Lecture 21

Design Patterns 2

Zach Tatlock / Spring 2018
Outline

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

A wrapper translates between incompatible interfaces.

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

The encapsulated class does most of the work.

<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

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
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
class NonScaleableRectangle { // not a Rectangle
    void setWidth(float width) { ... }
    void setHeight(float height) { ... }
    // no scale method
    ...
}
```
Adapter: Use subclassing

class ScaleableRectangle1
    extends NonScaleableRectangle
    implements Rectangle {
    void scale(float factor) {
        setWidth(factor * getWidth());
        setHeight(factor * getHeight());
    }
}
Adapter: use delegation

Delegation: forward requests to another object

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

Subclassing vs. delegation

Subclassing
– automatically gives access to all methods of superclass
– built in to the language (syntax, efficiency)

Delegation
– permits removal of methods (compile-time checking)
– objects of arbitrary concrete classes can be wrapped
– multiple wrappers can be composed

Delegation vs. composition
– Differences are subtle
– For CSE 331, consider them equivalent (?)
Types of adapter

Goal of adapter:
connect incompatible interfaces

Adapter with delegation

Adapter with subclassing:
no extension is permitted
Decorator

- Add functionality without changing the interface
- Add to existing methods to do something additional
  - (while still preserving the previous specification)
- Not all subclassing is decoration
Decorator example: Bordered windows

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:

class BorderedWindow1 extends WindowImpl {
    void draw(Screen s) {
        super.draw(s);
        bounds().draw(s);
    }
}

Via delegation:

class BorderedWindow2 implements Window {
    Window innerWindow;
    BorderedWindow2(Window innerWindow) {
        this.innerWindow = innerWindow;
    }
    void draw(Screen s) {
        innerWindow.draw(s);
        innerWindow.bounds().draw(s);
    }
}
A decorator can remove functionality

Remove functionality without changing the interface

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?...

• 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
Composite pattern

- Composite permits a client to manipulate either an atomic unit or a 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

- An extended example…
Composite example: Bicycle

- Bicycle
  - Wheel
    - Skewer
      - Lever
      - Body
      - Cam
      - Rod
  - Hub
  - Spokes
  - Nipples
  - Rim
  - Tape
  - Tube
  - Tire
- Frame
- Drivetrain
- ...
abstract class BicycleComponent {
    int weight();
    float cost();
}

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

class Wheel extends BicycleComponent {
    float assemblyCost;
    Skewer skewer;
    Hub hub;
    ...
    float cost() {
        return assemblyCost + skewer.cost()
                   + hub.cost() + ...;
    }
}
Composite example: Libraries

Library
  Section (for a given genre)
    Shelf
      Volume
        Page
          Column
            Word
                Letter

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

Outline

✓ Introduction to design patterns
✓ Creational patterns (constructing objects)
✓ Structural patterns (controlling heap layout)
⇒ Behavioral patterns (affecting object semantics)
  – Already seen: Observer
  – 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
  - How do we represent, say, \( x=\text{foo}*b+c/d; \)
  - How do we traverse/process these expressions?
Representing Java code

\[ x = \text{foo} \ast b + c \div d; \]
Abstract syntax tree (AST) for Java code

class PlusOp extends Expression { // + operation
    Expression leftExp;
    Expression rightExp;
}
class VarRef extends Expression { // variable use
    String varname;
}
class EqualOp extends Expression { // test a==b;
    Expression leftExp; // left-hand side: a in a==b
    Expression rightExp; // right-hand side: b in a==b
}
class CondExpr extends Expression { // a?b:c
    Expression testExp;
    Expression thenExp;
    Expression elseExp;
}
Object model vs. type hierarchy

- AST for $a + b$:

```
(PlusOp)
  a (VarRef)
  b (VarRef)
```

- Class hierarchy for Expression:

```
Expression
  PlusOp
  VarRef
  EqualOp
  CondExpr
```
Operations on abstract syntax trees

Need to write code for each entry in this table

<table>
<thead>
<tr>
<th>Operations</th>
<th>Types of Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CondExpr</td>
</tr>
<tr>
<td></td>
<td>typecheck</td>
</tr>
<tr>
<td></td>
<td>print</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:** collects code for similar **objects**, spreads apart code for similar operations
  - Makes it easy to add types of objects, hard to add operations
  - An instance of the **Composite** pattern

**Procedural:** collects code for similar **operations**, spreads apart code for similar objects
  - Makes it easy to add operations, hard to add types of objects
  - The **Visitor** pattern is a variety of the procedural pattern

(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)
Interpreter pattern

Add a method to each class for each supported operation

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

class EqualOp extends Expression {
  ...
  Type typecheck() { ... }
  String print() { ... }
}

class CondExpr extends Expression {
  ...
  Type typecheck() { ... }
  String print() { ... }
}
Procedural pattern

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

class Typecheck {
    Type typeCheckCondExpr(CondExpr e) {
        Type condType = typeCheckExpr(e.condition);
        Type thenType = typeCheckExpr(e.thenExpr);
        Type elseType = typeCheckExpr(e.elseExpr);
        if (condType.equals(BoolType) &&
            thenType.equals(elseType))
            return thenType;
        else
            return ErrorType;
    }
    Type typeCheckEqualOp(EqualOp e) {
        ......
    }
}
Definition of `typeCheckExpr` (using procedural pattern)

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 ...
    ...
    }

    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)

    Need similar code for each operation
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
    }
}
```

`n.accept(v)` traverses the structure rooted at `n`, performing `v`'s operation on each element of the structure.
Example: accepting visitors

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);
    }
}
Sequence of calls to accept and visit

a.accept(v)  
b.accept(v)  
d.accept(v)  
  v.visit(d)  
e.accept(v)  
  v.visit(e)  
  v.visit(b)  
c.accept(v)  
f.accept(v)  
  v.visit(f)  
  v.visit(c)  
v.visit(a)

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

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) { ... }
}

Now each operation has its cases back together

And type-checker should tell us if we fail to implement an abstract method in Visitor

Again: overloading just a nicety

Again: An OOP workaround for procedural pattern
- Because language/type-checker is not instance-of-test friendly