

Design Patterns

Design Pattern

- Definition
 - A named general reusable solution to common design problems
 - Used in Java libraries
- Major source: GoF book 1995
 - "Design Patterns: Elements of Reusable Object-Oriented Software"
 - 24 design patterns

Purpose-based Pattern Classification

- **Creational**
 - About the process of object creation
- **Structural**
 - About composition of classes or objects
- **Behavioral**
 - About how classes or objects interact and distribute responsibility

N. Meng, B. Ryder

3

Design pattern space

Scope	Class	Purpose		
		Creational	Structural	Behavioral
	Object	Factory Method (107) Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (class) (139) Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Interpreter (243) Template Method (325) Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

N. Meng, B. Ryder

4

Adapter Pattern

- Problem: incompatible interfaces
- Solution: create a wrapper that maps one interface to another
 - Key point: neither interface has to change and they execute in decoupled manner



N. Meng, B. Ryder

5

Example

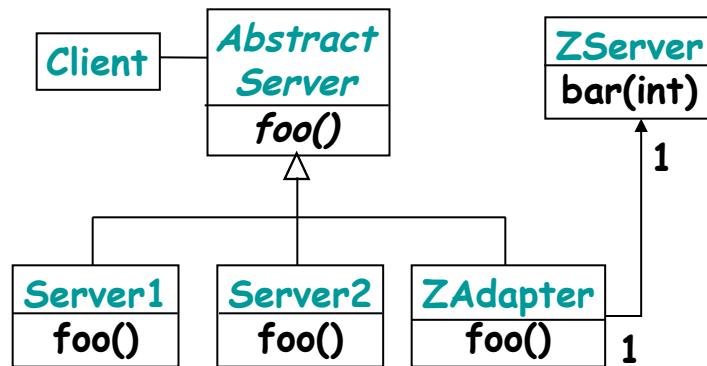


- Problem
 - Client written against some defined interface
 - Server with the right functionality but with a different interface
- Options
 - Change the client
 - Change the server
 - Create an adapter to wrap the server

N. Meng, B. Ryder

6

Example



N. Meng, B. Ryder

7

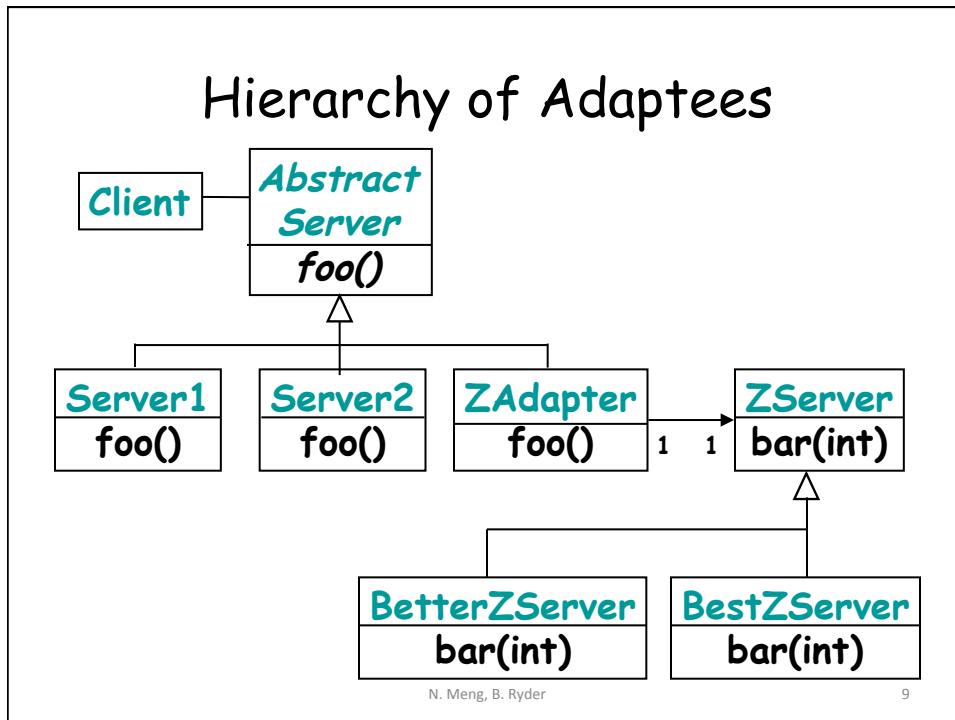
Sample Java Code

```

abstract class AbstractServer { abstract void foo(); }
class ZAdapter extends AbstractServer {
    private ZServer z;
    public ZAdapter() { z = new ZServer(); }
    public void foo() { z.bar(5000); }
    //wrap call to ZServer method
}
...
somewhere in client code:
AbstractServer s = new ZAdapter();
  
```

N. Meng, B. Ryder

8



Sample Java Code

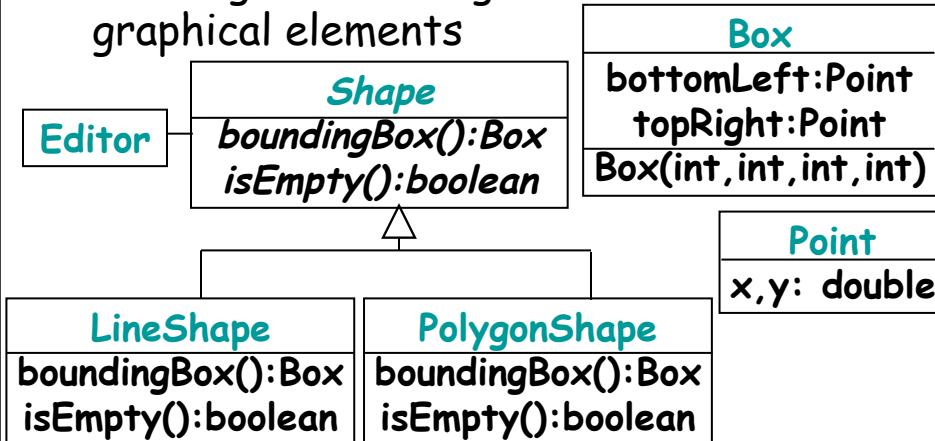
```

abstract class AbstractServer
{ abstract void foo(); }
class ZAdapter extends AbstractServer {
    private ZServer z;
    public ZAdapter(int perf) {
        if (perf > 10) z = new BestZServer();
        else if (perf > 3) z = new BetterZServer();
        else z = new ZServer();
    }
    public void foo() { z.bar(5000); }
}
  
```

N. Meng, B. Ryder 10

Another Adapter Example

- Drawing editor: diagrams built with graphical elements

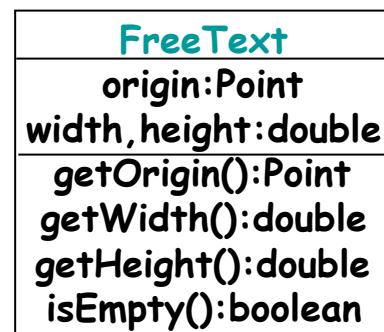


N. Meng, B. Ryder

11

Adding TextShape

- Problem: mismatched interfaces
- Solution: create a **TextShape** adapter



N. Meng, B. Ryder

12

Sample Java Code

```
class TextShape implements Shape {
    private FreeText t;
    public TextShape() { t = new FreeText(); }
    public boolean isEmpty() { return t.isEmpty(); }
    public Box boundingBox() {
        int x1 = toInt(t.getOrigin().getX());
        int y1 = toInt(t.getOrigin().getY());
        int x2 = toInt(x1 + t.getWidth());
        int y2 = toInt(y1 + t.getHeight());
        return new Box(x1,y1,x2,y2); }
    private int toInt(double) { ... } }
```

N. Meng, B. Ryder

13

Pluggable Adapters

- Preparation for future adaptation
 - Define a narrow interface
- Future users of our code will write adapters to implement the interfaces
 - E.g., ITaxCalculator

N. Meng, B. Ryder

14

Factory Pattern

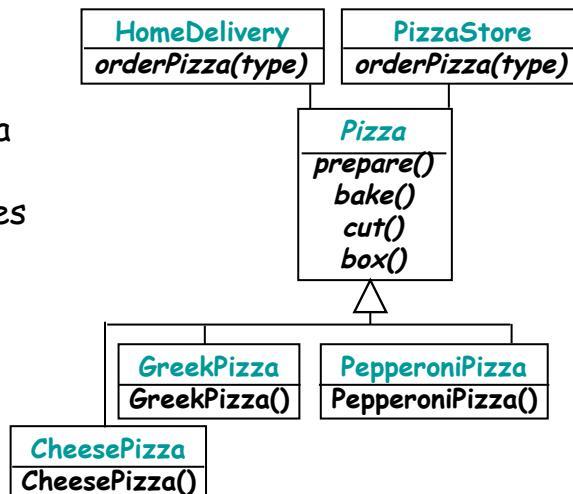
- Problem: there are many ways to create certain objects
- Solution: create a framework that is responsible for creating the objects
 - Key point: clients do not know details about object creation

N. Meng, B. Ryder

15

Example

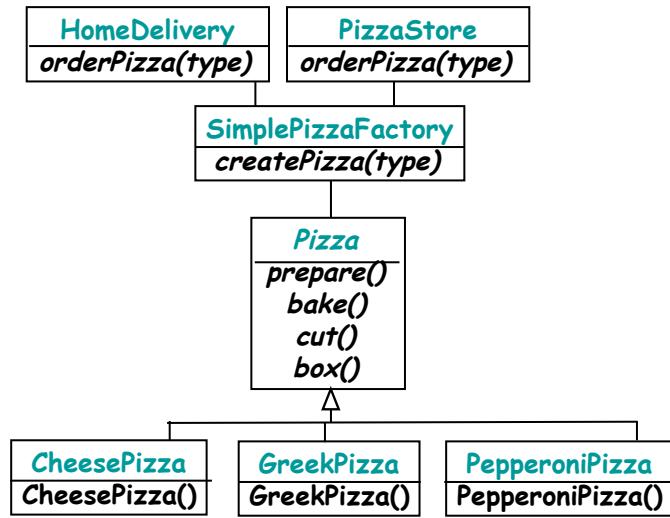
- Problem
 - Clients invoke different pizza constructors
 - New pizza types may be added
 - Clam, Veggie
 - Original pizza types may be removed
 - Greek



N. Meng, B. Ryder

16

Solution: Encapsulate object creation



N. Meng, B. Ryder

17

Sample Java Code

```

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;
    }
}
  
```

N. Meng, B. Ryder

18

```

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;
    }
}

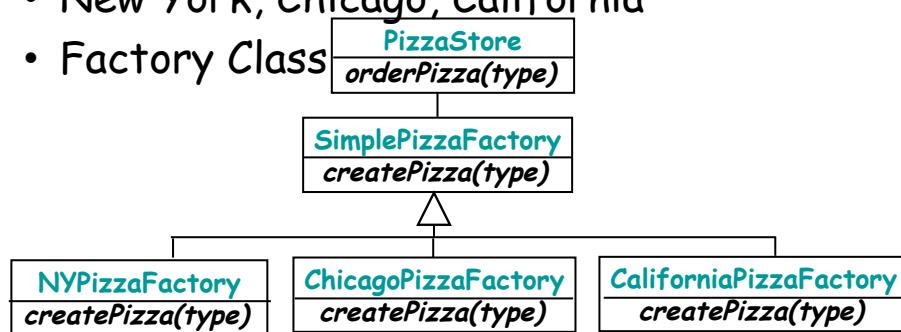
```

N. Meng, B. Ryder

19

Different Styles of Pizza?

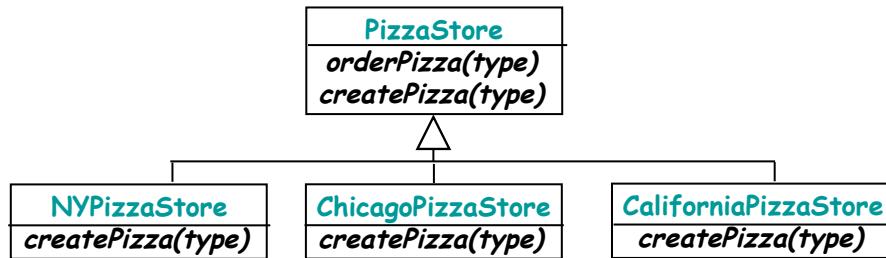
- New York, Chicago, California
- Factory Class



N. Meng, B. Ryder

20

An Alternative Approach: Factory Method



N. Meng, B. Ryder

21

Sample Code

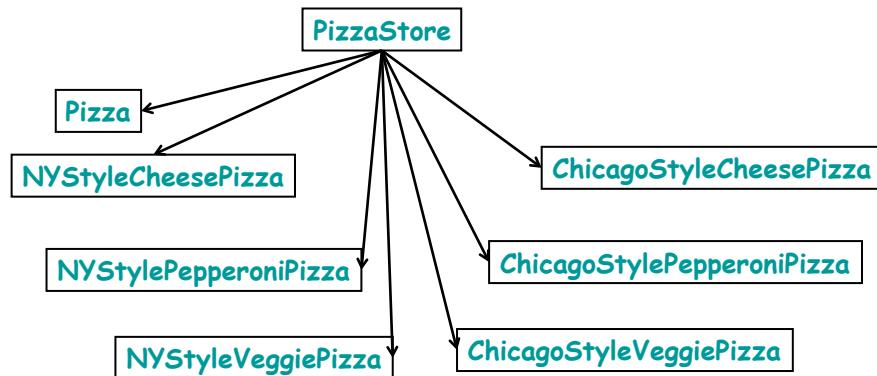
```

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

N. Meng, B. Ryder

22

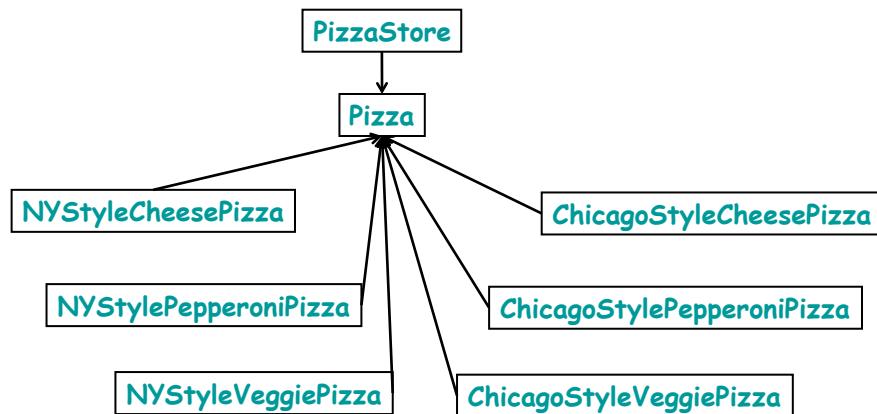
The Original Object Dependencies



N. Meng, B. Ryder

23

With Factory Pattern



N. Meng, B. Ryder

24

Factory Pattern

- The Dependency Inversion Principle
 - Depend upon abstractions instead of concretizations.
 - Use the pattern when
 - a class cannot anticipate the class of objects it will create
 - A class wants its subclasses to specify the objects to create

N. Meng, B. Ryder

25

Design Pattern Presentation

- Problem
- Solution
- Example(s)
- Keypoint(s)
- 8-10 slides
- Due: 10/21/2015 choices + slides (if you want to present design patterns)

N. Meng, B. Ryder

26