Design patterns (part 2)

CSE 331
Spring 2010
Outline

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

The 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>
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
Adapter example: scaling rectangles

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

class myClass {
    void myMethod(Rectangle r) {
        ...   r.scale(2);   ...
    }
}

Goal: be able to use this class instead:

class NonScaleableRectangle { // not a Rectangle
    void setWidth(float width) { ... } // no scale method
    void setHeight(float height) { ... } // not a Rectangle
    ...
}
```
Adapting scaled rectangles via subclassing

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

Adapting scaled rectangles via delegation

Delegation: forward requests to another object

class ScaleableRectangle2 implements Rectangle {
    NonScaleableRectangle r;
    ScaleableRectangle2(NonScaleableRectangle r) {
        this.r = r;
    }

    void scale(float factor) {
        setWidth(factor * r.getWidth());
        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 into the language (syntax, efficiency)

Delegation
- permits cleaner removal of methods (compile-time checking)
- wrappers can be added and removed dynamically
- objects of arbitrary concrete classes can be wrapped
- multiple wrappers can be composed

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

Delegation vs. composition
Differences are subtle
For CSE 331, consider them to be 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

```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);
    }
}
```

Delegation permits multiple borders on a window, or a window that is both bordered and shaded (or either one of those)
Proxy

Same interface and functionality as the wrapped class

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

• Good for dealing with part-whole relationships
Composite example: Bicycle

- Bicycle
  - Wheel
    - Skewer
    - Hub
    - Spokes
    - Nipples
    - Rim
    - Tape
    - Tube
    - Tire
  - Frame
  - Drivetrain
  - ...

Methods on components

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 the concatenation of the column texts ...
    }
}
Traversing composites

Goal: perform operations on all parts of a composite
Abstract syntax tree (AST) for Java code

class PlusOp extends Expression { // + operation
    Expression leftExp;
    Expression rightExp;
}
class VarRef extends Expression { // variable reference
    String varname;
}
class EqualOp extends Expression { // equality test a==b;
    Expression lvalue; // left-hand side; "a" in "a==b"
    Expression rvalue; // right-hand side; "b" in "a==b"
}
class CondExpr extends Expression { // a?b:c
    Expression condition;
    Expression thenExpr; // value of expression if a is true
    Expression elseExpr; // value of expression if a is false
}
Object model
vs. module dependence diagram

• AST for "a + b":

• Class hierarchy for Expression:
Perform operations on abstract syntax trees

Need to write code in each of the cells of this table:

<table>
<thead>
<tr>
<th>Operations</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CondExpr</td>
</tr>
<tr>
<td>typecheck</td>
<td></td>
</tr>
<tr>
<td>pretty-print</td>
<td></td>
</tr>
</tbody>
</table>

Question: Should we group together the code for a particular operation or the code for a particular expression?

(A separate issue: given an operation and an expression, how to select 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 objects, hard to add operations

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

Both interpreter and procedural have classes for objects
  The code for operations is similar
  The question is where to place that code

Selecting between interpreter and procedural:
  Are the algorithms central, or are the objects?
    (Is the system operation-centric or object-centric?)
  What aspects of the system are most likely to change?
Interpreter pattern

Add a method to each class for each supported operation

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

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

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

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

class Typecheck {
   // typecheck "a?b:c"
   Type tcCondExpr(CondExpr e) {
      Type condType = tcExpression(e.condition); // type of "a"
      Type thenType = tcExpression(e.thenExpr);  // type of "b"
      Type elseType = tcExpression(e.elseExpr);  // type of "c"
      if ((condType == BoolType) && (thenType == elseType)) {
         return thenType;
      } else {
         return ErrorType; }
   } // typecheck "a==b"
   Type tcEqualOp(EqualOp e) {
      ...  
   }
}
Definition of tcExpression
(in procedural pattern)

```java
class Typecheck {
    ...
    Type tcExpression(Expression e) {
        if (e instanceof PlusOp) {
            return tcPlusOp((PlusOp)e);
        } else if (e instanceof VarRef) {
            return tcVarRef((VarRef)e);
        } else if (e instanceof EqualOp) {
            return tcEqualOp((EqualOp)e);
        } else if (e instanceof CondExpr) {
            return tcCondExpr((CondExpr)e);
        } else ...
        ...
        }
    }
```

Maintaining this code is tedious and error-prone.
The cascaded if tests are likely to run slowly.
This code must be repeated in PrettyPrint and every other operation class.
Visitor pattern: a variant of the procedural pattern

Visitor encodes a traversal of a hierarchical data structure
Nodes (objects in the hierarchy) accept visitors
Visitors visit nodes (objects)

```java
class Node {
    void accept(Visitor v) {
        for each child of this node {
            child.accept(v);
        }
        v.visit(this);
    }
}

class Visitor {
    void visit(Node n) {
        perform work on n
    }
}
```

`n.accept(v)` performs a depth-first traversal of the structure rooted at `n`, performing `v`'s operation on each element of the structure
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
Implementing visitor

- You must add definitions of `visit` and `accept`.

- `visit` might count nodes, perform typechecking, etc.

- It is easy to add operations (visitors), hard to add nodes (modify each existing visitor).

- Visitors are similar to iterators: each element of the data structure is presented in turn to the `visit` method:
  - Visitors have knowledge of the structure, not just the sequence.
Calls to visit cannot communicate with one another

Can use an auxiliary data structure
Another solution: move more work into the visitor itself

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

Information flow is clearer (if visitor depends on children)
Traversal code repeated in all visitors (acceptor is extraneous)