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

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

Change an interface without changing functionality

- rename a method
- convert units
- implement a method in terms of another

Examples:

- angles passed in radians vs. degrees
- use old method names for legacy code

Adapter example: scaling rectangles

Library:

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

Client:

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

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

```
class RectangleImpl { // not an IRectangle
  void setWidth(float width) { ... }
  void setHeight(float height) { ... }
   // no scale method
  ...
}
```

Two ways to do it:

- Subclassing
- Delegation

Adapting scaled rectangles via subclassing

```
class RectangleImplSC extends RectangleImpl
                          implements IRectangle {
  void scale(float factor) {
    setWidth(factor * getWidth());
    setHeight(factor * getHeight());
  }
}
```

Adapting scaled rectangles via delegation

Delegation: forward requests to another object

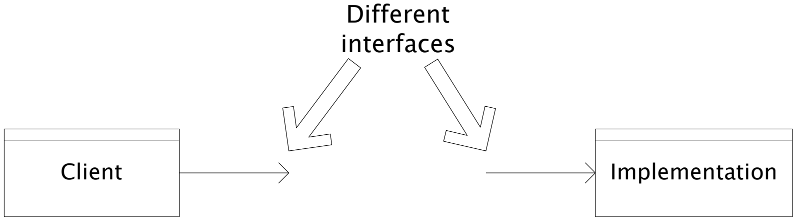
```
class RectangleImplD implements IRectangle {
  RectangleImpl r;
  RectangleImplD(RectangleImpl r) {
    this.r = r.clone();
  }

  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

Goal of adapter:
connect incompatible interfaces


Different interfaces

Client

Implementation

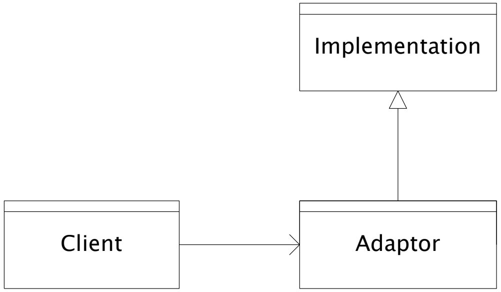
Adapter with delegation:
Can reference any subtype at run time


Client

Adaptor

Implementation

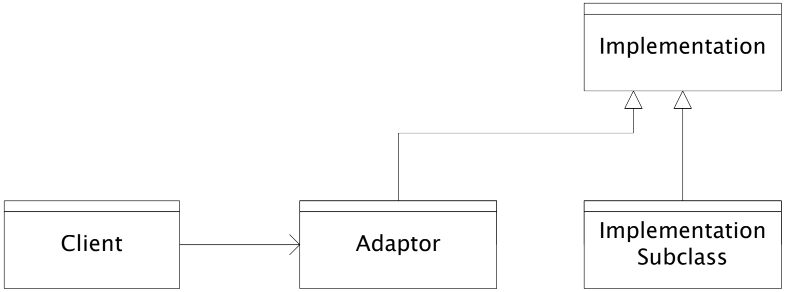
Adapter with subclassing


Implementation

Client

Adaptor

Adapter with subclassing:
No extension is permitted


Implementation

Client

Adaptor

Implementation Subclass

Decorator

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

Add functionality without changing the interface

Make existing methods do more

- 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);
  }
}
```

Advantages of delegation:

- A window can have multiple borders
- A window can have any combination of borders, shading, …
- Wrappers can be added and removed dynamically

A decorator can remove functionality

Remove functionality without changing the interface

Example: **`UnmodifiableList`**

What does it do about mutators 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 when removing functionality (if an interface exists)

Proxy

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

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

Subclassing vs. delegation

Subclassing

- automatically gives access to all methods of superclass
- built in to the language (syntax, efficiency)
- if this meets your needs, use it

Delegation

- permits removal of methods (with compile-time checking)
- 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

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
  - Frame
  - Drivetrain
  - Wheel
    - Tire
    - Tube
    - Tape
    - Rim
    - Nipples
    - Spokes
    - Hub
    - Skewer
      - Lever
      - Body
      - Cam
      - Rod
      - Acorn nut
  - ...

Methods on components

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

```
class Bicycle
    extends BicycleComponent {
  float assemblyCost;
  Frame frame;
  Drivetrain drivetrain;
  Wheel frontWheel;
  ...
  float cost() {
    return assemblyCost
           + frame.cost()
           + drivetrain.cost()
           + frontWheel.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