Applying the Factory Method Design Pattern

Trenton Computer Festival Professional Seminars
April 21, 2006

Michael P. Redlich
(908) 730-3416
michael.p.redlich@exxonmobil.com

About Myself

- **Degree**
  - **B.S. in Computer Science**
  - **Rutgers University (go Scarlet Knights!)**
- **ExxonMobil Research & Engineering**
  - **Clinton, New Jersey**
  - **Senior Research Technician (1988-1998, 2004-present)**
  - **Systems Analyst (1998-2002)**
- **Ai-Logix, Inc.**
  - **Somerset, New Jersey**
  - **Technical Support Engineer (2003-2004)**

About Myself (continued)

- **ACGNJ**
  - **Java Users Group Leader**
  - **Secretary**
- **Publications**
  - **“Avoid Excessive Subclassing with the Decorator Design Pattern”**
    - **Barry Burd and Michael Redlich**
    - ***Java Boutique*, January 27, 2006**
  - **“James: The Java Apache Mail Enterprise Server”**
    - **Barry Burd and Michael Redlich**
    - ***Java Boutique*, September 30, 2005**

Example Source Code

- **The example source code was adapted from:**
  - **Head First Design Patterns**
    - **Eric & Elisabeth Freeman (with Kathy Sierra & Bert Bates)**
- **Download example source code from:**
  - **http://tcf.redlich.net/**

Gang of Four (GoF)

- **Erich Gamma**
- **Richard Helm**
- **Ralph Johnson**
- **John Vlissides**
- **Design Patterns – Elements of Reusable Object-Oriented Software**
  - **ISBN 0-201-63361-2**
  - **1995**

Gang of Four (GoF) Next Generation?

- **Eric Freeman**
- **Elisabeth Freeman**
- **Kathy Sierra**
- **Bert Bates**
- **Head First Design Patterns**
  - **ISBN 0-596-00712-4**
  - **2004**

What are Design Patterns?

- **A pattern is a solution to a problem in a context**
- **The context is the situation in which the pattern applies**
- **The problem refers to the desired goal in the context, but also refers to any constraints that may occur**
- **The solution is a general design that anyone can apply**

"If you find yourself in a context with a problem that has a goal that is affected by a set of constraints, then you can apply a design that resolves the goal and constraints and leads to a solution."

What are Design Patterns? (continued)

- **Recurring solutions to software design problems that are repeatedly found in real-world application development**
- **All about the design and interaction of objects**
- **Four essential elements:**
  - **The pattern name**
  - **The problem**
  - **The solution**
  - **The consequences**

How Design Patterns Solve Design Problems

- **Find appropriate objects**
  - **Helps identify less obvious abstractions**
- **Design for change**
  - **Avoid creating objects directly**
  - **Avoid dependencies on specific operations**
  - **Avoid algorithmic dependencies**
  - **Avoid tight coupling**

Thinking in Design Patterns

- **Keep it simple**
  - **Goal should be simplicity**
- **Design patterns are not a magic bullet**
  - **No “plug and play”**
- **Know when to apply a design pattern**
  - **Ensure that a pattern fits the design**
- **Consider patterns during refactoring**
  - **Goal is to improve structure, not behavior**
- **Don’t be afraid to remove a design pattern**
  - **Especially if design has become too complex**

Design Pattern Categories

- **Creational**
  - **Abstracts the instantiation process**
  - **Dynamically create objects so that they don't have to be instantiated directly**
- **Structural**
  - **Composes groups of objects into larger structures**
- **Behavioral**
  - **Defines communication among objects in a given system**
  - **Provides better control of flow in a complex application**

Creational Patterns

- **Abstract Factory**
  - **Provides an interface for creating related objects without specifying their concrete classes**
- **Builder**
  - **Reuses the construction process of a complex object**
- **Factory Method**
  - **Lets subclasses decide which class to instantiate from a defined interface**
- **Prototype**
  - **Creates new objects by copying a prototype**

Creational Patterns (continued)

- **Singleton**
  - **Ensures a class has only one instance with a global point of access to it**

Structural Patterns

- **Adapter**
  - **Converts the interface of one class to an interface of another**
- **Bridge**
  - **Decouples an abstraction from its implementation**
- **Composite**
  - **Composes objects into tree structures to represent hierarchies**
- **Decorator**
  - **Attaches responsibilities to an object dynamically**

Structural Patterns (continued)

- **Façade**
  - **Provides a unified interface to a set of interfaces**
- **Flyweight**
  - **Supports large numbers of fine-grained objects by sharing**
- **Proxy**
  - **Provides a surrogate for another object to control access to it**

Behavioral Patterns

- **Chain of Responsibility**
  - **Passes a request along a chain of objects until the appropriate one handles it**
- **Command**
  - **Encapsulates a request as an object**
- **Interpreter**
  - **Defines a representation and an interpreter for a language grammar**
- **Iterator**
  - **Provides a way to access elements of an object sequentially without exposing its implementation**

Behavioral Patterns (continued)

- **Mediator**
  - **Defines an object that encapsulates how a set of objects interact**
- **Memento**
  - **Captures an object's internal state so that it can be later restored to that state if necessary**
- **Observer**
  - **Defines a one-to-many dependency among objects**
- **State**
  - **Allows an object to alter its behavior when its internal state changes**

Behavioral Patterns (continued)

- **Strategy**
  - **Encapsulates a set of algorithms individually and makes them interchangeable**
- **Template Method**
  - **Lets subclasses redefine certain steps of an algortithm**
- **Visitor**
  - **Defines a new operation without changing the classes on which it operates**

Pizza Store Application

- **Objective:**
  - **Develop a pizza shop application**
  - **Plan for expansion**

```
public class PizzaStore {

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

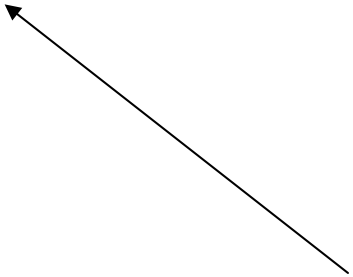
IS THIS A GOOD APPROACH?

Area of change

Area that we expect to remain unchanged

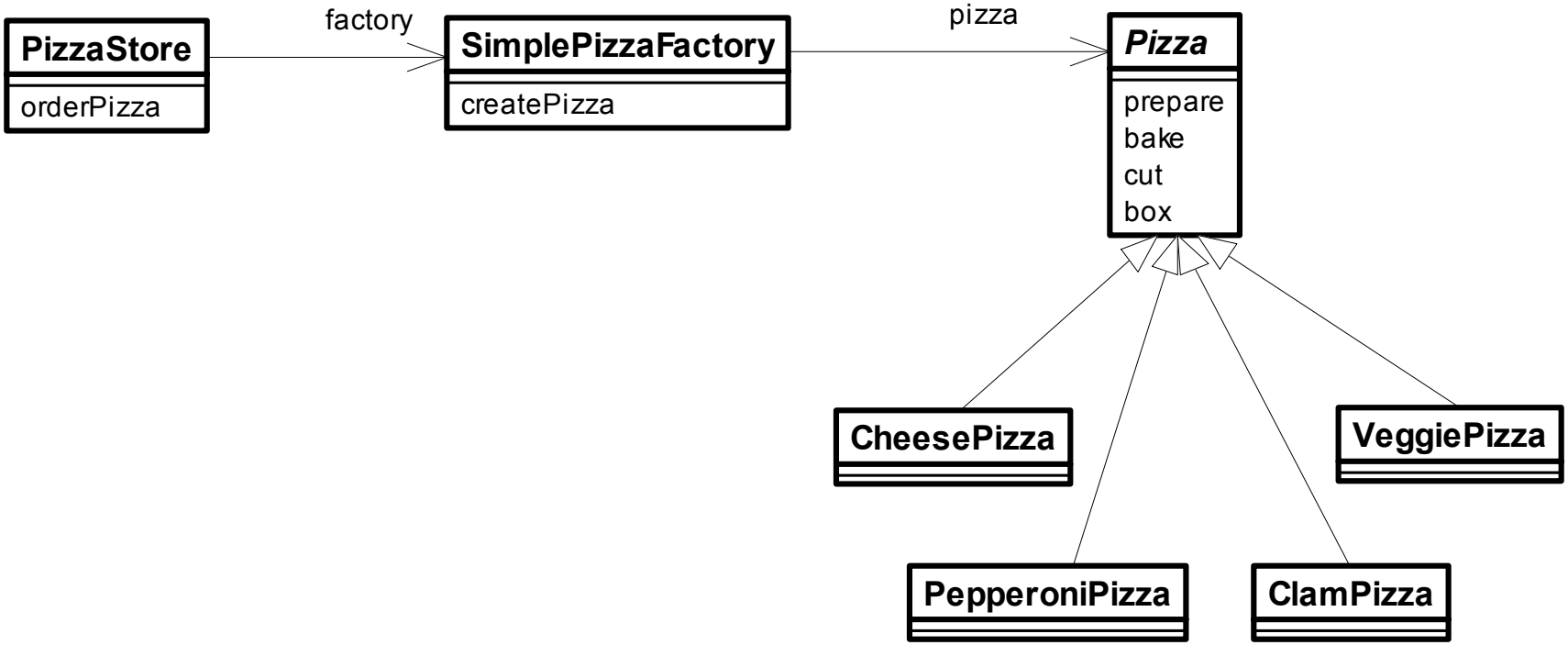
```
public class SimplePizzaFactory {
  public Pizza createPizza(String type) {
    Pizza pizza = null;
    if(type.equals("cheese"))
      pizza = new CheesePizza();
    else if(type.equals("pepperoni"))
      pizza = new PepperoniPizza();
    return pizza;
  }
}
```

```
public class PizzaStore {
  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;
  }
}
```

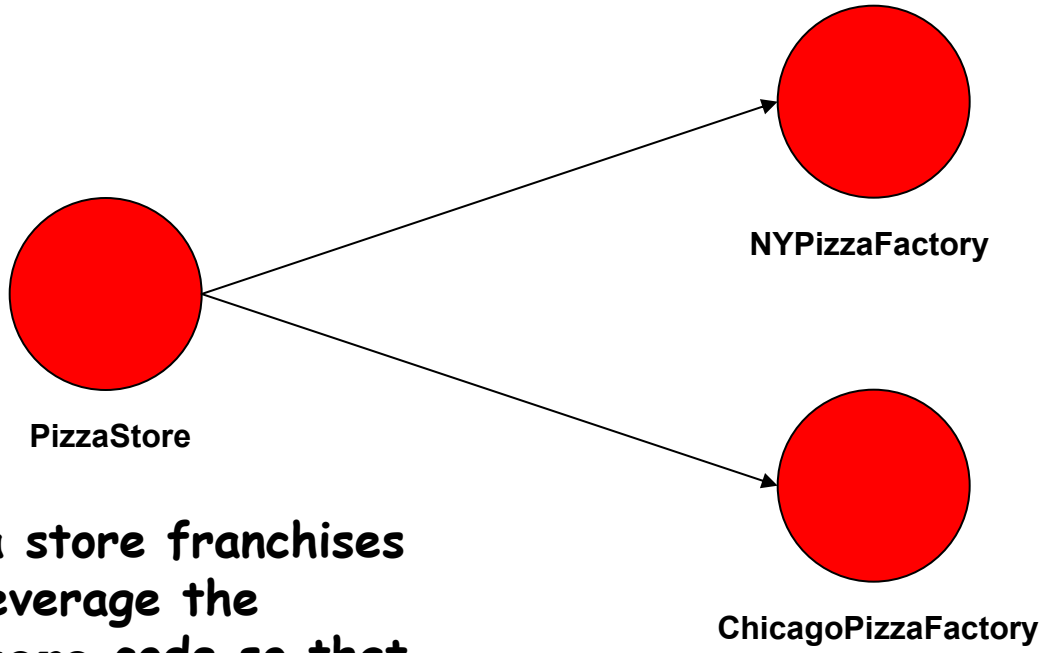


Notice how the `createPizza()` method eliminates the need to use the `new` keyword

Simple Factory



Expanding the Pizza Store



This franchise likes to make pizza with thin crust, tasty sauce, and just a little cheese.

This franchise likes to make pizza with thick crust, rich sauce and lots of cheese.

All pizza store franchises should leverage the `PizzaStore` code so that pizzas are prepared the same way.

```
public class PizzaTestDrive {

// instance variables...

NYPizzaFactory nyFactory = new NYPizzaFactory();

PizzaStore nyStore = new PizzaStore(nyFactory);

nyStore.orderPizza("pepperoni");

ChicagoPizzaFactory chicagoFactory = new ChicagoFactory();

PizzaStore chicagoStore = new PizzaStore(chicagoFactory);

chicagoStore.orderPizza("pepperoni");

}
```

IS THIS A BETTER APPROACH?

Here we get NY style pizza

Here we get Chicago style pizza

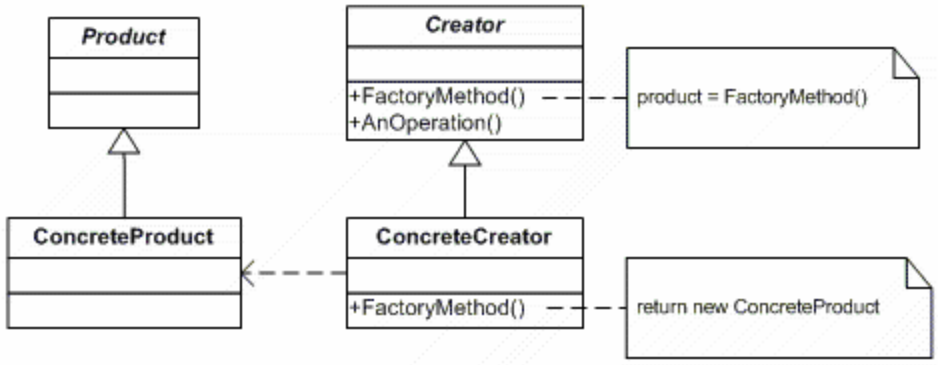
Factory Method

- **Intent**
  - **Defines an interface for creating an object, but lets subclasses decide which class to instantiate**
  - **Lets a class defer instantiation to subclasses**
- **Also known as**
  - **Virtual Constructor**
- **Motivation**
  - **To solve the problem of one class knowing *when* to create a class of another type, but not knowing *what kind* of class to create**

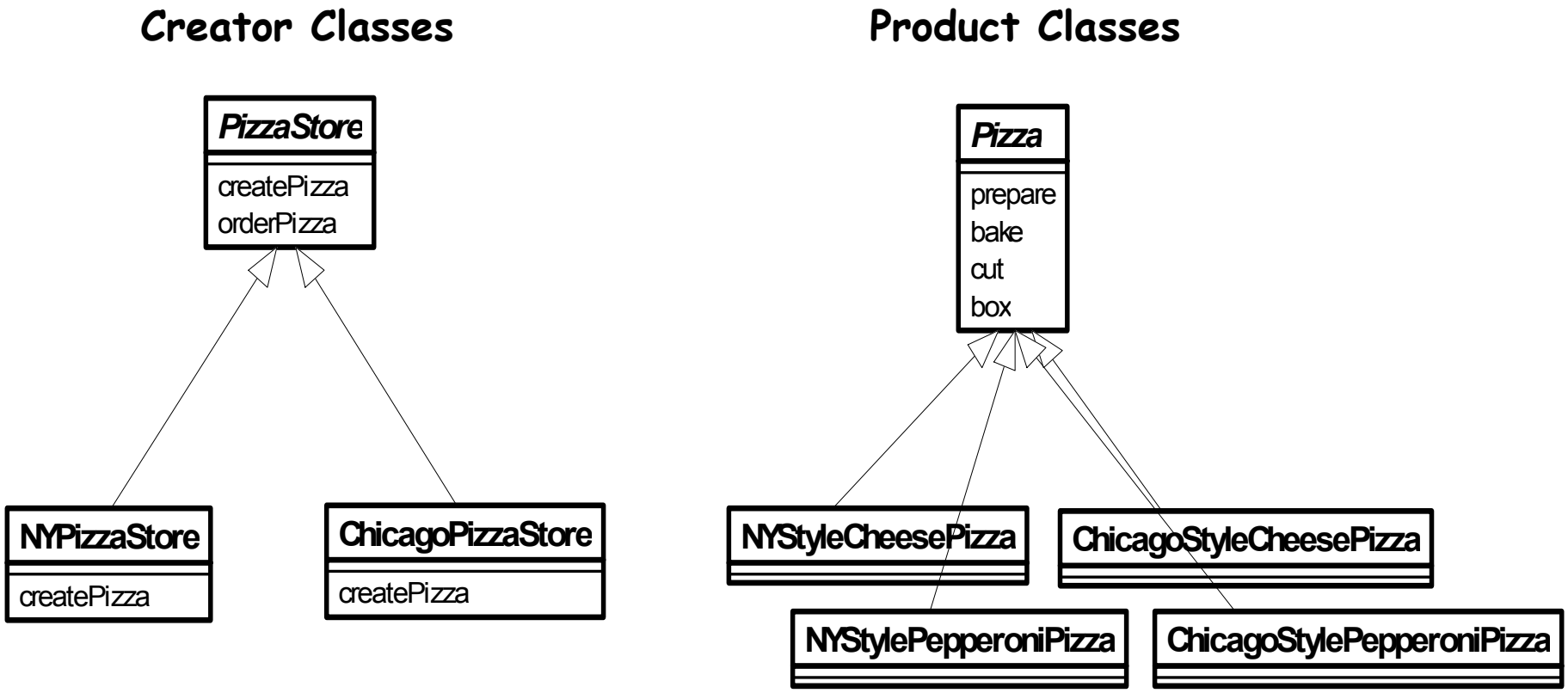
Factory Method (continued)

- **Design Principle**
  - **Depend upon abstractions; do not depend upon concrete classes**
- **Use this pattern when:**
  - **A class can't anticipate the class of objects is must create**
  - **A class would prefer for its subclasses to specify the objects it creates**
  - **There is a need for a class to localize one of several helper classes that can be delegated a responsibility**

Factory Method (continued)



Pizza Store


Creator Classes
Product Classes
PizzaStore
createPizza
orderPizza
NYPizzaStore
createPizza
ChicagoPizzaStore
createPizza
Pizza
prepare
bake
cut
box
NYStyleCheesePizza
ChicagoStyleCheesePizza
NYStylePepperoniPizza
ChicagoStylePepperoniPizza

And Now…

- **…for the code review and demonstration!**

Resources

- **Design Patterns – Elements of Reusable Object-Oriented Software**
  - **Erich Gamma, et. al**
  - **ISBN 0-201-63361-2**
- **Head First Design Patterns**
  - **Eric & Elisabeth Freeman (with Kathy Sierra & Bert Bates)**
  - **ISBN 0-596-00712-4**
  - **http://www.wickedlysmart.com/**
- **Java Design Patterns**
  - **James W. Cooper**
  - **ISBN 0-201-48539-7**
  - **http://www.patterndepot.com/put/8/JavaPatterns.htm**

Resources (continued)

- **UML Distilled**
  - **Martin Fowler (with Kendall Scott)**
  - **ISBN 0-201-32563-2**
- **Data & Object Factory**
  - **http://www.dofactory.com/**