LECTURE 19: DESIGN PATTERNS

Software Engineering

Mike Wooldridge

1 Introduction

- Recall that one goal of software engineering is *reuse*: instead of designing and implementing all software from scratch, make use of existing, developed software.
- *Design patterns* are about reuse in design.
- The idea is that certain stereotypical design problems arise again and again.
- Rather than develop solutions anew every time we are presented with a problem, we can make use of *design patterns* that others have used.
- We will look at the following design patterns:
  - interfaces & templates;
  - observers;
  - proxies.

2 Interfaces

- Suppose we have classes `Square`, `Circle`, `Triangle`, `Hexagon`, and so on for implementing shapes.

  We want to build a vector of shapes, and we want to process the vector to get the total area of all shapes.

  This is what we *want* to write.

```
Vector v = new Vector();
// add shapes to the vector
...
float totalArea = 0.0;
Enumeration e = v.elements();
while(e.hasMoreElements()) {
  totalArea += e.nextElement().area();
}
```

  This will cause a problem, since the compiler does not know that each object in the array implements the `area()` method.

- One possibility: define a class `Shape`, with a default method `area()`, and get each sub-class (`Triangle`, `Square`, ...) to overwrite this method.

  But what if the method is *not* overwritten?

- Solution is to implement an *interface* for shapes.
- The interface a number of methods, without providing an implementation for them.
- Any class that *implements* the interface *must* provide an implementation for these methods.

- Here is the interface for `Shape`s:

```
public interface Shape {
  public float area() ;
} // end interface Shape
```

- And here is how we state that `Rectangle` implements the `Shape` interface:

```
public class Rectangle implements Shape {
  ...
  public float area() {
    return (float)(width * height);
  }
  ...
} // end class Rectangle
```

- Similarly for `Circle`s:

```
public class Circle implements Shape {
  ...
  public float area() {
    return Math.PI * (radius * radius);
  }
  ...
} // end class Circle
```

- The code to process the vector of `Shape`s is then:

```
Vector v = new Vector();
// add shapes to the vector
...
float totalArea = 0.0;
Enumeration e = v.elements();
while(e.hasMoreElements()) {
  totalArea += ((Shape)e.nextElement()).area();
}
```

- We cast the `Object` returned by `nextElement()` to a `Shape` — the compiler knows that any class implementing `Shape` has a `float` valued method `area()`, and so is happy.
- This means that at *design* time, we can write code that needn't worry about the *implementation* of any class that implements `Shape`.

  We can treat the implementation as a black box, and rest safe in the knowledge that it must provide `area()`.

- Interfaces are thus like *certificates*, which say “I provide these services”.
- You *can’t* make an instance of an interface:

```
Shape s = new Shape();   // ERROR!
```

2.1 Abstract Methods & Classes

- With interfaces, the entire functionality of the interface is left unspecified.
- But suppose you want to provide *some* of the functionality of a class, but leave some of it unspecified.
- Use *abstract* methods.

```
public abstract class Shape {

  public abstract float area () ;

}
```

- As with interfaces, you *can't* make an instance of an `abstract` class directly.

  You must *sub-class it* and provide an implementation for the abstract methods.

```
public class Rectangle extends Shape {
  ...
  public float area() {
    return (float)(width * height);
  }
  ...
} // end class Rectangle
```

2.2 Notes…

- If you don't provide *any* implementation, then use an interface in preference to an abstract class.
- A class can implement many interfaces, but extend only one super-class;
- Interfaces are thus how Java provides (a kind of) *multiple inheritance*.
- If even one method in a class is declared to be `abstract`, then the whole class must be declared `abstract`.
- Both abstract classes and interfaces *can* contain constants, which will be inherited by classes that `extend` or `implement` them respectively.

3 Observers

- Suppose a collection of objects (*observers*) all want to monitor a particular object (*subject*), and be *notified* when a particular event happens within this object.
- EXAMPLE. A collection of applications (word processors, spreadsheets. . .) want to know when a printer handled by a `Printer` object becomes available.
- Bad solution:

  Each observer repeatedly polls the subject to see if event has occurred.
- Better solution:

  Subject notifies observers when the event occurs.

1. Subject `s` and observer objects `o1`, ... `on` are created.
2. Each observer `oi` invokes `register` method on subject, passing itself as an argument:

   ```
   s.register(this);
   ```

3. The `register` method causes the object passed as a parameter to be added to a list of observers of the subject.

   (This list is held within the subject.)

4. When the relevant event occurs inside `s`, the object `s` gets the list of registered observers and for each object `oi`, invokes a method `notify` on `oi`, passing itself as an argument.

```
public interface Subject {

 // adds the calling object to
 // the list of observers

 public void register(Object o)


 // removes the calling object
 // from the list of observers

 public void deRegister(Object o) ;

}

public interface Observer {

 // notify observer that the event
 // has occurred

 public void notify(Object o) ;

}
```

4 Proxies

- Particularly in distributed systems, it is useful for objects to "pretend" they are talking to objects that are actually distributed across the network.
- Standard method for doing this:

  proxy objects

- In distributed object systems, a proxy object `p` to an ordinary object `o` provides all the methods that `o` does.

  When a method `m` is invoked on `p`, by object `q`, the proxy `p` communicates across the network to a program that is managing `o`.

  This program then invokes the method `m` on `o`, collects any return parameters, and sends them back to `p`.

  Then `p` returns the arguments to `q`.

  As far as `q` is concerned, `p` actually executed the method itself.

- Clients need not be aware of where objects actually are.