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

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

• 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

Server 1: foo()
Server 2: foo()
ZAdapter: foo()
ZServer: bar(int)

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

```java
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

<table>
<thead>
<tr>
<th>Editor</th>
<th>Shape</th>
<th>Box</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>boundingBox():Box</td>
<td>isEmpty():boolean</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|                | PluginShape     | PluginShape              |
|                | boundingBox():Box | isEmpty():boolean        |

| LineShape      | PolygonShape    |                           |
|                | boundingBox():Box | isEmpty():boolean        |

Adding TextShape

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

<table>
<thead>
<tr>
<th>FreeText</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>origin:Point</td>
<td>width,height:double</td>
</tr>
<tr>
<td>getOrigin():Point</td>
<td>getWidth():double</td>
</tr>
<tr>
<td>getHeight():double</td>
<td>isEmpty():boolean</td>
</tr>
</tbody>
</table>
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

```
PizzaStore
<table>
<thead>
<tr>
<th>orderPizza(type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HomeDelivery</td>
</tr>
<tr>
<td>orderPizza(type)</td>
</tr>
</tbody>
</table>
| Pizza
| prepare()       |
| bake()          |
| cut()           |
| box()           |
| GreekPizza()    |
| GreekPizza()    |
| PepperoniPizza()|
| PepperoniPizza()|
| CheesePizza()   |
| CheesePizza()   |
```

N. Meng, B. Ryder

10/12/15
Solution: Encapsulate object creation

```
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;
    }
}
```

---

Sample Java Code
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 Diagram:

```
PizzaStore
  orderPizza(type)
    SimplePizzaFactory
      createPizza(type)
        NYPizzaFactory
          createPizza(type)
        ChicagoPizzaFactory
          createPizza(type)
        CaliforniaPizzaFactory
          createPizza(type)
```
An Alternative Approach: Factory Method

Sample Code

```java
public abstract class PizzaStore {
    public Pizza orderPizza(String type) {
        Pizza pizza = createPizza(type);
        pizza.prepare();
        ... ...
    }
    abstract Pizza createPizza(String type);
}
```
The Original Object Dependencies

With Factory Pattern
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

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)