity();
                        etPressure();

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

Coding to implementations: adding displays requires changing the program

Not so bad: here's a common interface!

Encapsulate stuff that changes!
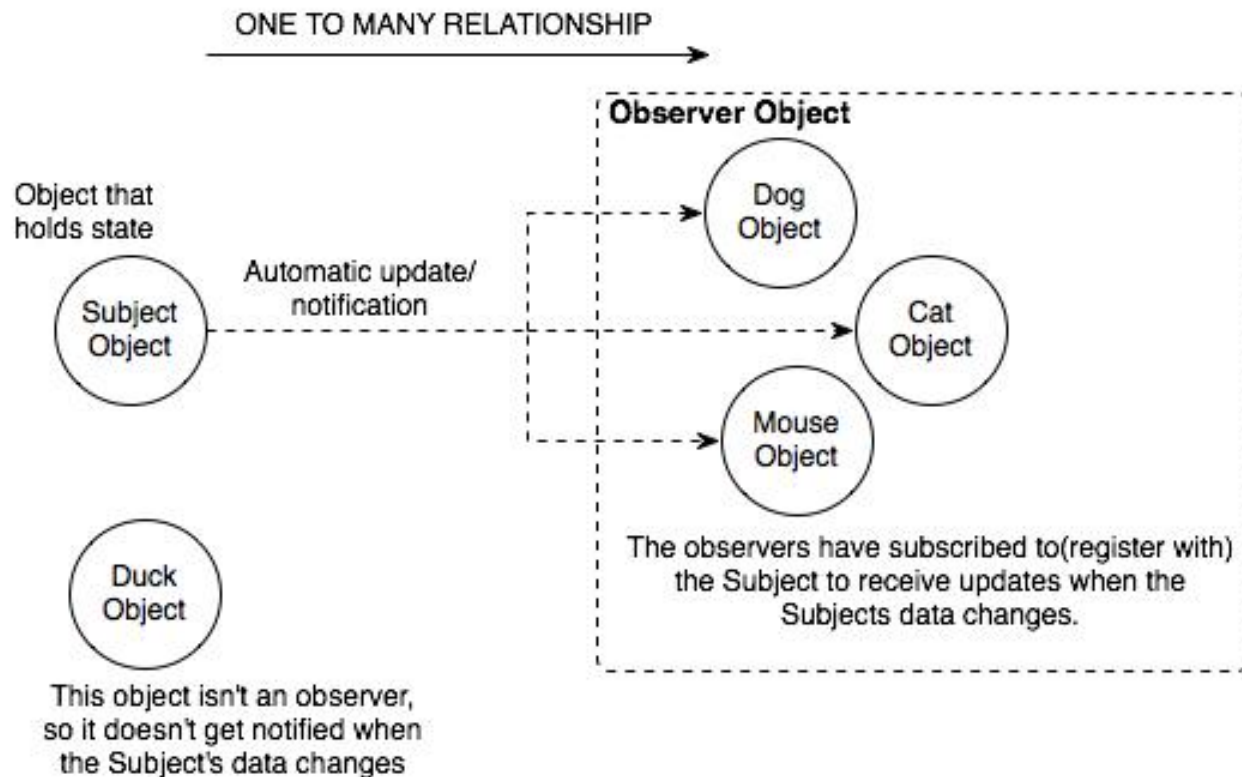
What's Wrong?

Publish/Subscribe

- Just like newspapers and magazines
  - You subscribe and receive any new additions
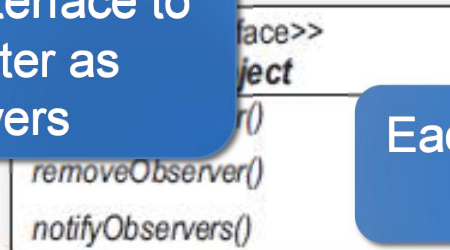 - You unsubscribe and stop receiving anything

The Observer Pattern

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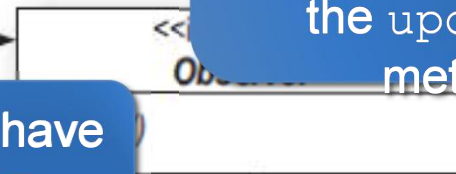
The Observer Pattern

Objects use the `Subject` **interface to (de)register as observers**

All potential observers need to implement the `Observer` **interface and provide the** `update()` **method**

Each subject can have many observers

A concrete subject always implements the `Subject` **interface and the** `notifyObservers()` **method**
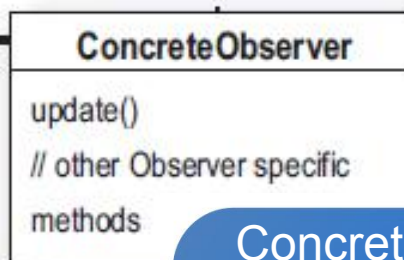
Concrete observers can be any class that implements the `Observer` **interface and registers with a concrete subject**

Diagram elements:

```
<<interface>>
Subject
...r()
removeObserver()
notifyObservers()
```

```
observers
```

```
<<interface>>
Observer
...()
```

```
ConcreteSubject
registerObserver() {...}
removeObserver() {...}
notifyObservers() {...}
```

```
subject
```

```
ConcreteObserver
update()
// other Observer specific
methods
```

The Power of Loose Coupling

- The only thing a subject knows about an observer is that it implements a given interface

- We can add new observers at any time

- We never need to modify the subject to add new types of observers

- We can reuse subjects or observers independently of each other

- Cha ... affect eac ...

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

Exercise

- Draw the class diagram for the weather data app

Weather Data Interfaces

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

> These first two methods take an `Observer` as an argument

> This method is called to notify all observers when the `Subject`'s state has changed

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

> The `Observer` interface is implemented by all observers, giving them the `update()` method

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

> We added in a `DisplayElement` interface since all of the display types share the need to `display()`

Implementing the Subject Interface

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

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

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

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

This `ArrayList` holds our observers, and we'll have to maintain it...

These methods were required by the `Subject` interface.

Notify Methods

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

This one was required by the `Subject` interface, too.

```
public void measurementsChanged() {
  notifyObservers();
}
```

We notify the observers when we get updated measurements fro the weather station

```
public void setMeasure                    float
  humidity, float press
    this.temperature = tempreature;
    this.humidity = humidity;
    this.pressure = pressure;
    meassurementChanged();
}
```

A Display Element

This display element is an `Observer` so it can get changes from the `WeatherData` object

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

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

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

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

The constructor is passed the `Subject`, and we use it to register as an observer

When `update()` is called, we save the measurements and call `display()`

Client Test

- public class WeatherData {
-     public static void main (String[] args) {
-        WeatherData weatherdate = new WeatherData();

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

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

The Observer Pattern in Java

- Java provides the `Observer` interface and the `Observable` class in the package `java.util`
  - Similar to `Subject` and `Observer`
- Enable both push and pull style interactions (as opposed to only push as before)

Another Design…

The Observable class keeps track of all your observers and notifies them for you.

This should look familiar. In fact, it's exactly the same as our previous class diagram!

We left out the DisplayElement interface, but all the displays still implement it too.

Observable is a CLASS not an interface, so WeatherData extends Observable.

Observable
addObserver()
deleteObserver()
notifyObservers()
setChanged()

observers

<<interface>>
Observer
update()

WeatherData
getTemperature()
getHumidity()
getPressure()

subject

GeneralDisplay
update()
display()

StatisticsDisplay
update()
display()

ForecastDisplay
update()
display()

this doesn't look familiar! Hold tight, we'll get to this in a sec…

Here's our Subject, which we can now also call the Observable. We don't need the register(), remove() and notifyObservers() methods anymore; we inherit that behavior from the superclass.
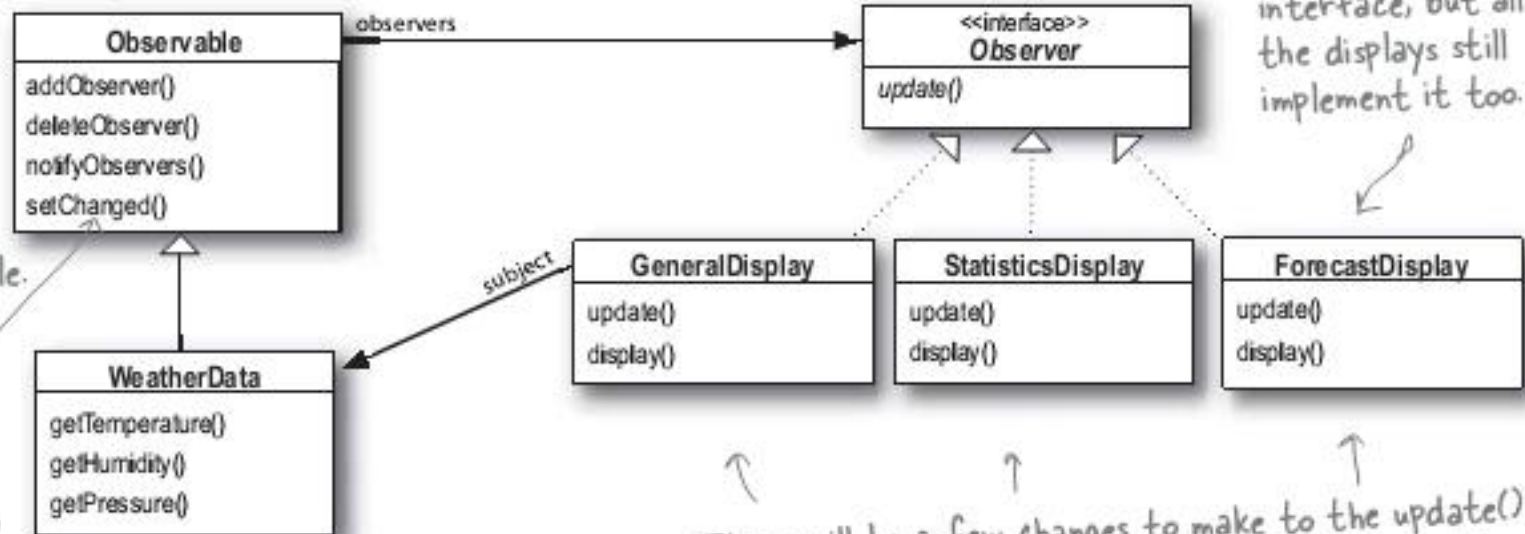
There will be a few changes to make to the update() method in the concrete Observers, but basically it's the same idea… we have a common Observer interface, with an update() method that's called by the Subject.

The Java Observer Pattern

- For an object to become an observer
  - Just implement the `Observer` interface (as before)
- For the observable to send notifications
  - Become observable by *extending* the `java.util.Observable` superclass
  - Call the `setChanged()` method to signify that the state of the object has changed
  - Call one of two notification methods:
    - `notifyObservers()`
    - `notifyObservers(Object arg)`

Notification Revisited

- For an observer to receive notifications
  - Provides a definition of the update method:

    ```
    update(Observable o, Object arg)
    ```

    The subject that sent the notification

    The data object passed through `notifyObservers(arg)` (or `null`)

- To *push* data
  - Pass the data as a data object through the `notifyObservers(arg)` method

- To have the Observer *pull* data
  - The Observer must use the `Observable` object passed to it using the object's getters and setters

How to pull

- import java.util.Observable;
- import java.util.Observer;

- public class WeatherData extends Observable{
- ...;
- public void measurementChanged(){
- setChanged();
- notifyObservers();
- }
- ...;
- public float getTempeature(){
- return temperature;
- }
- ...;
- }

how to pull (pseudo)

- setChanged(){
- changed = true
- }

- notifyObservers(Object arg){
- if (changed){
- for every obsrver on the list {
- call update (this, arg)
- }
- changed = false
- }

- notifyObservers(){
- ...;
- }

Rebuilding CurrentConditionsDisplay

```java
public class CurrentConditionsDisplay implements Observer, DisplayElement{
    Observable observable;
    ...;
    public CurrentConditionsDisplay(Observable observable){
        this.observable = observable;
        obserable.addObservable(this);
    }

    public void update (Observable obs, Object arg){
        if (obs instanceof WeahterData){
            WeatherData weatherData = (WeatherData) obs;
            this.temperature = weatherData.getTemperature();
            this.humidity = weatherData.getHumidity();
            display();
        }
    ...;
}
```

The Dark Side of Java Observables

- Observable is a class, not an interface
  - You have to *subclass* it, so you can't add the Observable behavior onto a class that already extends something else
    - Limits reuse potential
  - Because there's no Observable interface, you cannot create your own implementations of Observables
- Observable protects crucial methods
  - E.g., setChanged() can only be called by subclasses
    - Limits flexibilty; you cannot favor composition over inheritance

Observers are Everywhere

- Especially in Java Swing
  - E.g., JButton and associated listeners
- Etc.

THE FACTORY PATTERN

Creating Objects

- …it's more than just `new`

- A key tenet
  - Constructor usages often lead to unintended *coupling*

- Remember the note in the Strategy pattern?
  - When we say "`new`" to create a new object by calling a constructor, we're directly programming to an implementation
  - E.g., `Duck duck = new MallardDuck()`

We really WANT to use the interface…

But we're forced to create an instance of a concrete class!

It's More Complicated than That

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

- … especially when you think about the fact that things might change
  - E.g., you add a new type of duck and have to figure out when/how to instantiate it
  - And you make new kinds of Ducks in all different parts of your code

"Open for Extension, Closed for Modification"

- A key design goal:
  - Allow classes to be easily extended to incorporate new behavior
  - Without modifying existing code
    - Because everytime you modify it, you risk introducing new bugs
- This results in designs that are resilient to change but also flexible enough to accept new functionality to meet changing requirements

Back to Identifying Things that Change

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

It'd really be nice to use an abstract class here, but, alas, one cannot instantiate an abstract class

These methods can now be specific to a particular type of pizza (i.e., the pizza type knows how to prepare itself) or generic to all pizza types

Change… it's Coming

- The pizza business is a trendy one
  - Greek pizza is so yesterday…
  - But with all of the people moving in from CA, we've got increasing demands for Veggie pizza. And some weirdos who want clam pizza
- What to do?

- The `orderPizza` method is not **closed for modification**

- What varies? What stays the same?
  - The choices of pizza types change over time
  - The process (algorithm) for filling an order stays the same

Information Hiding?

- What is it we're supposed to do with the stuff that changes?
- Encapsulate it!
- Practically, since the thing that's changing is **object creation**, we need an object that encapsulates object creation
- This object is called a **factory**
  - Then the `orderPizza` method is a **client** of the factory
    - Anytime it needs a pizza, it goes to the factory to request that one is created

The Pizza Factory

```java
public class SimplePizzaFactory {
  public Pizza createPizza(String type) {
    Pizza pizza = null;
    if (type.equals("cheese")) {
      pizza = new CheesePizza();
    } else if (type.equals("pepperoni") {
      pizza = new PepperoniPizza();
    } else if (type.equals("clam") {
      pizza = new ClamPizza();
    } else if (type.equals("veggie") {
      pizza = new VeggiePizza();
    }
    return pizza;
  }
}
```

Wait, what?

- Well, that seems silly. All we did is copy the code out of the `orderPizza` method, but it still has all of the same problems…

- Does it?

- The `SimplePizzaFactory` might have lots of clients (not just the `orderPizza` method)
  - That was why we want to encapsulate the thing that changes!

- Also, the `orderPizza` method no longer needs to know anything at all about concrete `Pizza`s!

Rebuilding PizzaStore

```
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;
    }
...;
}
```

Simple Factory: Not Quite a Pattern

- But it is a **programming idiom**, and it's commonly used

This is the **factory** where we create pizzas; it should be the only part of our application that refers to concrete Pizza classes.

This is the **product** of the factory: pizza!

We've defined Pizza as an abstract class with some helpful implementations that can be overridden.

This is the **client** of the factory, PizzaStore now goes through the SimplePizzaFactory to get instances of pizza.

The create method is often declared statically.

These are our **concrete products**. Each product needs to implement the Pizza interface (which in this case means "extend the abstract Pizza class") and be concrete. As long as that's the case, it can be created by the factory and handed back to the client.

PizzaStore
orderPizza()

SimplePizzaFactory
createPizza()

Pizza
prepare()
bake()
cut()
box()

CheesePizza

VeggiePizza

PepperoniPizza

ClamPizza

Franchising the Pizza Store

- Now you want to spread your successful business
  - We want to localize the pizza making activities to the `PizzaStore` class
    - For quality control
  - But we want to give regional franchises the liberty to have their own pizza styles
- General framework:
  - Make the `PizzaStore` abstract
  - Put the `createPizza` method back in `PizzaStore`, but make it abstract
  - Create a `PizzaStore` subclass for every regional type of pizza

The Abstract Method

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

This is the "factory method"

Delegating to the Subclasses

- We've perfected the pizza ordering method, and it stays the same across all of the subclasses
- But now the regional franchises can differ in the style of pizza they make
  - E.g., thin crust in New York, thick crust in Chicago
- While the `orderPizza` method *looks* like it's defined in the `PizzaStore` class, this class is abstract
  - It can't *actually* do anything
  - So when it is executed, it is actually executing in the context of a concrete subclass
  - This context gets determined when the (abstract) method `createPizza` gets called

Delegating to the Subclasses (cont.)

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.

PizzaStore

createPizza()
orderPizza()

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.

NYStylePizzaStore

createPizza()

ChicagoStylePizzaStore

createPizza()

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

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

```
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();
    }
}
```

What's a Franchise Look Like?

- Bonus. The franchises get all of the benefits of the perfected PizzaStore ordering process
- All they have to do is define how to create pizzas!

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

A Generic Factory Method

```
abstract Product factoryMethod(String type)
```

A factory method is abstract so the subclasses are counted on to handle object creation

The factory method isolates the client (the code in the superclass) from knowing what kind of concrete `Product` is created

A factory method returns a `Product` that is typically used within methods defined in the superclass

A factory method may be parameterized (or not) to select among several variations of a `Product`

Ordering a Pizza

- What happens when a customer wants to order a pizza?
  - Write down the sequence of steps involved in a customer ordering a NY style (thin crust) cheese pizza.

Solution

1. First, the customer needs to get a NY PizzaStore:
   - `PizzaStore nyPizzaStore = new NYPizzaStore();`

2. Now the pizza store can accept our order
   - `nyPizzaStore.orderPizza("cheese");`

3. The `orderPizza` method calls the `createPizza` method
   - `Pizza pizza = createPizza("cheese");`
     - Remember the `createPizza` method is implemented in the subclass, so we're automagically getting a NY style cheese pizza here

4. The `orderPizza` method finishes preparing our pizza
   - `pizza.prepare();pizza.bake();pizza.cut();pizza.box();`
     - These methods are defined in the abstract `PizzaStore` class, which doesn't need to know which kind of pizza it is in order to follow the steps

The Entire Solution

• The whole thing requires some pizzas to tie everything together; you can check out the sample source code to see how it all fits
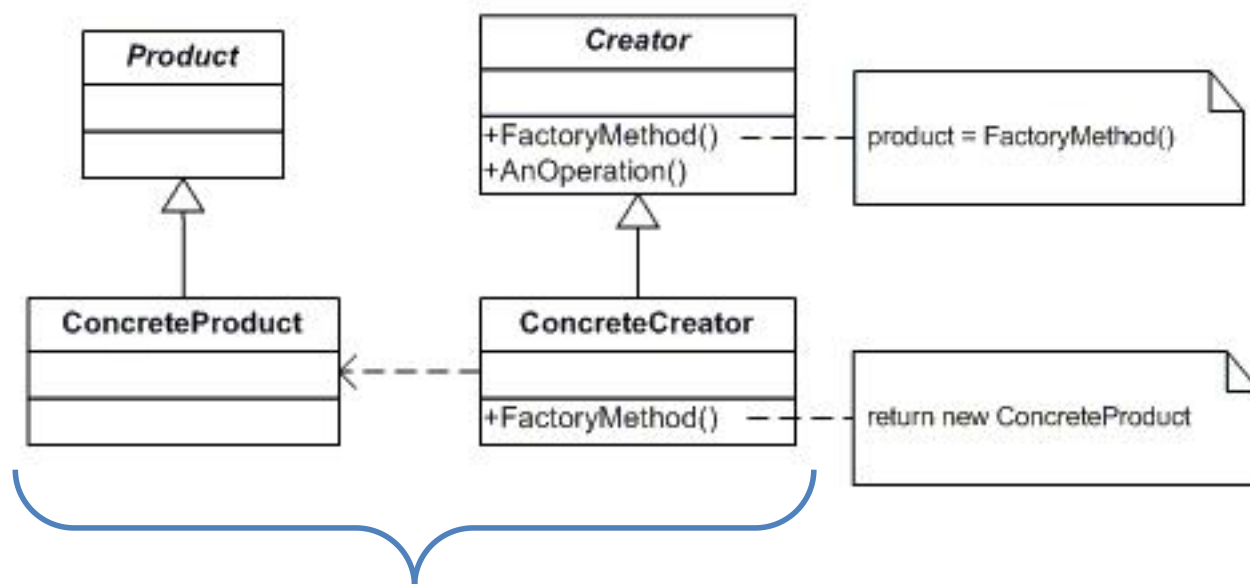
Parallel Class Hierarchies

The Factory Method Pattern

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
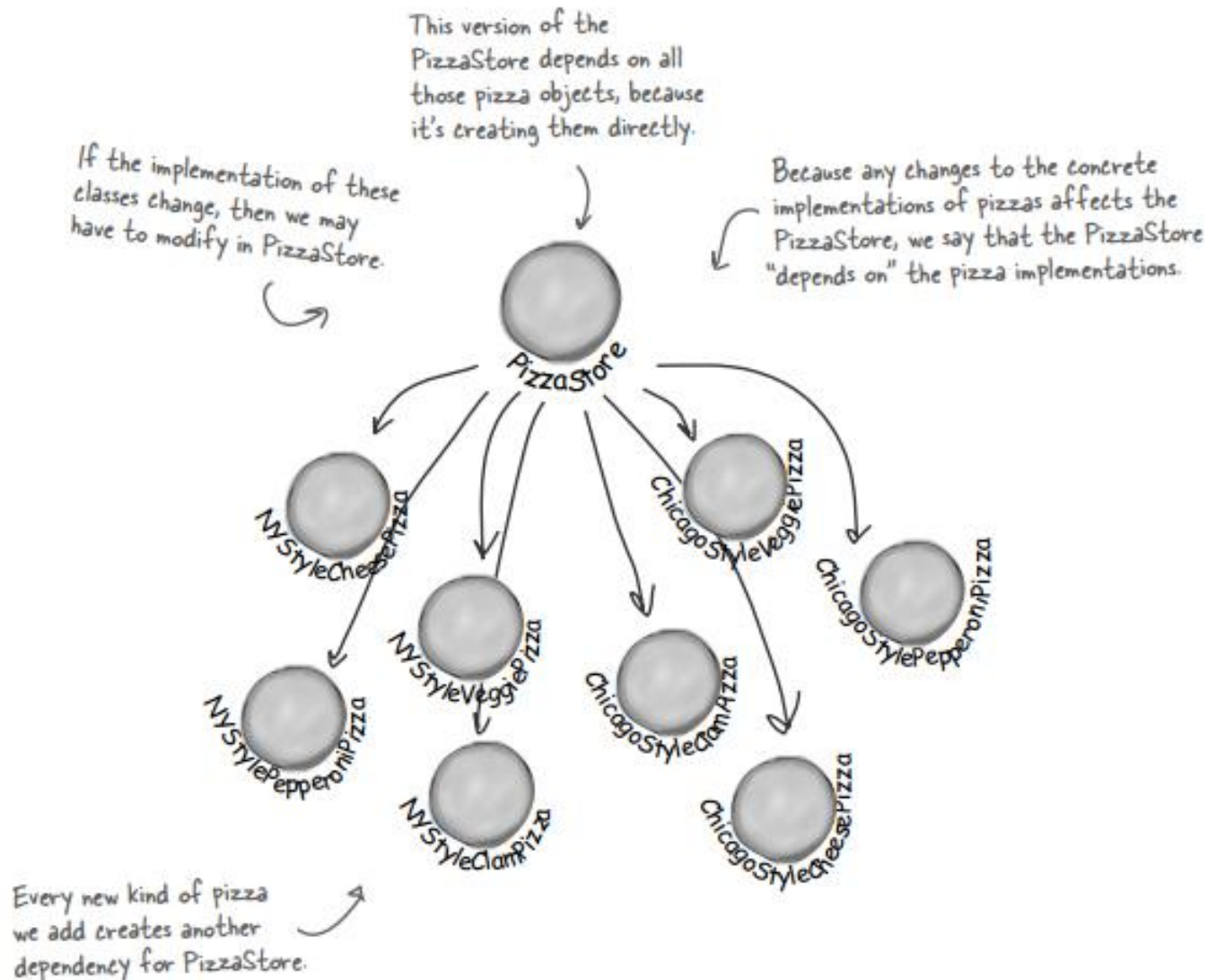
Our "Dumb" Pizza Store Revisited

- Imagine going back to the beginning and creating a PizzaStore that amassed all of the decision making
  - Inside the createPizza method of this pizza store, I would just have a huge, nested set of if statements to determine which *style* of pizza and then which *type* of pizza to create

The intuitive way of design

```java
public class DependentPizzaStore{
    public Pizza createPizza (String style, String type){
        Pizza pizza = null;
        if (style.euqals("NY")){
            if (type.equals("cheese")){
                pizza = new NYStyleCheesePizza();
            } else if (type.equals("veggie")){
                pizza = new NYStyleVeggiePizza();
            } else if (type.equals("clam")){
                pizza = new NYStyleClamPizza();
            } else if (type.equals("pepperoni")){
                pizza = new NYStylePepperoniPizza();
            }
        } else if (style.equals("Chicago")){
            ...;
        }
        pizza.prepare();
        pizza.bake();
        ...;
        return pizza;
    }
}
```

What does this Look Like?

This version of the PizzaStore depends on all those pizza objects, because it's creating them directly.

If the implementation of these classes change, then we may have to modify in PizzaStore.

Because any changes to the concrete implementations of pizzas affects the PizzaStore, we say that the PizzaStore "depends on" the pizza implementations.

PizzaStore

NYStyleCheesePizza

ChicagoStyleVeggiePizza

NYStyleVeggiePizza

ChicagoStylePepperoniPizza

NYStylePepperoniPizza

ChicagoStyleClamPizza

NYStyleClamPizza

ChicagoStyleCheesePizza

Every new kind of pizza we add creates another dependency for PizzaStore.
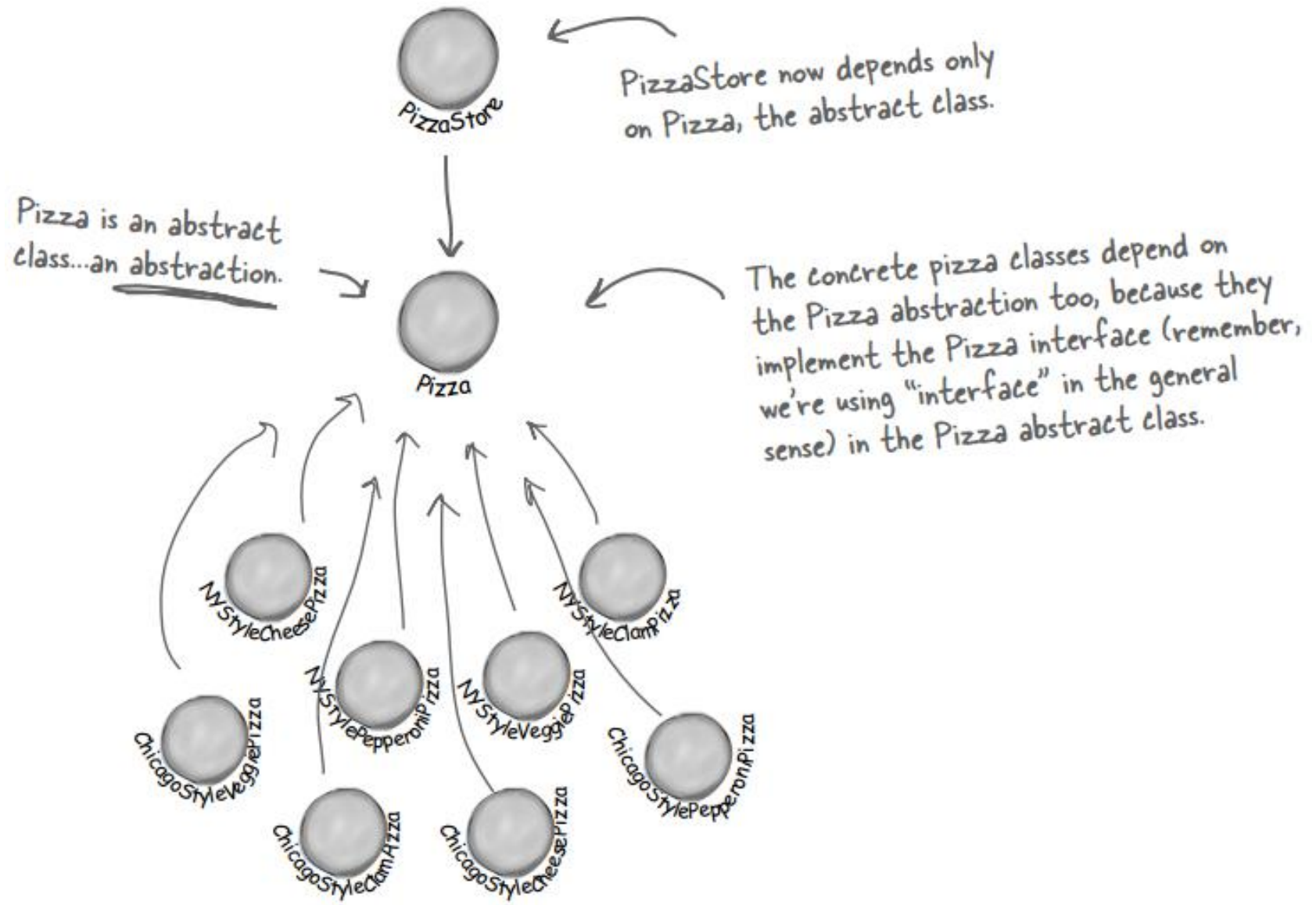
Another Design Principle

- This seems like a bad idea. We're definitely not encapsulating for change.

- If we change any of the concrete pizza classes, we have to change the PizzaStore because it **depends** on them

- Instead we should **depend upon abstractions. Do not depend upon concrete classes**

  - High level components should not depend on low-level components; instead, both should depend on abstractions

  - For example, in the previous pizza store, the store depended on all of the pizza types

  - Instead, the pizza store should depend on the abstract notion of Pizza, and the concrete pizza types should too

    - This is exactly what the Factory Method pattern we applied did!

Dependency Inversion

PizzaStore now depends only on Pizza, the abstract class.

Pizza is an abstract class...an abstraction.

The concrete pizza classes depend on the Pizza abstraction too, because they implement the Pizza interface (remember, we're using "interface" in the general sense) in the Pizza abstract class.

Guidelines that Help

- **NOT RULES TO FOLLOW**
- No variable should hold a reference to a concrete class
  - If you use new, you'll be holding a reference to a concrete class
  - Use a factory to get around that!
- No class should derive from a concrete class
  - If you do, you're depending on the concrete class
  - Instead, derive from an abstraction (like an interface or an abstract class)
- No method should override an implemented method of any of its base classes
  - If you do, then your base class wasn't really an abstraction
  - The methods implemented in the base class are meant to be shared by the derived classes

THE ABSTRACT FACTORY

Controlling Pizza Quality

- Some of your franchises have gone rogue and are substituting inferior ingredients to increase their per-pizza profit

- Time to enter the pizza ingredient business
  - You'll make all the ingredients yourself and ship them to your franchises
  - But this is not so easy…

- You have the same product families (e.g., dough, sauce, cheese, veggies, meats, etc.) but different implementations (e.g., thin vs. thick or mozzarella vs. reggiano) based on region

The Ingredient Factory Interface

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

Then What?

1. For each region, create a subclass of the `PizzaIngredientFactory` that implements the concrete methods

2. Implement a set of ingredients to be used with the factory (e.g., `ReggianoCheese`, `RedPeppers`, `ThickCrustDough`)

    - These can be shared among regions if appropriate

3. Integrate these new ingredient factories into the `PizzaStore` code

The New York Ingredient Factory

```java
public class NYPizzaIngredientFactory implements 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();
  }
}
```

Connecting to the Pizzas

- Now, we need to force our franchise owners to only use factory produced ingredients
- Before, the abstract `Pizza` class just had `String`s to name its ingredients
  - It implemented the `prepare()` method (and `bake()`, `cut()`, and `box()`)
  - The concrete `Pizza` classes just defined the constructor which, in some cases, specialized the ingredients (and sometimes cut corners) and maybe overwrote other methods
- Now, the abstract `Pizza` class has actual ingredient objects
  - And the `prepare()` method is abstract
  - The concrete pizza classes will collect the ingredients from the factories to prepare the pizza

Concrete Pizzas

- Now, we only need one CheesePizza class (before we had a ChicagoCheesePizza and a NYCheesePizza)

- When we create a CheesePizza, we pass it an IngredientFactory, which will provide the (regional) ingredients

An Example Pizza

```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();
  }
}
```

Which cheese is created is determined at run time by the factory passed at object creation time

Fixing the Pizza Stores

```
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");
    } // more of the same…
    return pizza;
  }
}
```
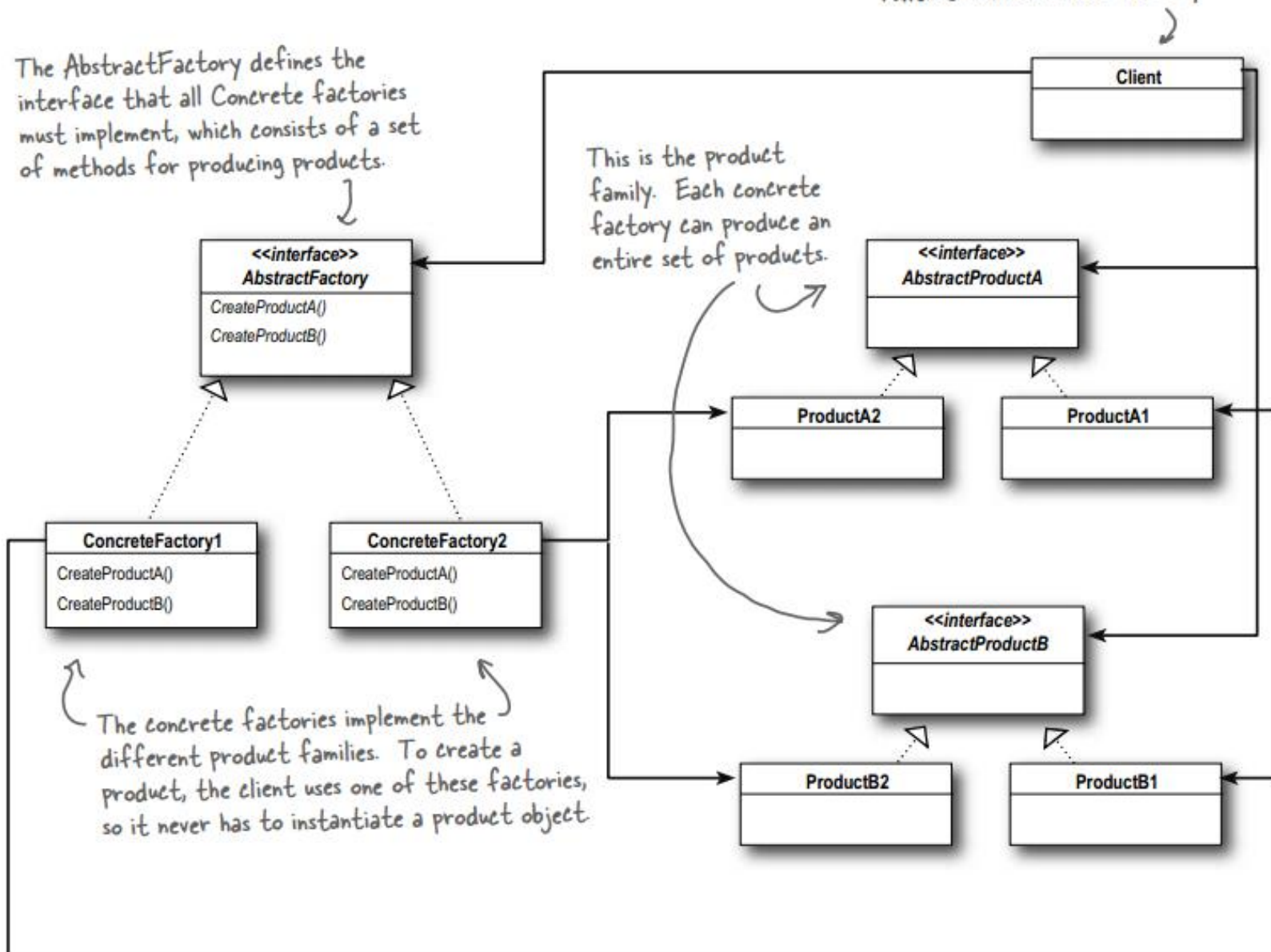
For each type of pizza, we instantiate a new pizza and give it the factory it needs to get its ingredients

Whew. Recap.

- We provided a means of creating a family of ingredients for pizzas by introducing a new type of factory: the **abstract factory**

- An abstract factory provides an interface for creating a family of products
  - Decouples code from the actual factory that creates the products
  - Makes it easy to implement a variety of factories that produce products for different contexts (we used regions, but it could just as easily be different operating systems, or different "look and feels")

- We can substitute different factories to get different behaviors

The Client is written against the abstract factory and then composed at runtime with an actual factory.

The AbstractFactory defines the interface that all Concrete factories must implement, which consists of a set of methods for producing products.

This is the product family. Each concrete factory can produce an entire set of products.

Client

<<interface>>
AbstractFactory

CreateProductA()
CreateProductB()

<<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

The Abstract Factory Pattern

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

Factory Method vs. Abstract Factory

- Decouples applications from specific implementations

- Creates objects through inheritance
  - Create objects by extending a class and overriding a factory method

- Useful if you don't know ahead of time what concrete classes will be needed

- Decouples applications from specific implementations

- Creates objects through object composition
  - Create objects by providing an abstract type for a family of products
  - Subclasses define how products are produced

- Interface must change if new products are added

QUESTIONS?