Lecture 21

Design Patterns 2

Announcements

• Guest Speaker Friday!!! <3
  – Kendra Yourtee

• Quiz 7 due Thursday 8/9

• Homework 8 due Thursday 8/9
  – HW8 has a regression testing component: HW5, 6, 7 tests must pass.

Outline

✓ Introduction to design patterns
✓ Creational patterns (constructing objects)
  ⇒ Structural patterns
• Behavioral patterns (affecting object semantics)
Wrappers

Structural patterns: Wrappers

Wrappers are a thin veneer over an encapsulated object
- Modify the interface
- Extend behavior
- Restrict access

We've seen this before!
- Wrappers can serve as an alternative to subtyping!

Terminology:
- Also called composition or delegation
- The wrapped or encapsulated object is called the delegate

Wrappers can change the interface or functionality of the encapsulated object.
- The encapsulated object does most of the work

Different kinds of wrappers:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Functionality</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adapter</td>
<td>same</td>
<td>different</td>
</tr>
<tr>
<td>Decorator</td>
<td>different</td>
<td>same</td>
</tr>
<tr>
<td>Proxy</td>
<td>same*</td>
<td>same</td>
</tr>
</tbody>
</table>

Some wrappers have qualities of more than one of adapter, decorator, and proxy

Adapter

- An adapter translates between incompatible interfaces
- Change an interface without changing functionality
  - Rename a method
  - Convert units
  - Implement a method in terms of another
- Example: angles passed in radians vs. degrees
- Example: use “old” method names for legacy code

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Functionality</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adapter</td>
<td>same</td>
<td>different</td>
</tr>
<tr>
<td>Decorator</td>
<td>different</td>
<td>same</td>
</tr>
<tr>
<td>Proxy</td>
<td>same*</td>
<td>same</td>
</tr>
</tbody>
</table>

*from client perspective

Client Implementation
Different interfaces

Client
Implementation
Adapter example: scaling rectangles

We have this Rectangle interface

```java
interface Rectangle {
    // grow or shrink this by the given factor
    void scale(float factor);
    ...
    float getWidth();
    float area();
}
```

Goal: client code wants to use this library to “implement” Rectangle without rewriting code that uses Rectangle:

```java
// doesn’t implement Rectangle interface
class SimpleRectangle {
    void setWidth(float width) { ... }
    void setHeight(float height) { ... }
    // no scale method
    ...
    float getWidth() { ... }
    float area() { ... }
}
```

Client interface Implementation needs adapter!

---

**Adapter: Use subclassing**

```java
class ScaleableRectangle1 
    extends SimpleRectangle 
    implements Rectangle {
    void scale(float factor) {
        setWidth(factor * getWidth());
        setHeight(factor * getHeight());
    }
    ... 
}
```

Client Implementation Adaptor

---

**Adapter: use delegation**

Delegation: forward requests to another object

```java
class ScaleableRectangle2 implements Rectangle {
    SimpleRectangle r;
    ScaleableRectangle2(float w, float h) {
        this.r = new SimpleRectangle(w, h);
    }
    void scale(float factor) {
        r.setWidth(factor * r.getWidth());
        r.setHeight(factor * r.getHeight());
    }
    float getWidth() { return r.getWidth(); }
    float area() { return r.area(); }
    ...
}
```

Client Implementation Adaptor

---

**Adapter: subclassing vs. delegation**

**Subclassing**
- automatically gives access to all methods of superclass
- built into the language (syntax, efficiency)

**Delegation**
- also known as composition
- permits “removal” of methods
- objects of arbitrary concrete classes can be wrapped
  - don’t need to worry about true subtypes
- multiple wrappers can be composed
Types of adapter

Goal of adapter: connect incompatible interfaces

- Adapter with delegation
- Adapter with subclassing: no extension is permitted

Summary: Adapter

- An adapter translates between incompatible interfaces
- Can do it by subclassing the original implementation or by delegating to an object of the original implementation

Decorator

- Add functionality without changing the interface
- Add to existing methods to do something additional – (while still preserving the previous specification)
- More flexible than subclassing alone
- Note: decorator is a lot of things.
  – what we say about one kind of decorator might not apply to other kinds of decorator

Decorator from lecture on subtyping...

```java
public class InstrumentedHashSet<E> extends HashSet<E> {
    private int addCount = 0; // count # insertions
    public InstrumentedHashSet(Collection<? extends E> c) {
        super(c);
    }
    public boolean add(E o) {
        addCount++;
        return super.add(o);
    }
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount() { return addCount; }
}
```

Problem: Unspecified whether `addAll` will call `add`. Don't know whether to increment count here.
Decorator from lecture on subtyping...

```java
public class InstrumentedHashSet<E> {
    private final HashSet<E> s = new HashSet<E>();
    private int addCount = 0;
    public InstrumentedHashSet(Collection<? extends E> c) {
        this.addAll(c);
    }
    public boolean add(E o) {
        addCount++; return s.add(o);
    }
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size(); return s.addAll(c);
    }
    public int getAddCount() { return addCount; }
    // ... and every other method specified by HashSet<E>
}
```

Decorator example: Bordered windows

```java
interface Window {
    // rectangle bounding the window
    Rectangle bounds();
    // draw this on the specified screen
    void draw(Screen s);
    ...
}

class WindowImpl implements Window {
    ...
}
```

Bordered window implementations

Via subclassing:
```java
class BorderedWindow1 extends WindowImpl {
    void draw(Screen s) {
        super.draw(s);
        bounds().draw(s);
    }
}
```

Via delegation:
```java
class BorderedWindow2 implements Window {
    Window innerWindow;
    BorderedWindow2(Window innerWindow) {
        this.innerWindow = innerWindow;
    }
    void draw(Screen s) {
        innerWindow.draw(s);
        innerWindow.bounds().draw(s);
    }
}
```

Decorators can solve inheritance issues

![Diagram showing decorated windows](image)
A decorator can remove functionality

Remove functionality without changing method signatures
– it does change the spec though

Example: UnmodifiableList
– What does it do about methods like `add` and `put`?

Problem: UnmodifiableList is a Java subtype, but not a true subtype, of List

Decoration via delegation can create a class with no Java subtyping relationship, which is often desirable

Proxy

• Same interface and functionality as the wrapped class
– So, uh, why wrap it?...
– Note that functionality here means from the client perspective.

• Control access to other objects
– Communication: manage network details when using a remote object
– Locking: serialize access by multiple clients
– Security: permit access only if proper credentials
– Creation: object might not yet exist (creation is expensive)
  • Hide latency when creating object
  • Avoid work if object is never used

Summary: Wrappers

Wrappers can change the interface or functionality of the encapsulated object.
– The encapsulated object, or delegate, does most of the work

Adapter – make client interface compatible with an implementation
Decorator – add mix-in features
Proxy – add hidden functionality so the client can use the same abstraction for different objects

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Functionality</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adapter</td>
<td>same</td>
<td>different</td>
</tr>
<tr>
<td>Decorator</td>
<td>different</td>
<td>same</td>
</tr>
<tr>
<td>Proxy</td>
<td>same*</td>
<td>same</td>
</tr>
</tbody>
</table>

*from client perspective

Composite
Composite pattern

- Composite permits a client to manipulate either an atomic unit or a hierarchical collection of units in the same way
  - So no need to “always know” if an object is a collection of smaller objects or not
- Good for dealing with “part-whole” relationships
- **Different from composition!**
  - composition is simply where any object (the wrapper) has another object (the delegate) as an instance field and calls on the delegate to execute certain operations
  - composite pattern involves dealing with object hierarchies as though they were atomic units

- An extended example...

**Composite example: Bicycle**

```java
abstract class BicycleComponent {
    public int weight();
    public float cost();
}

class Skewer extends BicycleComponent {
    private float price;
    public float cost() { return price; }
}

class Wheel extends BicycleComponent {
    private float assemblyCost;
    private Skewer skewer;
    private Hub hub;
    ...
    public float cost() {
        return assemblyCost + skewer.cost() + hub.cost() + ...;
    }
}
```

Supports polymorphism for individual parts and hierarchies of parts!

**Composite example: Libraries**

```java
interface Text {
    String getText();
}

class Page implements Text {
    String getText() {
        ... return concatenation of column texts ...
    }
}
```

Library

- Section (for a given genre)
- Shell
- Volume
- Page
- Column
- Word
- Letter

Supports polymorphism for individual parts and hierarchies of parts!
**Composite example: Abstract Syntax Tree**

\[ x = \text{foo} \times b + c / d; \]

**Abstract syntax tree for Java code**

class `PlusOp` extends Expression {  // + operation
    Expression leftExp;
    Expression rightExp;
}
class `VarRef` extends Expression {  // variable use
    String varname;
}
class `GreaterOp` extends Expression {  // test a > b
    Expression leftExp;
    Expression rightExp;
}
class `CondOp` extends Expression {  // a ? b:c
    Expression testExp;
    Expression thenExp;
    Expression elseExp;
}

**Object model vs. type hierarchy**

- AST for `a + b`:

  ![AST diagram for a + b]

- Class hierarchy for `Expression`:

  ![Class hierarchy diagram]

**Summary: Composite pattern**

- Manipulate either an atomic unit or a hierarchical collection of units in the same way

- Different from composition!
  - composition is simply where any object (the wrapper) has another object (the delegate) as an instance field and calls on the delegate to execute certain operations
  - composite pattern involves dealing with object hierarchies as though they were atomic units
Outline

✓ Introduction to design patterns
✓ Creational patterns (constructing objects)
✓ Structural patterns
⇒ Behavioral patterns (affecting object semantics)
  – Already seen: Observer / Listeners
  – Will just do 2-3 related ones

Traversing composites

• Goal: perform operations on all parts of a composite

• Idea: generalize the notion of an iterator – process the components of a composite in an order appropriate for the application

• Example: arithmetic expressions in Java (abstract syntax trees)
  – How do we traverse/process these expressions?

Operations on abstract syntax trees

Suppose we are writing a compiler!
Need to write code for each entry in this table

<table>
<thead>
<tr>
<th>Types of Objects</th>
<th>CondExpr</th>
<th>EqualOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>typecheck</td>
<td></td>
<td></td>
</tr>
<tr>
<td>print</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Question: Should we group together the code for a particular operation or the code for a particular expression?
  – That is, do we group the code into rows or columns?

• Given an operation and an expression, how do we “find” the proper piece of code?
Interpreter and procedural patterns

**Interpreter pattern**

Add a method to each class for each supported operation

```java
abstract class Expression {
    ...
    Type typecheck();
    String print();
}
```

```java
class EqualOp extends Expression {
    ...
    Type typecheck() {
        ...
    }
}
```

```java
class CondOp extends Expression {
    ...
    Type typecheck() {
        ...
    }
}
```

Dynamic dispatch chooses the right implementation, for a call like `e.typeCheck()`

Overall type-checker spread across classes

**Procedural pattern**

Create a class per operation, with a method per operand type

```java
class Typecheck {
    ...
    Type typeCheckExpr(Expression e) {
        if (e instanceof PlusOp) {
            return typeCheckPlusOp((PlusOp)e);
        } else if (e instanceof VarRef) {
            return typeCheckVarRef((VarRef)e);
        } else if (e instanceof EqualOp) {
            return typeCheckEqualOp((EqualOp)e);
        } else if (e instanceof CondExpr) {
            return typeCheckCondExpr((CondExpr)e);
        } else ...
        ...
    }
}
```

How to invoke the right method for an expression `e`?

(See also many offerings of CSE341 for an extended take on this question

- Statically typed functional languages help with procedural whereas statically typed object-oriented languages help with interpreter)
**Definition of `typeCheckExpr` (using procedural pattern)**

```java
class Typecheck {
    ...
    Type typeCheckExpr(Expression e) {
        if (e instanceof PlusOp) {
            return typeCheckPlusOp((PlusOp)e);
        } else if (e instanceof VarRef) {
            return typeCheckVarRef((VarRef)e);
        } else if (e instanceof EqualOp) {
            return typeCheckEqualOp((EqualOp)e);
        } else if (e instanceof CondExpr) {
            return typeCheckCondExpr((CondExpr)e);
        } else {
            // Need similar code for each operation
        }
    }
    ...
}
```

• Maintaining this code is tedious and error-prone
  • No help from type-checker to get all the cases
    (unlike in functional languages)
  • Cascaded if tests are likely to run slowly (in Java)

**Visitor pattern:** A variant of the procedural pattern

• Nodes (objects in the hierarchy) accept visitors for traversal
• Visitors visit nodes (objects)

```java
class SomeExpression extends Expression {
    void accept(Visitor v) {
        for each child of this node {
            child.accept(v);
        }
        v.visit(this);
    }
}
class SomeVisitor extends Visitor {
    void visit(SomeExpression n) {
        perform work on n;
    }
}
```

**Example: accepting visitors**

```java
class VarOp extends Expression {
    ...
    void accept(Visitor v) {
        v.visit(this);
    }
}
class EqualsOp extends Expression {
    ...
    void accept(Visitor v) {
        leftExp.accept(v);
        rightExp.accept(v);
        v.visit(this);
    }
}
class CondOp extends Expression {
    ...
    void accept(Visitor v) {
        testExp.accept(v);
        thenExp.accept(v);
        elseExp.accept(v);
        v.visit(this);
    }
}
```

First visit all children
Then pass "self" back to visitor
The visitor has a `visit` method for each kind of expression, thus picking the right code for this kind of expression
• Overloading makes this look more magical than it is...

Lets clients provide unexpected visitors

**Sequence of calls to accept and visit**

```java
a.accept(v)
b.accept(v)
d.accept(v)
e.accept(v)
c.accept(v)
f.accept(v)
```

Sequence of calls to visit:  d, e, b, f, c, a
Example: Implementing visitors

```java
class TypeCheckVisitor implements Visitor {
    void visit(VarOp e) { ... }
    void visit(EqualsOp e) { ... }
    void visit(CondOp e) { ... }
}

class PrintVisitor implements Visitor {
    void visit(VarOp e) { ... }
    void visit(EqualsOp e) { ... }
    void visit(CondOp e) { ... }
}
```

Behavioral patterns: Summary

**Interpreter Pattern:** group operations by object
- Java is well suited for it

**Procedural Pattern:** group operations by operation
- awkward in Java

**Visitor Pattern:** Variant of procedural pattern that uses double dispatch to alleviate some of the awkwardness
- code is grouped by operation in visitors
- each class in the hierarchy implements accept method