Design Patterns

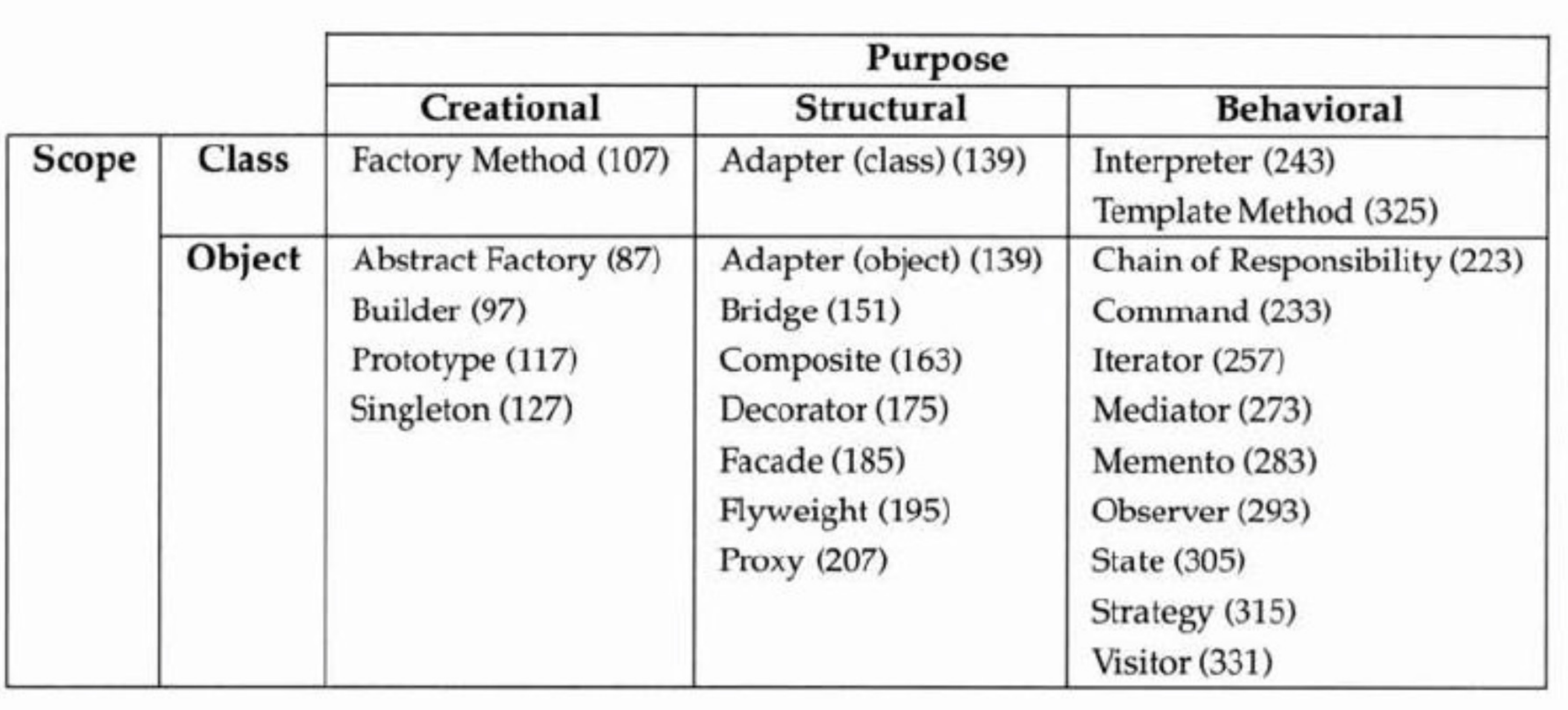
Design Pattern

- 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

Design pattern space

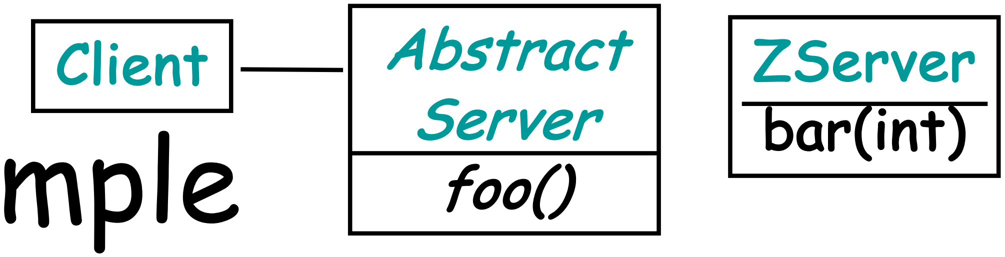


		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight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

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

Example


Client

Abstract Server

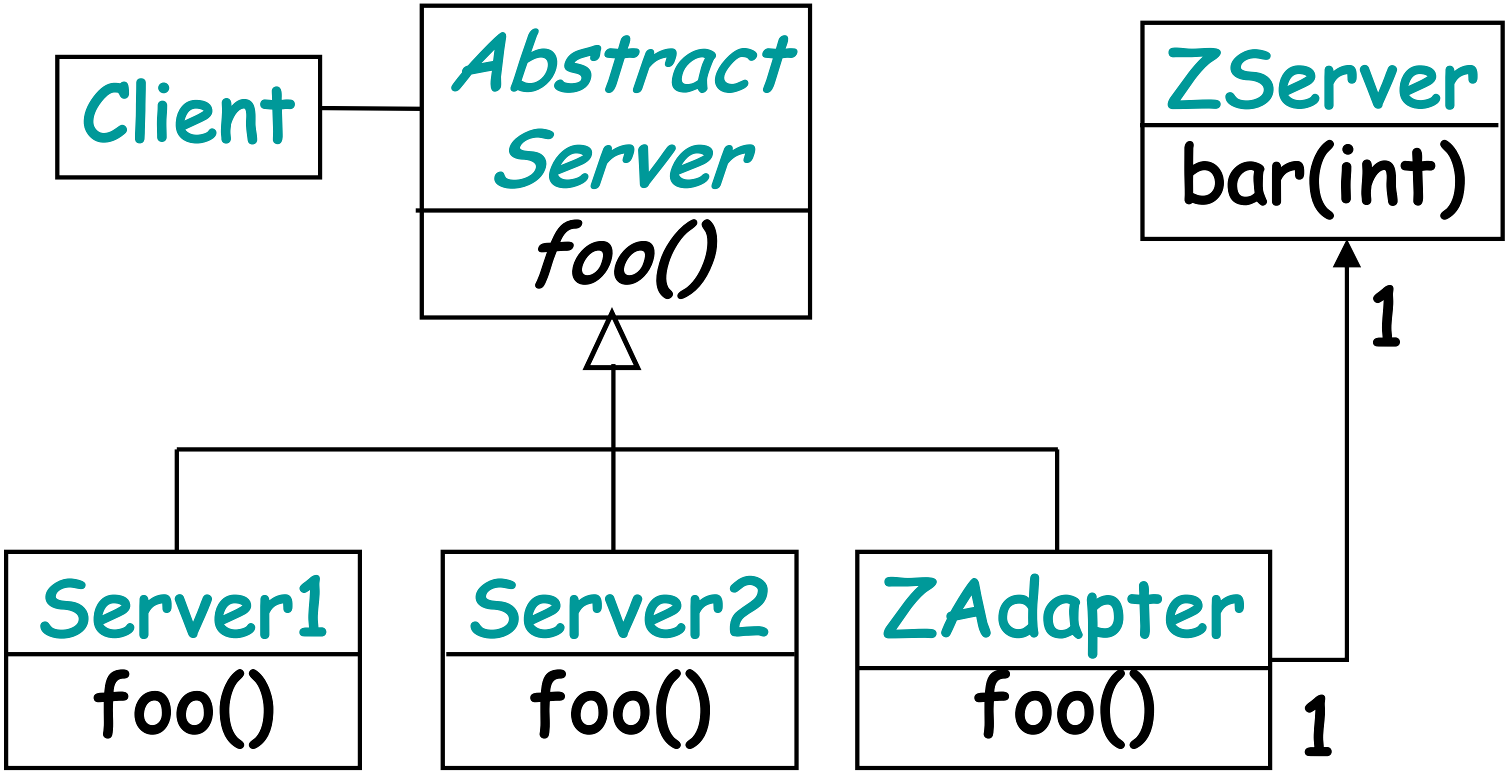
foo()

ZServer

bar(int)

- 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

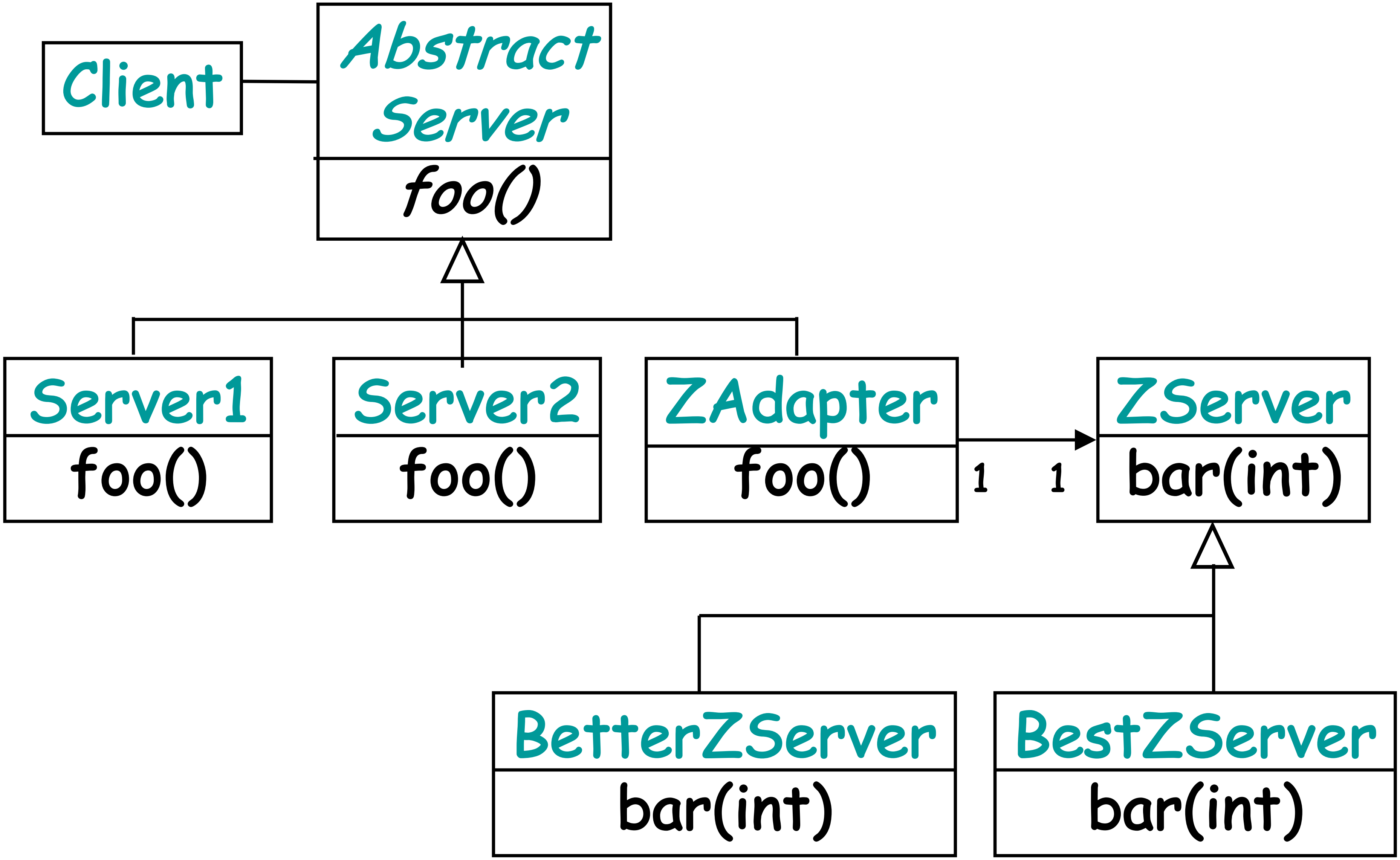
Example


Client
Abstract Server
foo()
ZServer
bar(int)
1
Server1
foo()
Server2
foo()
ZAdapter
foo()
1

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();
```

Hierarchy of Adaptees

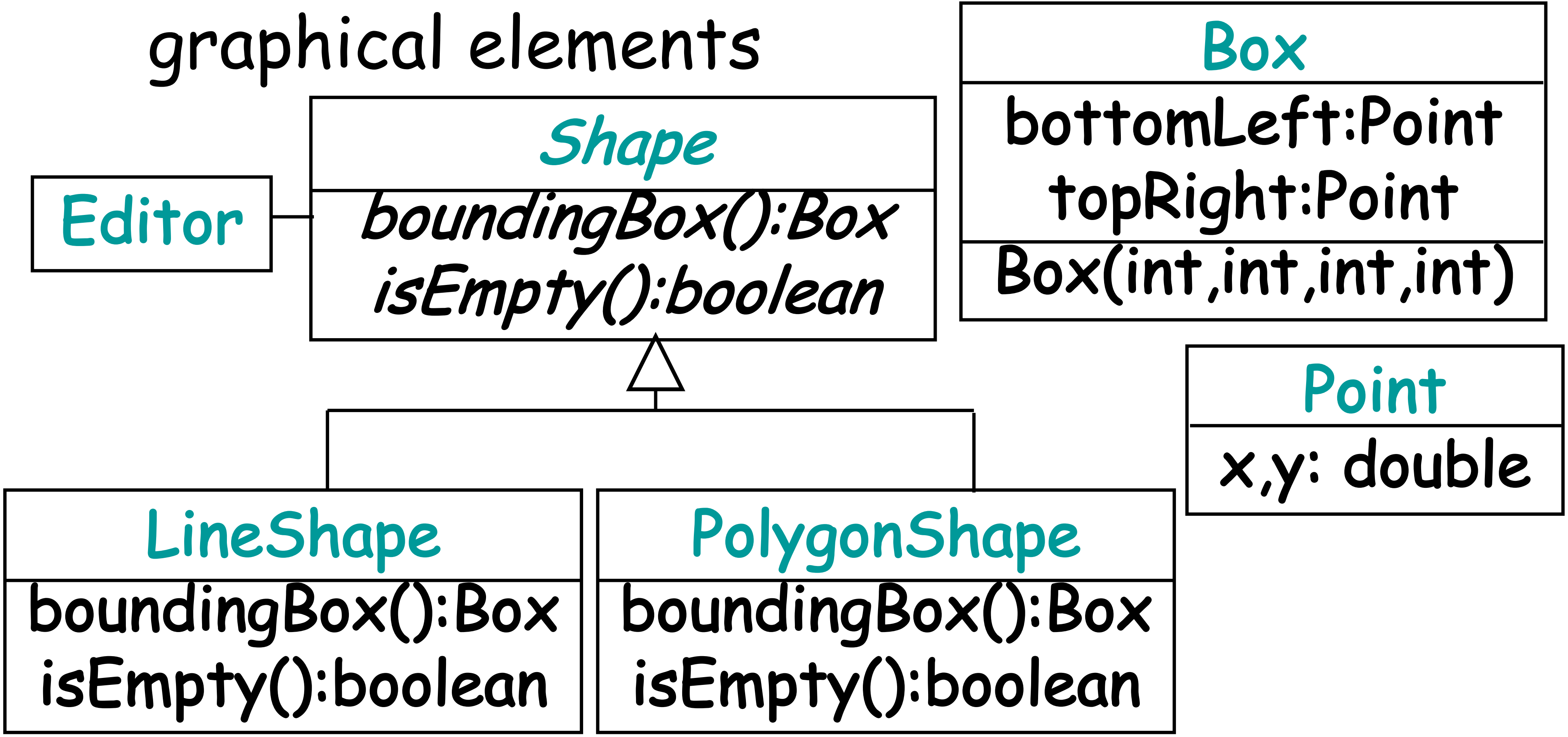


Sample Java Code

```
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); }
}
```

Another Adapter Example

- Drawing editor: diagrams built with graphical elements



Editor

Shape

boundingBox():Box

isEmpty():boolean

LineShape

boundingBox():Box

isEmpty():boolean

PolygonShape

boundingBox():Box

isEmpty():boolean

Box

bottomLeft:Point

topRight:Point

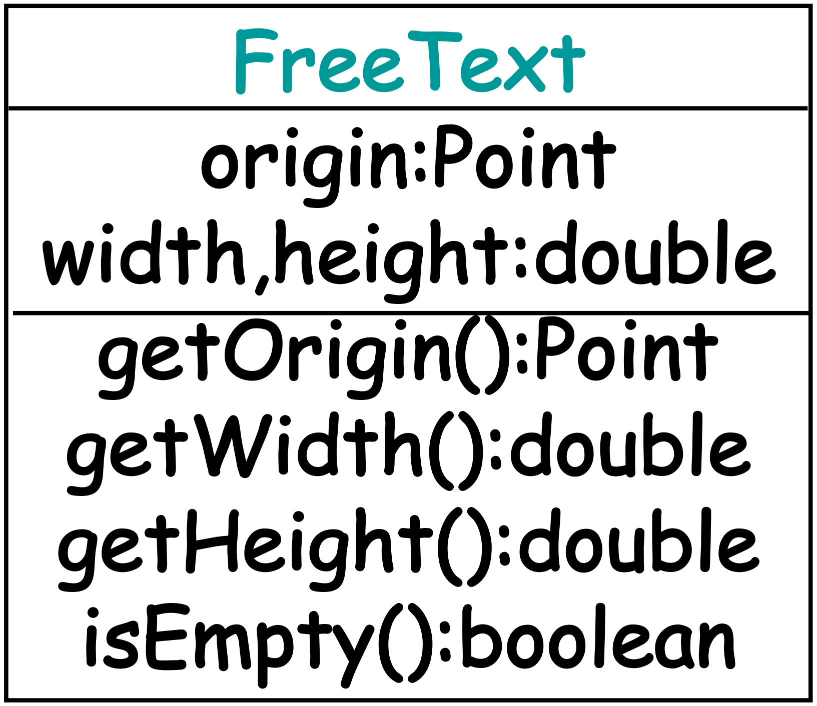
Box(int,int,int,int)

Point

x,y: double

Adding TextShape

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



FreeText

origin:Point
width,height:double

getOrigin():Point
getWidth():double
getHeight():double
isEmpty():boolean

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(y2 + t.getHeight());
    return new Box(x1,y1,x2,y2); }
   private int toInt(double) { … } }
```

Pluggable Adapters

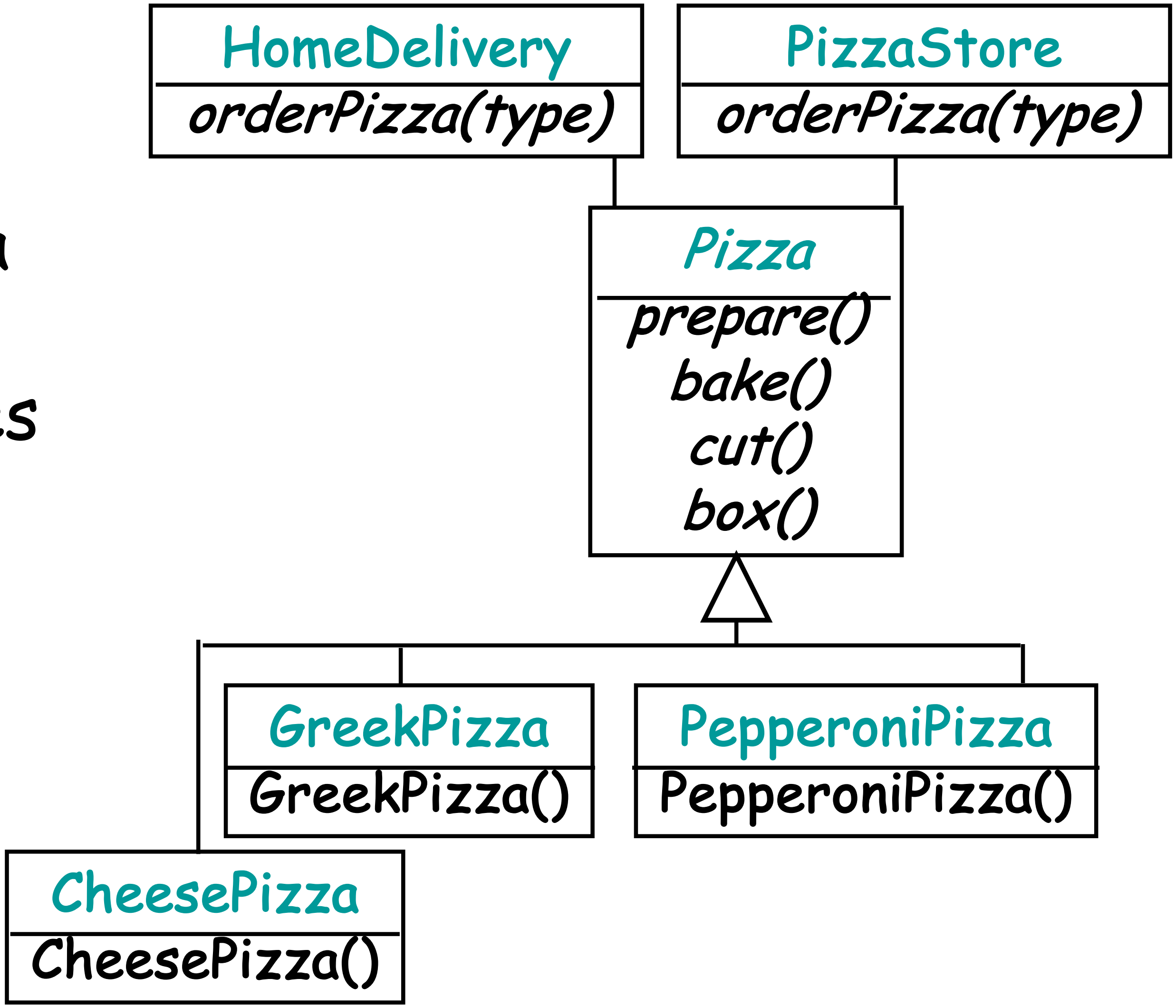
- Preparation for future adaptation
  - Define a narrow interface
- Future users of our code will write adapters to implement the interfaces
  - E.g., ITaxCalculator

Factory Pattern

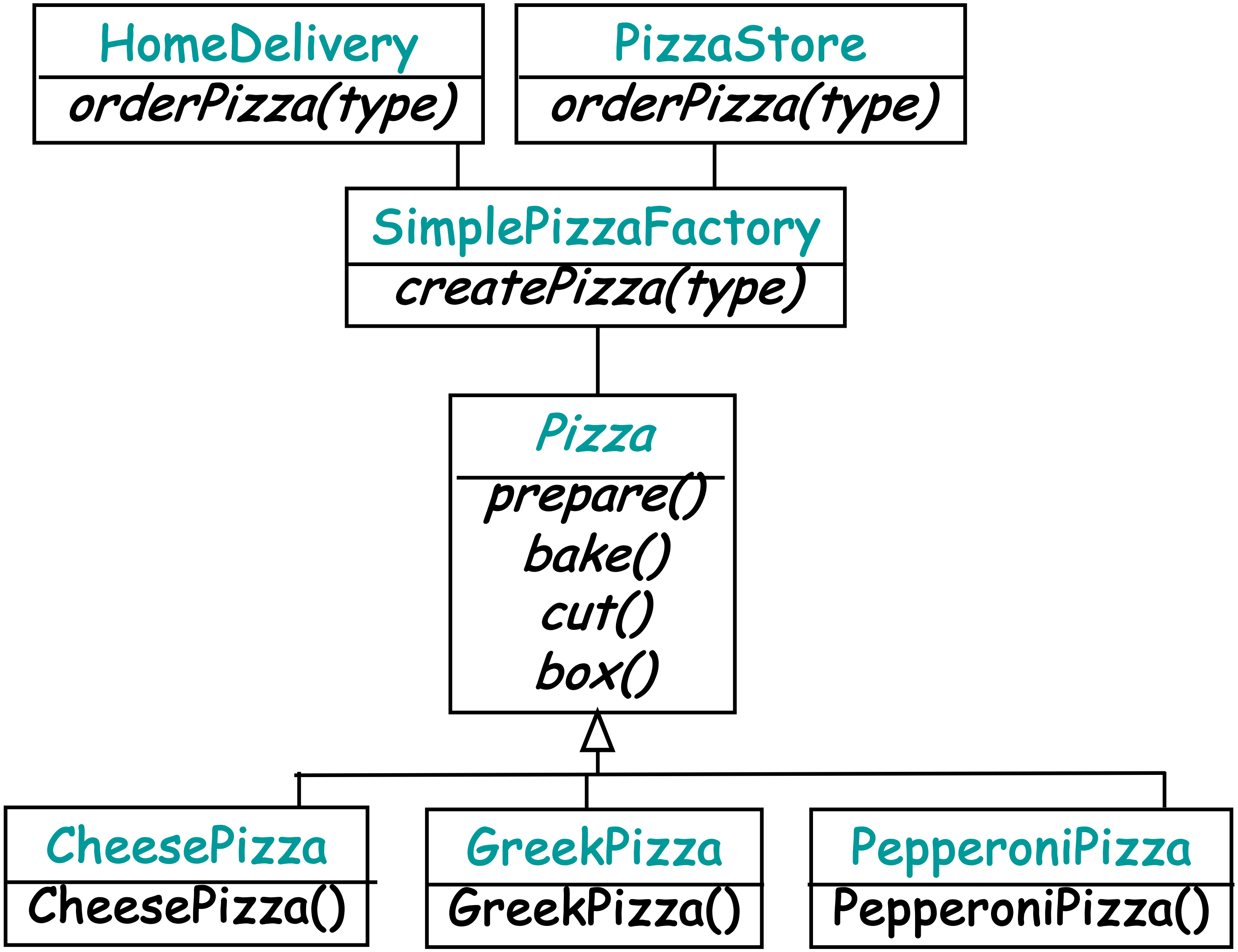
- Problem: there are many ways to create certain objects
- Solution: create a framework that is responsible for creating the objects
  - Key point: clients do not know details about object creation

Example

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



Solution: Encapsulate object creation



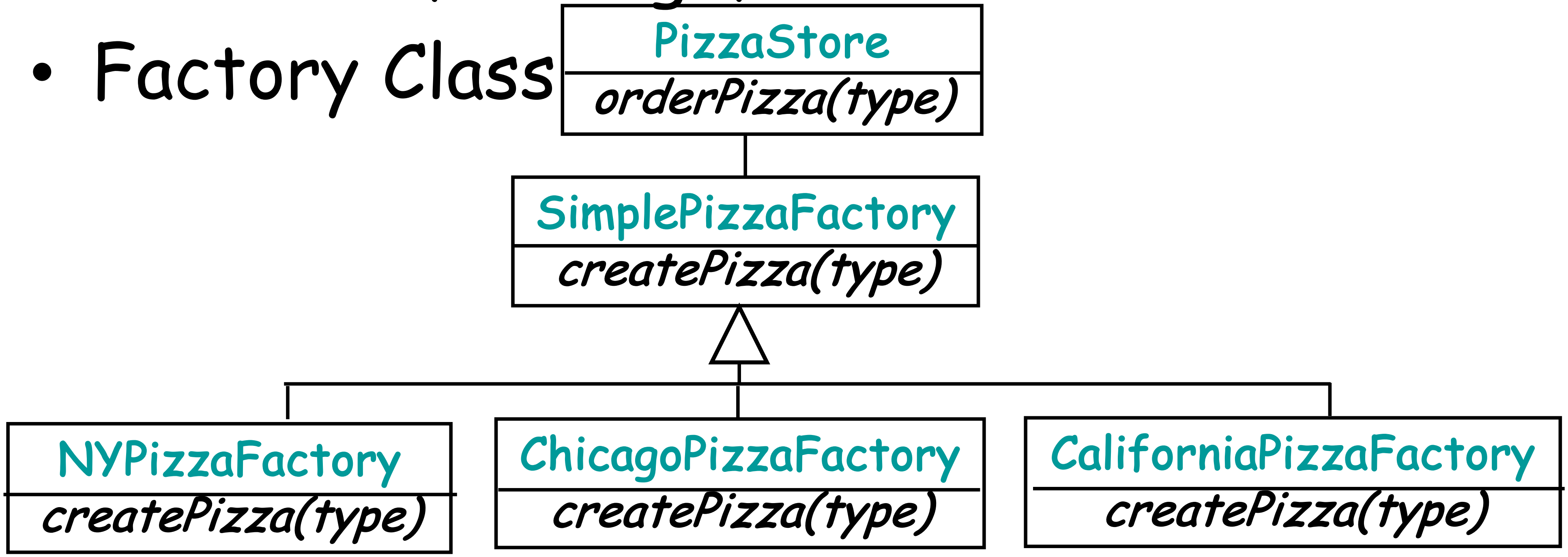
Sample Java Code

```
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;
    }
}
```

```
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;
}
```

Different Styles of Pizza?

- New York, Chicago, California
- Factory Class



PizzaStore

orderPizza(type)

SimplePizzaFactory

createPizza(type)

NYPizzaFactory

createPizza(type)

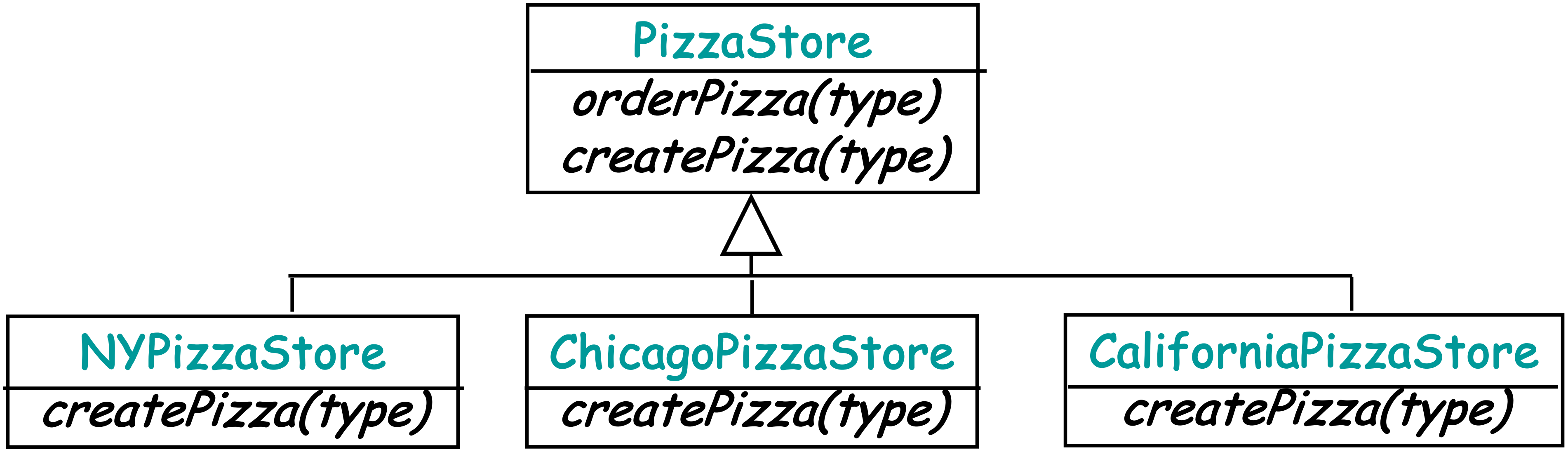
ChicagoPizzaFactory

createPizza(type)

CaliforniaPizzaFactory

createPizza(type)

An Alternative Approach: Factory Method



PizzaStore

orderPizza(type)

createPizza(type)

NYPizzaStore

createPizza(type)

ChicagoPizzaStore

createPizza(type)

CaliforniaPizzaStore

createPizza(type)

Sample Code

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

Factory Pattern

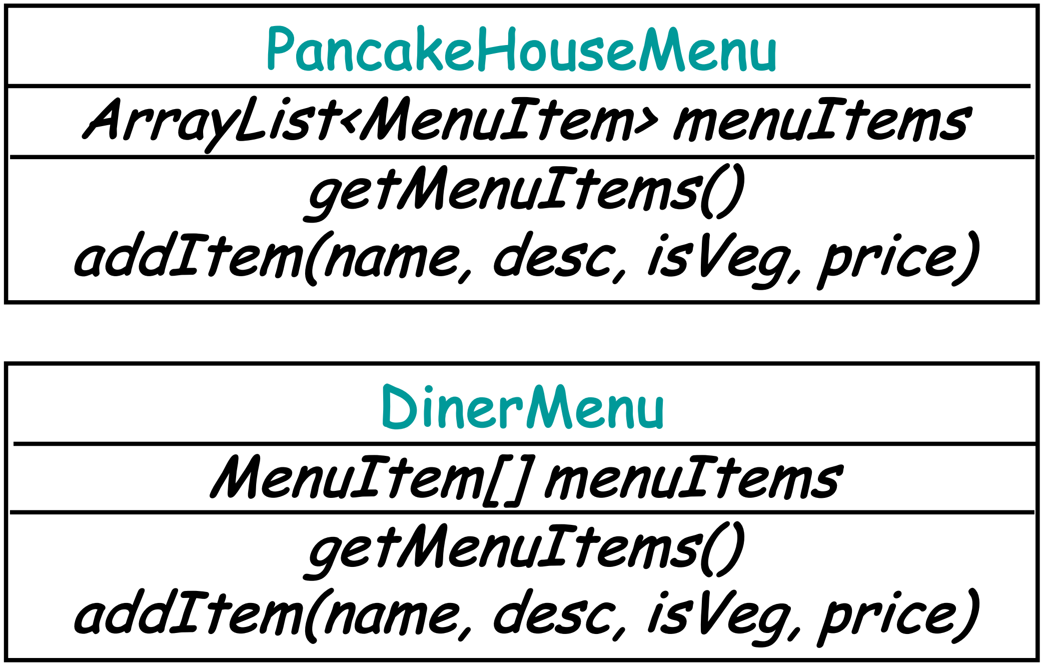
- The Dependency Inversion Principle
  - Depend upon abstractions instead of concretizations
  - Use the pattern when
    - a class cannot anticipate the class of objects it will create
    - A class wants its subclasses to specify the objects to create

Iterator Pattern

- Problem
  - There are lots of ways to stuff objects into a collection
    - Array, stack, list, hashmap, …
  - When clients want to iterate over those objects, you don’t want to expose data structure implementation

Example: Diner and Pancake House Merge

- The Pancake House menu will serve as the breakfast menu
- The Diner's menu will serve as the lunch menu



PancakeHouseMenu

ArrayList<MenuItem> menuItems

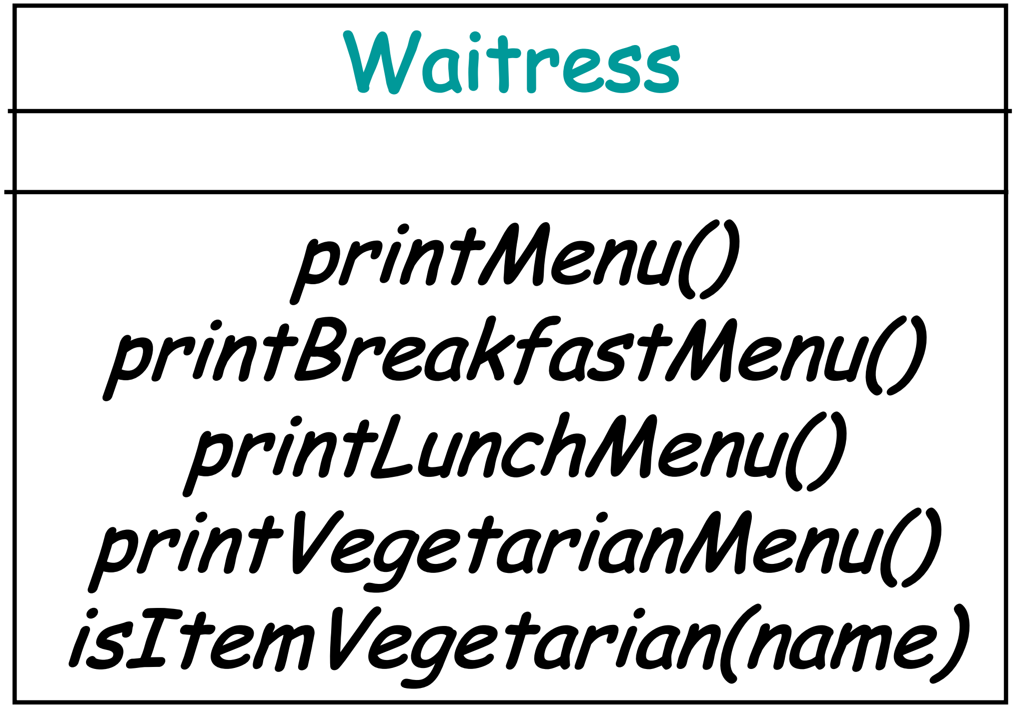
getMenuItems()
addItem(name, desc, isVeg, price)

DinerMenu

MenuItem[] menuItems

getMenuItems()
addItem(name, desc, isVeg, price)

Problem: A Java-Enabled Waitress



Waitress

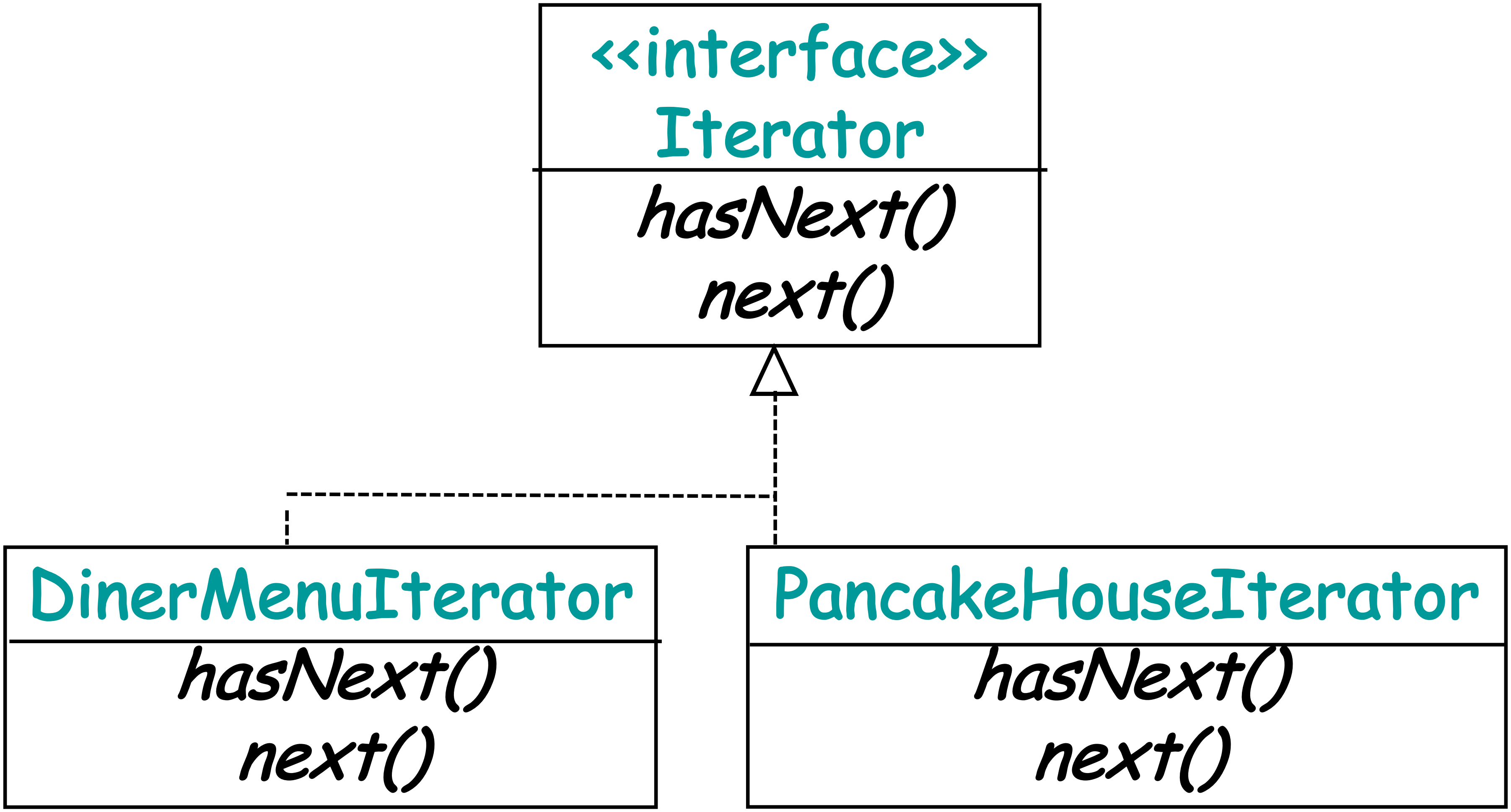
printMenu()
printBreakfastMenu()
printLunchMenu()
printVegetarianMenu()
isItemVegetarian(name)

```
printMenu() {
  for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = breakfastItems.get(i);
    System.out.print(menu.getName() + “ ”);
    System.out.println(menuItem.getPrice() + “”);
    System.out.println(menuItem.getDescription());
  }
  for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
    System.out.print(menu.getName() + “ ”);
    System.out.println(menuItem.getPrice() + “”);
    System.out.println(menuItem.getDescription());
  }
}
```

- Problem: We always need to know the internal data structure of both menus to iterate through them
- Solution: Decouple the Waitress from the concrete implementations

```
Iterator iterator = breakfastMenu.createIterator();
while (iterator.hasNext()) {
   MenuItem menuItem = iterator.next(); …
}
iterator = lunchMenu.createIterator();
while(iterator.hasNext()) {
   MenuItem menuItem = iterator.next(); …
}
```

Iterator Pattern



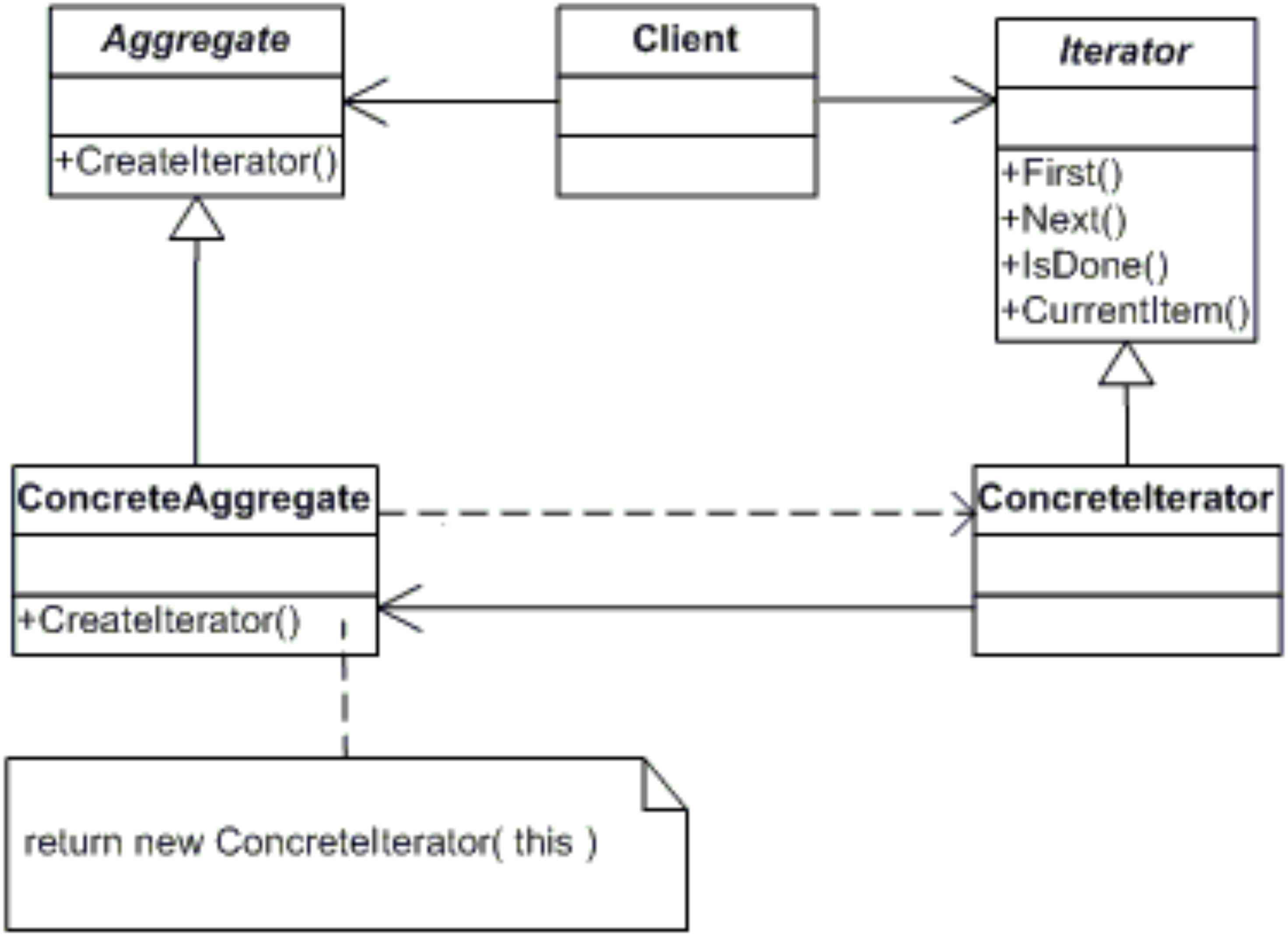
Integrate Iterator with Menus

```
public class DinerMenu {
    public Iterator createIterator() {
      return new DinerMenuIterator(menuItems);}}
public class DinerMenuIterator implements Iterator {
    MenuItem[] items;
    int position = 0;
    public DinerMenuIterator(MenuItem[] items)
    {this.items=items;}
    public MenuItem next() {
      MenuItem menuItem = items[position];
      position = position+1;
      return menuItem;}
    public boolean hasNext() {
      if (position >= items.length || items[position] == null)
         {return false;} else {return true;}}}
```

Fix up the Waitress Code

```
public void printMenu() {
    Iterator pancakeIterator =
    pancakeHouseMenu.createIterator();
    Iterator dinerIterator = dinerMenu.createIterator();
    printMenu(pancakeIterator);
    printMenu(dinerIterator);
}
private void printMenu(Iterator iterator) {
    while(iterator.hasNext()) {
        MenuItem menuItem = iterator.next();
        System.out.print(menuItem.getName() + “, ”);…}}
```

General Form



Other Examples

- java.util.ArrayList: subclass of AbstractList
- Interface java.util.Iterator
- Interface java.util.Iterable

```
public class SOList<Type> implements Iterable<Type> {
    private Type[] arrayList;
    private int currentSize;
    public SOList(Type[] newArray) {
        arrayList = newArray;
        currentSize = arrayList.length;}
    @Override public Iterator<Type> iterator() {
      Iterator<Type> it = new Iternator<Type>() {
        private int currentIndex = 0;
        @Override public boolean hasNext() {…}
        @Override public Type next() {…}
        @Override public void remove() {…}
      }; return it;}}
```