Design patterns (part 2)

CSE 331
University of Washington

Michael 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

Library:

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

Client:

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

Goal: enable MyClass to use this library (without rewriting MyClass):

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

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) {
    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 (with 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

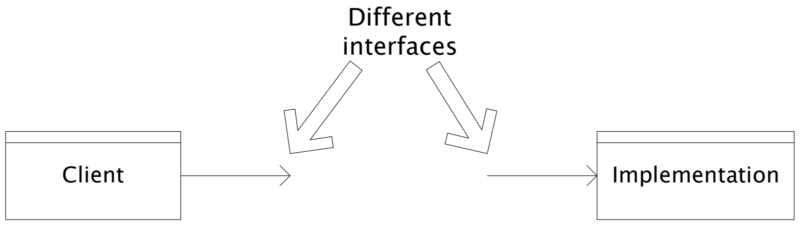
Delegation vs. composition

Differences are subtle

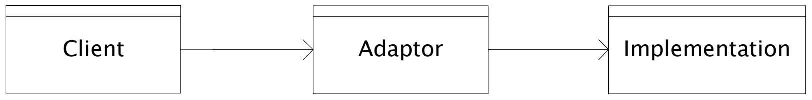
For CSE 331, consider them to be equivalent

Types of adapter

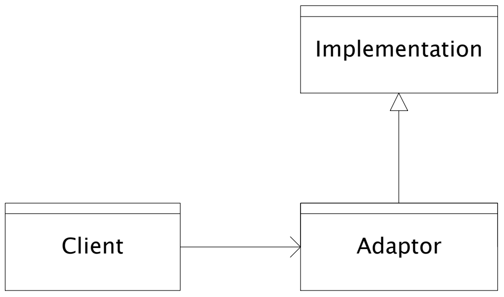
Goal of adapter:
connect incompatible interfaces



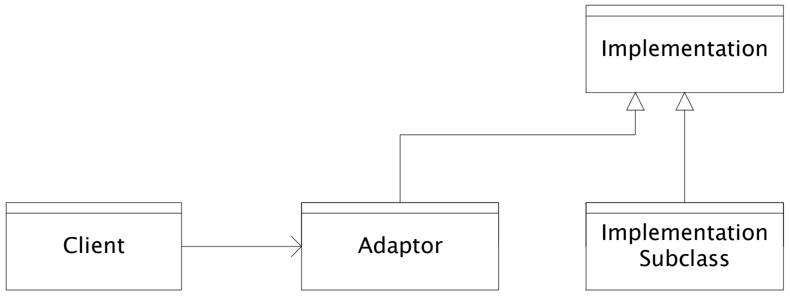
Adapter with delegation



Adapter with subclassing



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 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?

- Problem: UnmodifiableList is a Java subtype, but not a true subtype, of List
- Decoration can create a class with no Java subtyping relationship, which is desirable

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
      - Lever
      - Body
      - Cam
      - Rod
      - Acorn nut
    - 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 ...
  }
}
```

Next time: Traversing composites

Goal: perform operations on all parts of a composite