CSE 331
Software Design & Implementation

Hal Perkins

Winter 2012

Design Patterns Part 2

(Slides by David Notkin and Mike Ernst)

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

Pattern	Functionality	Interface
Adapter	same	different
Decorator	different	same
Proxy	same	same

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

- We have this **`Rectangle`** interface

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

- Goal: we want to use instances of this class to “implement” **`Rectangle`**:

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

Adaptor: Use subclassing

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

Adaptor: use delegation

Delegation: forward requests to another object

```
class ScaleableRectangle2 implements Rectangle {
  NonScaleableRectangle r;
  ScaleableRectangle2(w,h) {
    this.r = new NonScaleableRectangle(w,h);
  }

  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

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 subclasssing:

```
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);
  }
}
```

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

A decorator can remove functionality

- Remove functionality without changing the interface
- Example: **`UnmodifiableList`**
  - What does it do about methods like add and put?

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

Visitor pattern: Traversing composite objects

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

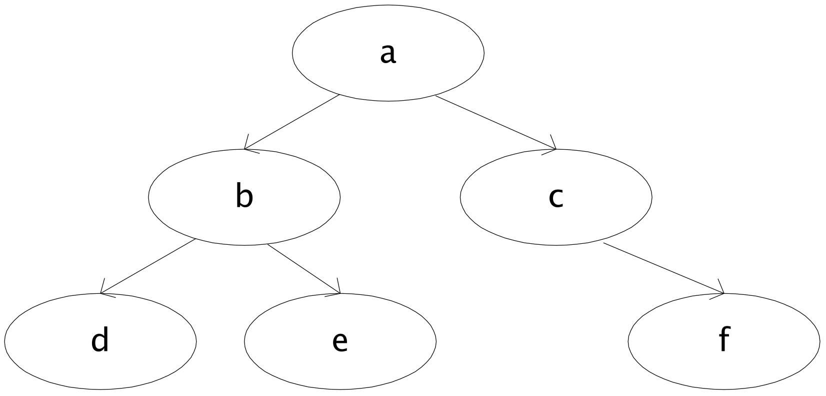
```
class Node {
  void accept(Visitor v) {
    for each child of this node {
      child.accept(v);
    }
    v.visit(this);
  }
}
class Visitor {
  void visit(Node n) {
    perform work on n
  }
}
```

`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