Tecniche di Progettazione: Design Patterns

GoF: Factory Method e Abstract Factory

Factory Patterns

- Factory: a class whose sole job is to easily create and return instances of other classes
- *Creational patterns abstract the object instantiation process.*
  - They hide how objects are created and help make the overall system independent of how its objects are created and composed.
  - They make it easier to construct complex objects instead of calling a constructor, use a method in a "factory" class to set up the object saves lines and complexity to quickly construct / initialize objects
- examples in Java:
  - borders (BorderFactory),
  - key strokes (KeyStroke),
  - network connections (SocketFactory)

Factory Patterns

- *Class creational patterns focus on the use of inheritance to decide* the object to be instantiated
  - Factory Method

- *Object creational patterns focus on the delegation of the* instantiation to another object
  - Abstract Factory

The Problem With "New"

- 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's type is set to an "interface", in this case "Duck", the class that contains this statement depends on "DecoyDuck"

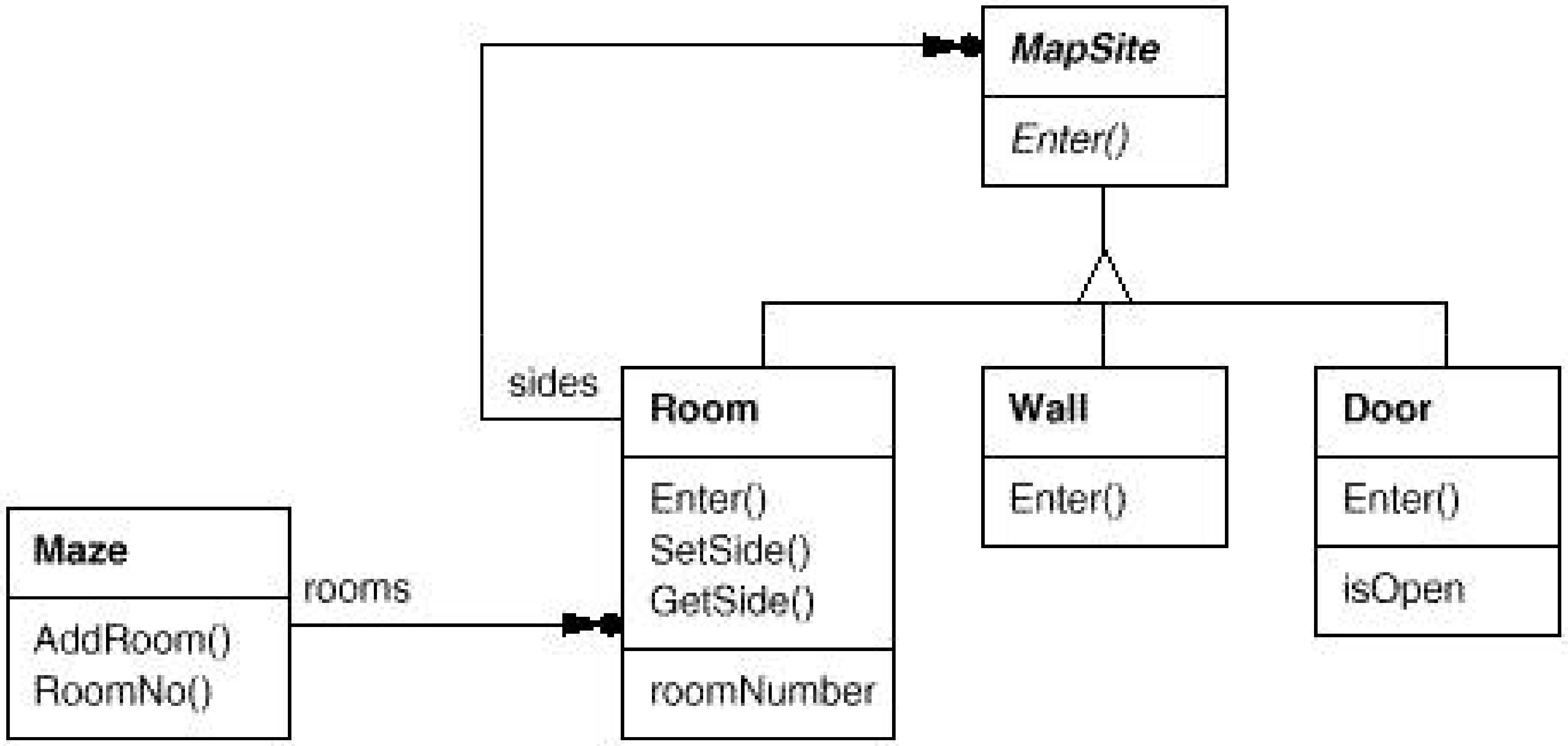
In addition

- if you have code that checks a few variables and instantiates a particular type of class based on the state of those variables, then the containing class depends on each referenced concrete class

```
    if (hunting) { return new DecoyDuck(); }             //decoy=da richiamo
       else { return new RubberDuck();}
```

- Obvious Problems: needs to be recompiled if classes change
  - 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

Example: Maze



Here's a MazeGame class with a createMaze() method

```
/**
* MazeGame.
*/
public class MazeGame {
// Create the maze.
    public Maze createMaze() {
            Maze maze = new Maze();
            Room r1 = new Room(1);
            Room r2 = new Room(2);
            Door door = new Door(r1, r2);
            maze.addRoom(r1);
            maze.addRoom(r2);
            r1.setSide(MazeGame.North, new Wall());
            r1.setSide(MazeGame.East, door);
            r1.setSide(MazeGame.South, new Wall());
            r1.setSide(MazeGame.West, new Wall());
            r2.setSide(MazeGame.North, new Wall());
            r2.setSide(MazeGame.East, new Wall());
            r2.setSide(MazeGame.South, new Wall());
            r2.setSide(MazeGame.West, door);
            return maze;
    }
}
```

The problem with this createMaze() method is its *inflexibility*.

- What if we wanted to have enchanted mazes with EnchantedRooms and EnchantedDoors? Or a secret agent maze with DoorWithLock and WallWithHiddenDoor?

- What would we have to do with the createMaze() method? As it stands now, we would have to make significant changes to it because of the explicit instantiations using the *new operator of the* objects that make up the maze. How can we redesign things to make it easier for createMaze() to be able to create mazes with new types of objects?

Let's add factory methods to the MazeGame class

```
/**
* MazeGame with a factory methods.
*/
public class MazeGame {
    public Maze makeMaze() {return new Maze();}
    public Room makeRoom(int n) {return new Room(n);}
    public Wall makeWall() {return new Wall();}
    public Door makeDoor(Room r1, Room r2)  {return new Door(r1, r2);}
```

Abstract or concrete

```
public Maze createMaze() {
    Maze maze = makeMaze();
    Room r1 = makeRoom(1);
    Room r2 = makeRoom(2);
    Door door = makeDoor(r1, r2);
    maze.addRoom(r1);
    maze.addRoom(r2);
    r1.setSide(MazeGame.North, makeWall());
    r1.setSide(MazeGame.East, door);
    r1.setSide(MazeGame.South, makeWall());
    r1.setSide(MazeGame.West, makeWall());
    r2.setSide(MazeGame.North, makeWall());
    r2.setSide(MazeGame.East, makeWall());
    r2.setSide(MazeGame.South, makeWall());
    r2.setSide(MazeGame.West, door);
    return maze;
  }
}
```

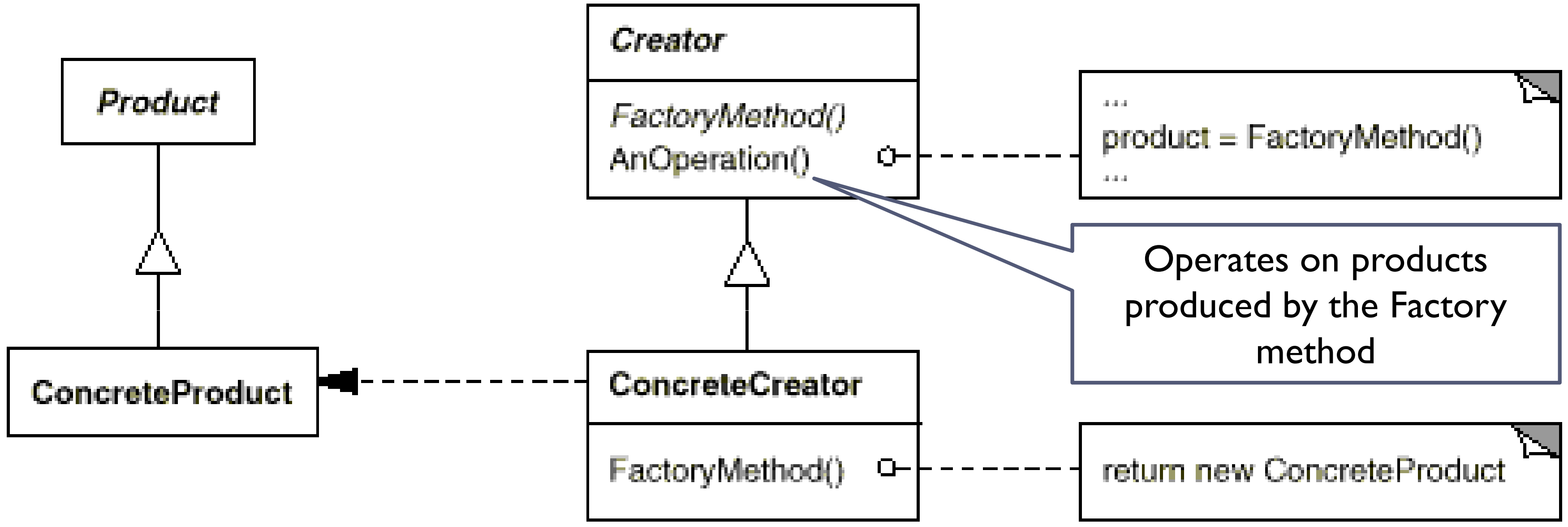
We made createMaze() just slightly more complex, but a lot more flexible!

- Consider this EnchantedMazeGame class:

```
public class EnchantedMazeGame extends MazeGame {
    public Room makeRoom(int n) {return new EnchantedRoom(n);}
    public Wall makeWall() {return new EnchantedWall();}
    public Door makeDoor(Room r1, Room r2){return new EnchantedDoor(r1, r2);}
}
```

- The createMaze() method of MazeGame is inherited by EnchantedMazeGame
  - It can be used to create regular mazes or enchanted mazes *without modification!*

The Factory Method Pattern


Creator
FactoryMethod()
AnOperation()
...
product = FactoryMethod()
...
Product
Operates on products produced by the Factory method
ConcreteProduct
ConcreteCreator
FactoryMethod()
return new ConcreteProduct

In the official definition:
Factory method lets the subclasses ***decide*** which class to instantiate
Decide: --not because the classes themselves decide at runtime
-- but because the creator is written withount knowlwdge of the actual products that will be created, which is decided by the choice of the subclass that is usd

The Factory Method Pattern: Participants

- Product
  - Defines the interface for the type of objects the factory method creates
- ConcreteProduct
  - Implements the Product interface
- Creator
  - Declares the factory method, which returns an object of type Product
- ConcreteCreator
  - Overrides the factory method to return an instance of a ConcreteProduct

Factory Method pattern at work: Maze

- The reason this works is that the createMaze() method of MazeGame defers the creation of maze objects to its subclasses.
- In this example, the correlations are:
  - Creator => MazeGame
  - ConcreteCreator => EnchantedMazeGame

    (MazeGame is also a ConcreteCreator)
  - Product => MapSite
  - ConcreteProduct => Wall, Room, Door, EnchantedWall, EnchantedRoom, EnchantedDoor
  - Maze is a concrete Product (but also Product)

The Factory Method Pattern

- Applicability
  - Use the Factory Method pattern in any of the following situations:
    - A class can't anticipate the class of objects it must create
    - A class wants its subclasses to specify the objects it creates

Example3: Pizza

- Consider a pizza store that makes different types of pizzas

```
public class PizzaStore {

Pizza orderPizza(String type){

  Pizza pizza;

  If (type == CHEESE)
                pizza = new CheesePizza();
  else if (type == PEPPERONI)
                pizza = new PepperoniPizza();
  else if (type == PESTO)
                pizza = new PestoPizza();
  pizza.prepare();
  pizza.bake();
  pizza.package();
  pizza.deliver();
  return pizza
  }
}
```

This becomes unwieldy as we add to our menu

This part stays the same

Idea: pull out the creation code and put it into an object that only deals with creating pizzas - the PizzaFactory

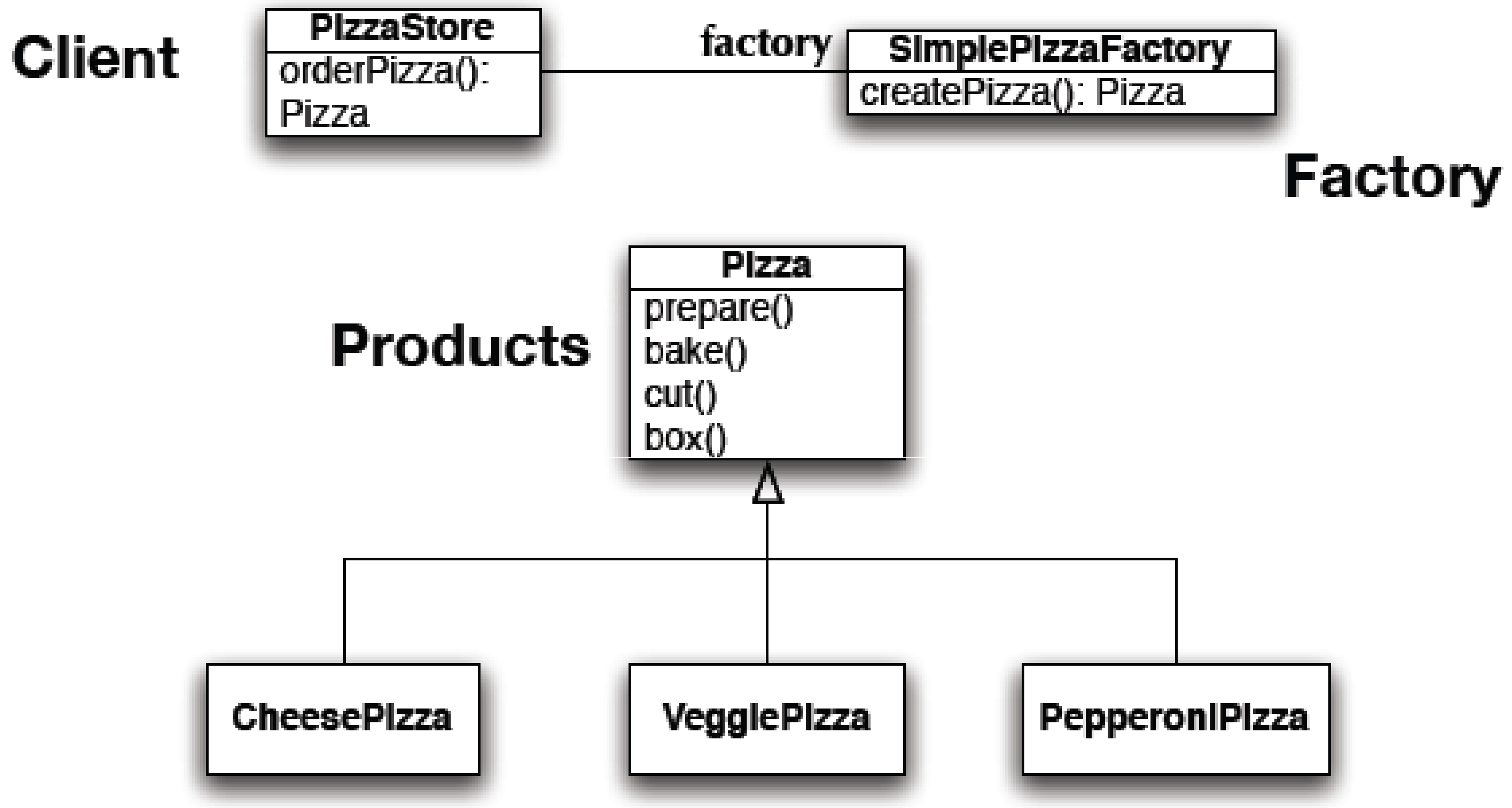
Example3: Pizza
Simple solution: just a factory

```
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;
      }
}
```

```
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();
          }
     }
}
```

Replace concrete instantiation with call to the PizzaFactory to create a new pizza
Now we don't need to mess with this code if we add new pizzas

Class Diagram of New Solution



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

Example3: Pizza
Simple Factory to Factory Method

- To demonstrate the factory method 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 will need its own factory to create pizzas that 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

Example3: Pizza: Factory Method

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

```
public class NYPizzaStore extends PizzaStore {
    public Pizza createPizza(String type) {
             if (type.equals("cheese")) {
                 return new NYCheesePizza();
             } else if (type.equals("greek")) {
                 return new NYGreekPizza();
             } else if (type.equals("pepperoni")) {
                 return new NYPepperoniPizza();
             }
             return null;
    }
}
```

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". Nice.

Consequences

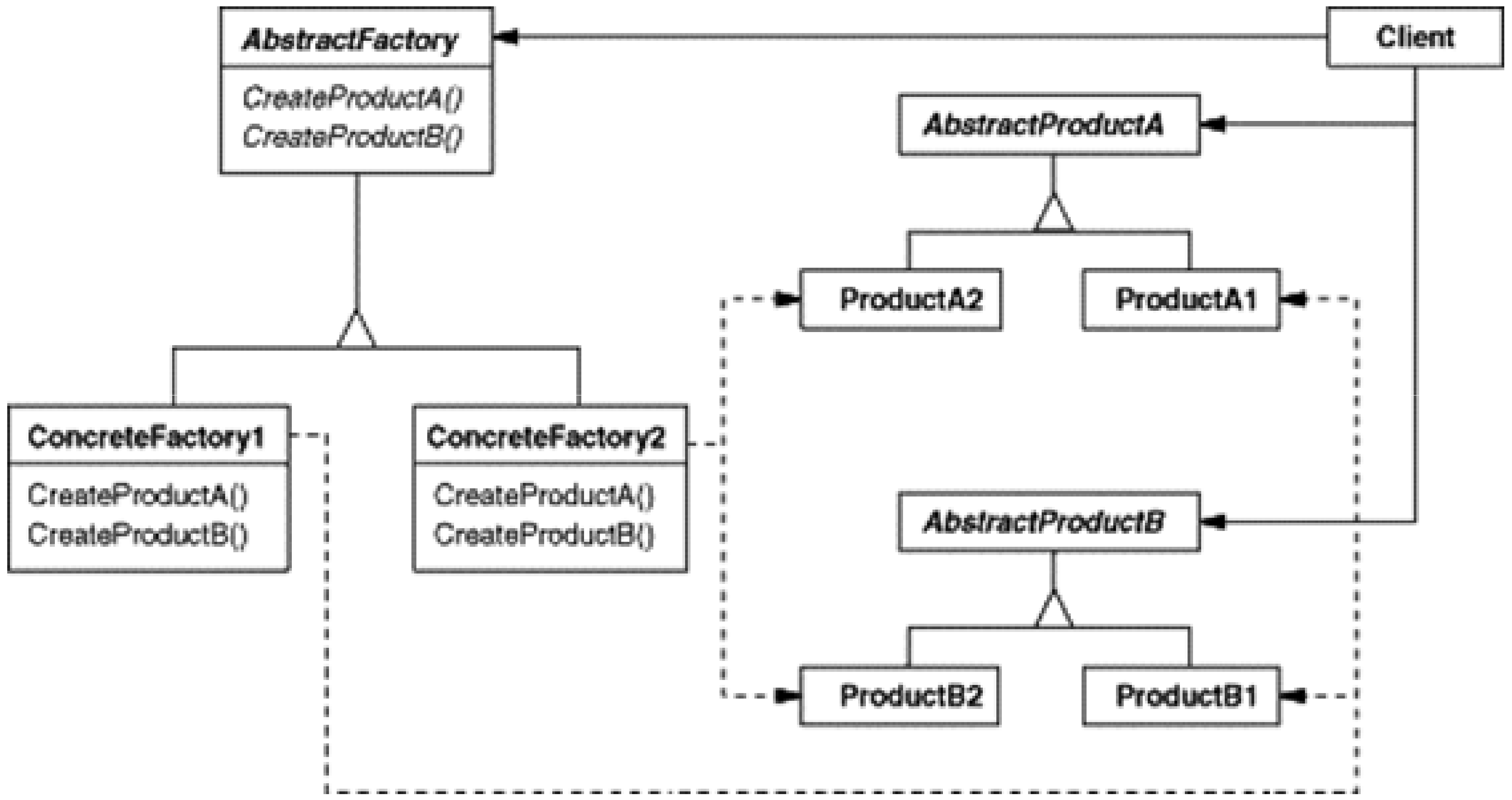
- Benefits
  - Code is made more flexible and reusable by the elimination of instantiation of application-specific classes
  - Code deals only with the interface of the Product class and can work with any ConcreteProduct class that supports this interface
- Liabilities
  - Clients might have to subclass the Creator class just to instantiate a particular ConcreteProduct
- Implementation Issues
  - Creator can be abstract or concrete
  - Should the factory method be able to create multiple kinds of products? If so, then the factory method has a parameter (possibly used in an if-else!) to decide what object to create.

Abstract Factory

Intent

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- The Abstract Factory pattern is very similar to the Factory Method pattern.
  - One difference between the two is that with the Abstract Factory pattern, a class delegates the responsibility of object instantiation to another object via composition whereas the Factory Method pattern uses inheritance and relies on a subclass to handle the desired object instantiation.
- Actually, the delegated object frequently uses factory methods to perform the instantiation!

Abstract Factory: structure



Participants

- AbstractFactory
  - Declares an interface for operations that create abstract product objects
- ConcreteFactory
  - Implements the operations to create concrete product objects
- AbstractProduct
  - Declares an interface for a type of product object
- ConcreteProduct
  - Defines a product object to be created by the corresponding concrete factory
  - Implements the AbstractProduct interface
- Client
  - Uses only interfaces declared by AbstractFactory and AbstractProduct classes

Abstract Factory applied to the MazeGame

```
// MazeFactory.
    public class MazeFactory {
            public Maze makeMaze() {return new Maze();}
            public Room makeRoom(int n) {return new Room(n);}
            public Wall makeWall() {return new Wall();}
            public Door makeDoor(Room r1, Room r2) {
            return new Door(r1, r2);}
            }
```

Note that the MazeFactory class is just a collection of factory methods!

Also, note that MazeFactory acts as both an AbstractFactory and a ConcreteFactory.

Abstract Factory applied to the MazeGame

- The createMaze() method of the MazeGame class takes a MazeFactory reference as a parameter:

```
public class MazeGame {
    public Maze createMaze(MazeFactory factory) {
            Maze maze = factory.makeMaze();
            Room r1 = factory.makeRoom(1);
            Room r2 = factory.makeRoom(2);
            Door door = factory.makeDoor(r1, r2);
            maze.addRoom(r1);
            maze.addRoom(r2);
            r1.setSide(MazeGame.North, factory.makeWall());
            …
            return maze;
}}
```

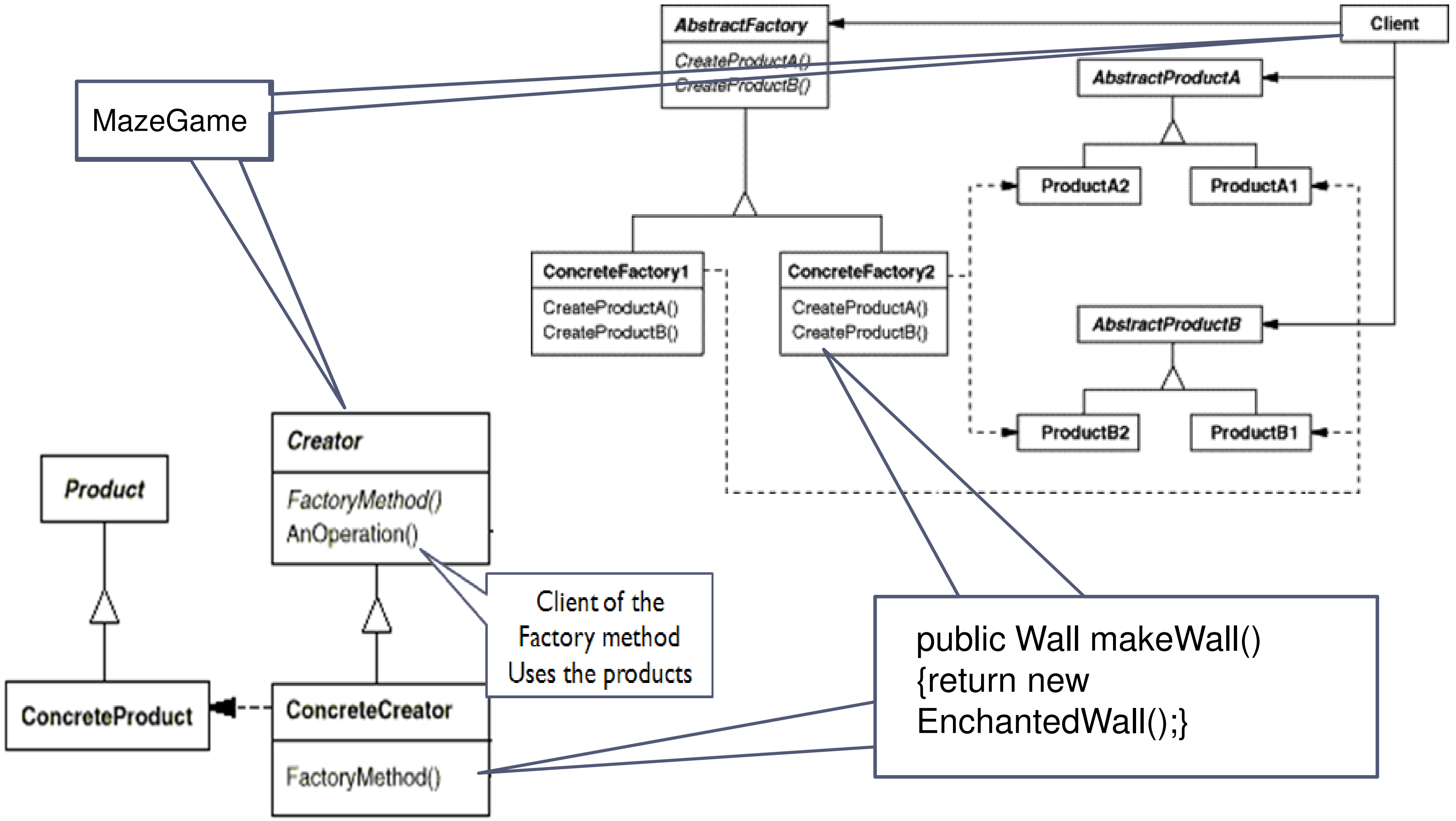
createMaze() delegates the responsibility for creating maze objects to the MazeFactory object

Extend MazeFactory to create other factories

```
public class EnchantedMazeFactory extends MazeFactory {
    public Room makeRoom(int n) {return new EnchantedRoom(n);}
    public Wall makeWall() {return new EnchantedWall();}
    public Door makeDoor(Room r1, Room r2)
                            {return new EnchantedDoor(r1, r2);}
    }
```

- In this example, the correlations are:
  - AbstractFactory => MazeFactory
  - ConcreteFactory => EnchantedMazeFactory (MazeFactory is also a ConcreteFactory)
  - AbstractProduct => MapSite
  - ConcreteProduct => Wall, Room, Door, EnchantedWall, EnchantedRoom, EnchantedDoor

Factory Method   Abstract Factory



public Wall makeWall()
{return new
EnchantedWall();}

The Abstract Factory Pattern: Consequences

- Benefits
  - Isolates clients from concrete implementation classes
  - Makes exchanging product families easy, since a particular concrete factory can support a complete family of products
  - Enforces the use of products only from one family
- Liabilities
  - Supporting new kinds of products requires changing the AbstractFactory interface

The Abstract Factory Pattern: Implementation Issues

- How many instances of a particular concrete factory should there be?
  - An application typically only needs a single instance of a particular concrete factory
- How can the factories create the products?
  - Factory Methods
  - Factories
- How can new products be added to the AbstractFactory interface?
  - AbstractFactory defines a different method for the creation of each product it can produce
  - We could change the interface to support only a make(String kindOfProduct) method

How Do Factories Create Products? Method 1: Use Factory Methods

```
/**
* WidgetFactory.
* This WidgetFactory is an abstract class.
* Concrete Products are created using the
    factory methods
* implemented by sublcasses.
*/
public abstract class WidgetFactory {
    public abstract Window createWindow();
    public abstract Menu createScrollBar();
    public abstract Button createButton();
}
```

```
/**
* MotifWidgetFactory.
* Implements the factory methods of its
    abstract superclass.
*/
public class MotifWidgetFactory
extends WidgetFactory {
public Window createWindow() {return
    new MotifWindow();}
public ScrollBar createScrollBar() {
    return new MotifScrollBar();}
public Button createButton() {return new
    MotifButton();}
}
```

Typical client code: (Note: the client code is the same no matter how the factory creates the product!)

```
...
WidgetFactory wf = new MotifWidgetFactory();        // Create new factory.
Button b = wf.createButton();                       // Create a button.
Window w = wf.createWindow()                         // Create a window.
```

How Do Factories Create Products? Method 2: Use Factories

```
/**
* WidgetFactory.
* This WidgetFactory contains references to factories
* (composition!) used to create the Concrete Products.
* But it relies on a subclass constructor to create the
* appropriate factories.
*/
public abstract class WidgetFactory {
    protected WindowFactory windowFactory;
    protected ScrollBarFactory scrollBarFactory;
    protected ButtonFactory buttonFactory;
    public Window createWindow() {return
            windowFactory.createWindow();}
    public ScrollBar createScrollBar() {return
            scrollBarFactory.createScrollBar();}
    public Button createButton() {return
            buttonFactory.createButton();}
}
```

```
/**
* MotifWidgetFactory.
* Instantiates the factories used by its superclass.
*/
public class MotifWidgetFactory
        extends WidgetFactory {
    public MotifWidgetFactory() {
      windowFactory = new MotifWindowFactory();
      scrollBarFactory = new MotifScrollBarFactory();
      buttonFactory = new MotifButtonFactory();
    }
}
```

How Do Factories Create Products? Method 3: Use Factories With No Required Subclasses

```
/**
* WidgetFactory.
* This WidgetFactory contains reference to factories used
* to create Concrete Products. But it does not need to be
* subclassed. It has an appropriate constructor to set
* these factories at creation time and mutators to change
* them during execution.
*/
public class WidgetFactory {
private WindowFactory windowFactory;
private ScrollBarFactory scrollBarFactory;
private ButtonFactory buttonFactory;
public WidgetFactory(WindowFactory wf,
                ScrollBarFactory sbf,  ButtonFactory bf) {
    windowFactory = wf;
    scrollBarFactory = sbf;
                buttonFactory = bf;
}
public void setWindowFactory(WindowFactory wf) {
    windowFactory = wf;
}
public void setScrollBarFactory(ScrollBarFactory sbf) {
    scrollBarFactory =sbf;
}
public void setButtonFactory(ButtonFactory bf) {
    buttonFactory = bf;
}
public Window createWindow() {return
    windowFactory.createWindow();}
public ScrollBar createScrollBar() {return
    scrollBarFactory.createScrollBar();}
public Button createButton() {return
    buttonFactory.createButton();}
}
```

This is Strategy…

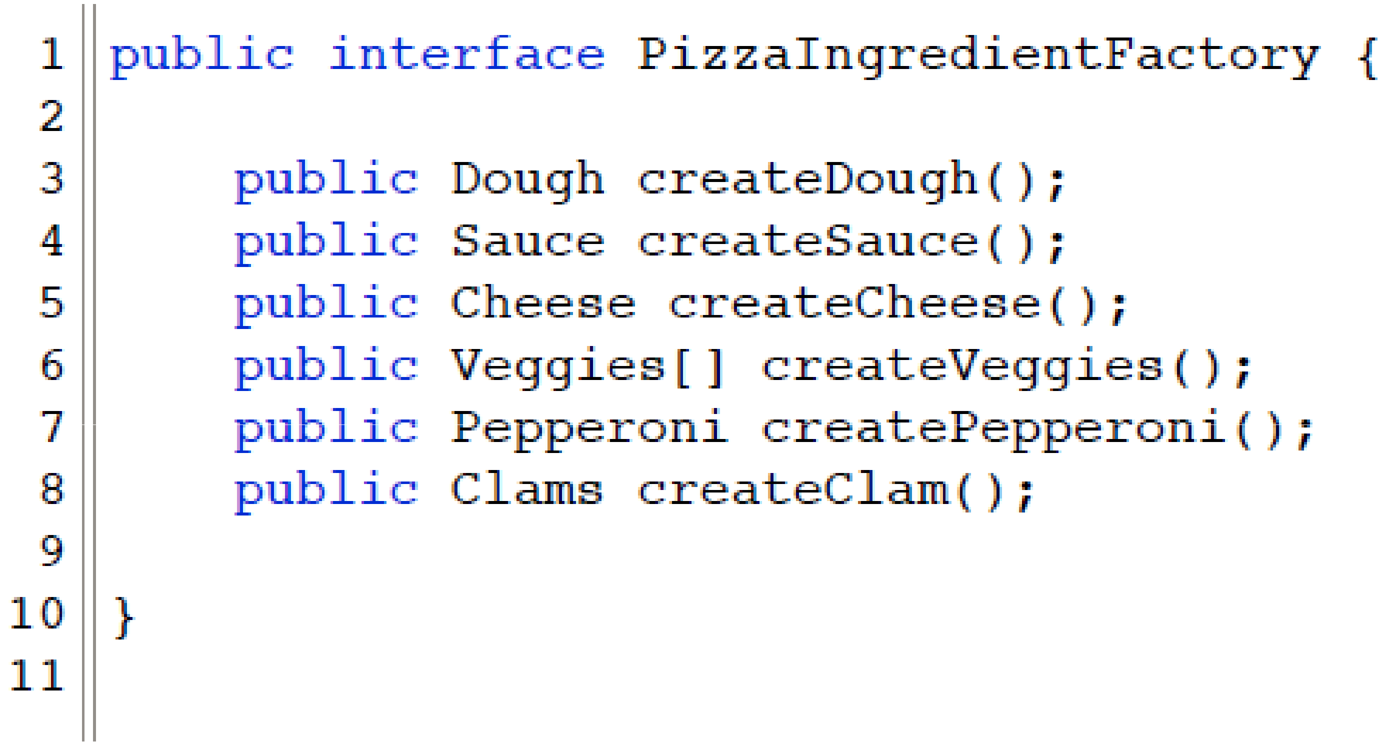
Moving On

- •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!"

Abstract Factory to the Rescue!

- 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. ☺

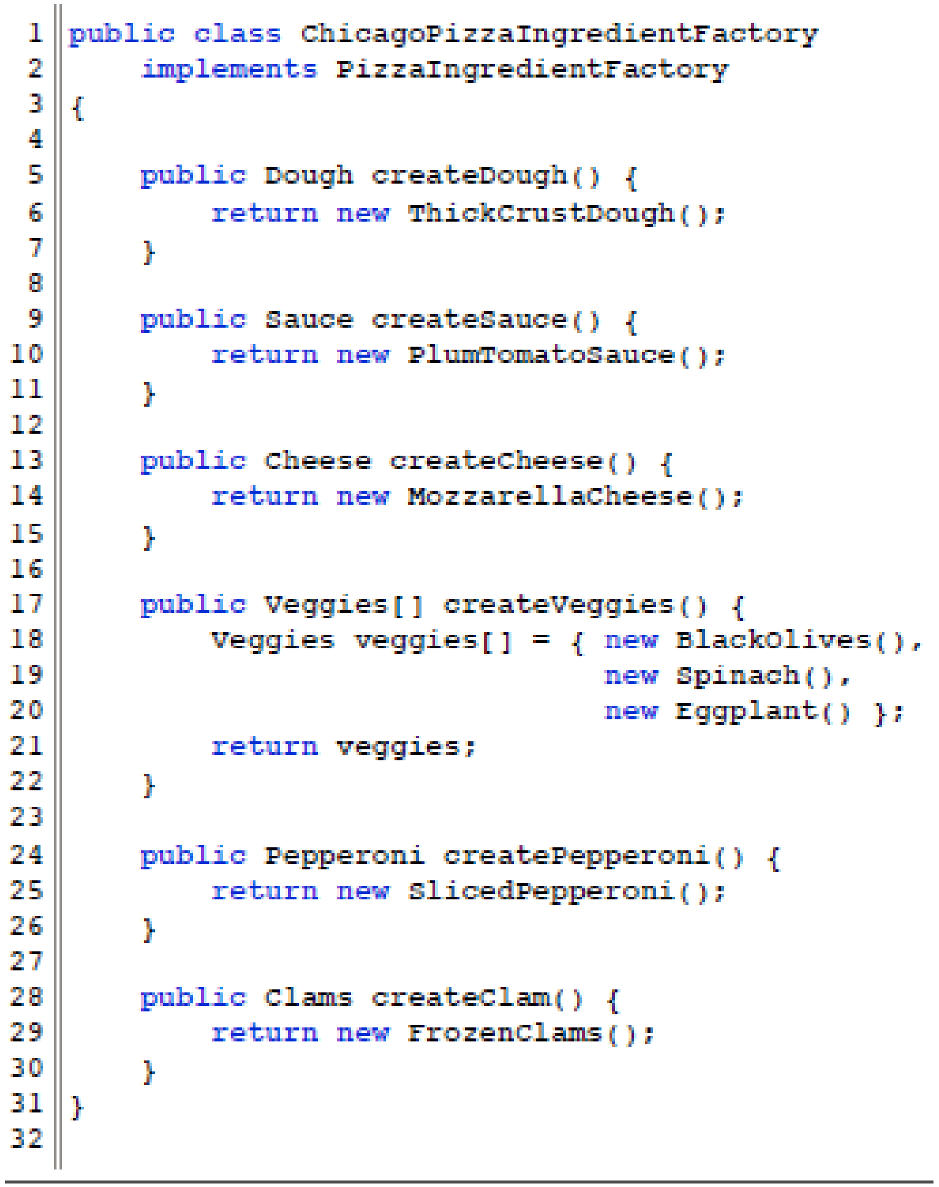
First, We need a Factory Interface



```java
public interface PizzaIngredientFactory {

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

}
```

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

Second, We implement a Region-Specific Factory

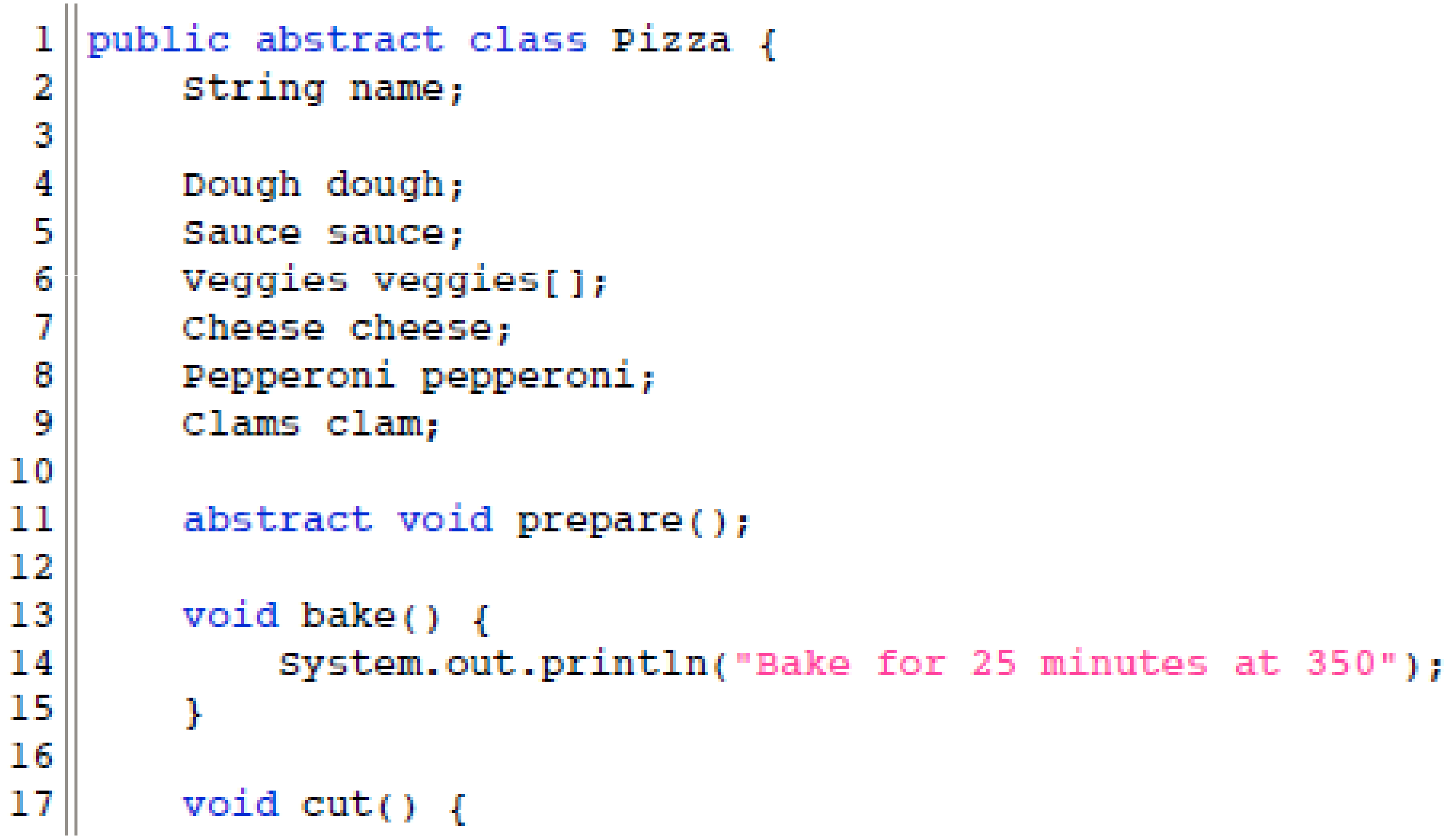


```java
public class ChicagoPizzaIngredientFactory
    implements PizzaIngredientFactory
{

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

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

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

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

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

    public Clams createClam() {
        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 who live in Chicago
- But how (or where) is this factory used?

Within Pizza Subclasses… (I)

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



```java
public abstract class Pizza {
    String name;

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

    abstract void prepare();

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

    void cut() {
```

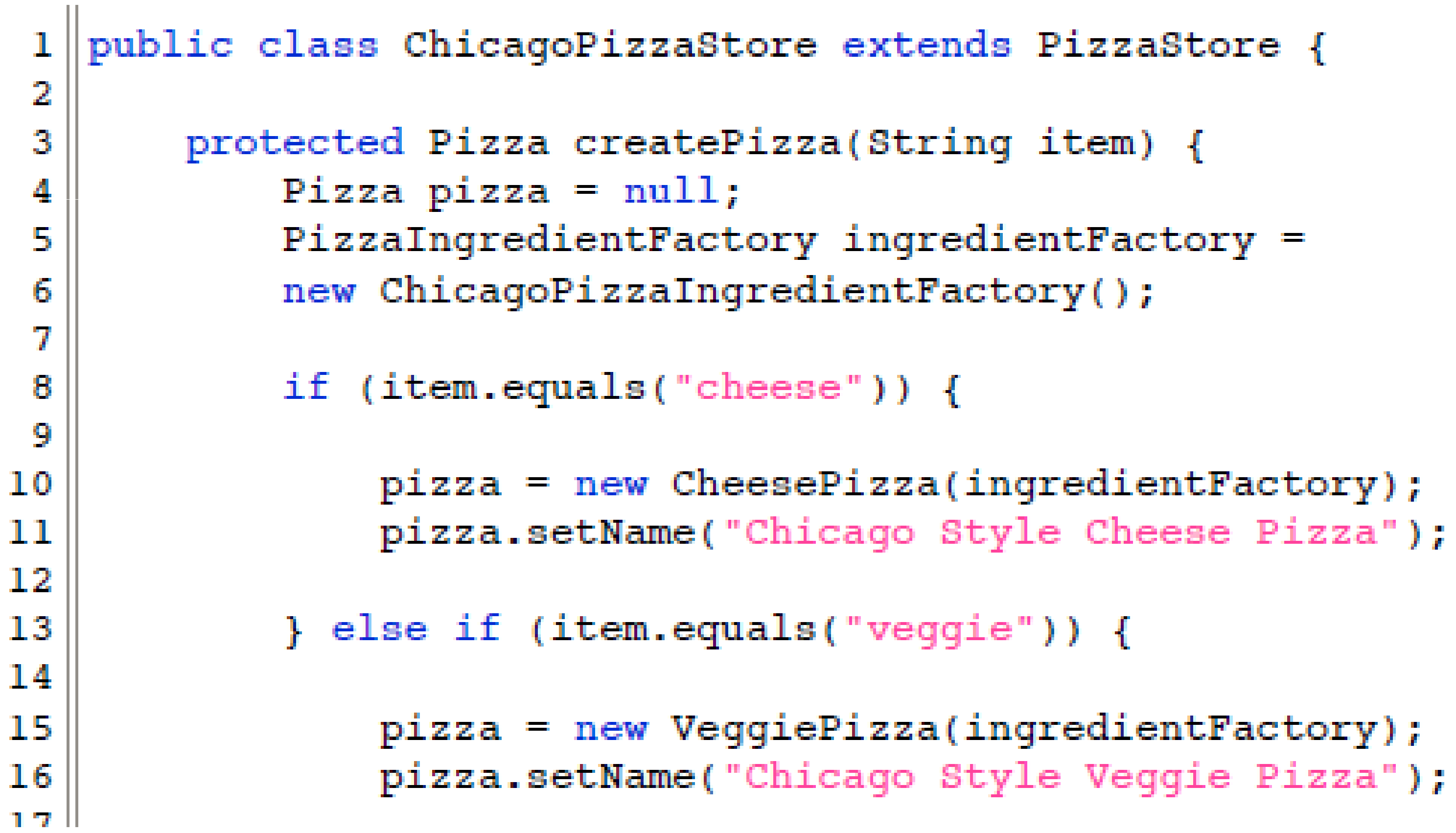
Within Pizza Subclasses… (II)

- 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

```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();
    }
}

```

One last step…

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



```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");
```

...

Summary: What did we just do?

1) We created an ingredient factory interface to allow for the creation of a family of ingredients for a particular pizza
2) This abstract factory gives us an interface for creating a family of products
   1) The factory interface decouples the client code from the actual factory implementations that produce context-specific sets of products
3) 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)