Object-Oriented Design **Lecture 20**
CSU 370 Fall 2007 (Pucella) **Friday, Nov 30, 2007**

The Way Forward

These notes are very sparse.

In this lecture, I want to touch on a number of topics that are intrinsically useful, but that we do not have time to cover. These topics are important because chances are you will actually encounter them in upcoming years.

(1) Design Patterns: The Reference

We've been talking about design patterns these last few weeks, and saw a few examples. There are several other design patterns, that cover most situations you can imagine.

The book that first introduced design patterns to the world, and that remains the best reference on the topic, is "Design Patterns: Elements of Reusable Object-Oriented Software" (Addison-Wesley Professional Computing Series), by by Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides, also known as "The Gang of Four".

It was the first book to "formalize" the idea of design patterns for software, and, probably even more helpful, gave a taxonomy of patterns for the working programmer.

All patterns come with sample code, but the code is in C++. However, once you know the design pattern you need, you can look up online and find a Java or C# version pretty easily.

(2) JUnit

We have advocated testing in this course, and in particular blackbox unit testing. A blackbox unit tester is what you had to implement in Homework 3, to test the `Pitch` class you implemented in Homework 2.

JUnit is a product (see `http://junit.org/`) to automate the testing of software. The basic idea is to write tests in your code as special classes, and the framework will automatically discover those tests and run them. Here is an example. (This is not the latest version of JUnit, but it illustrates the point anyways.)

```
public class HelloWorld extends TestCase {
  public void testMultiplication()
  {
    // Testing if 3*2=6:
    assertEquals ("Multiplication", 6, 3*2);
  }
}
```

This describes a class that implements tests (because it extends `TestCase`, a JUnit class), and that test performs a simple check. How does JUnit know what tests to run? It uses *reflection* (see below) to discover at runtime all the classes that extend `TestCase`, and all the methods that those classes implement, and run all those test methods.

More recent versions of JUnit do not rely on reflection, but on a new feature of Java called annotations. I will let you discover what those are for yourself.

(3) Reflection

As I said, JUnit used to rely on reflection. Reflection is a strange beast in the Java world. For most guarantees that Java provides, reflection offers a backdoor that invalidates those guarantees. Because of that, it is a dangerous toy. It is also an inefficient toy, in that using reflection can slow down programs. However, there are things that can only be done cleanly using reflection.

Very roughly speaking, reflection gives you access to (a representation of) the source code of your program from within your program.

Consider the following example. Suppose we have a class `Foo` with a method `bar` that takes no arguments. Creating an object of type `Foo` and calling its `bar` method is simple enough:

```
  Foo foo = new Foo ();
  foo.bar();
```

However, suppose that `Foo` has several different methods, all taking no arguments, and the programmer does not know a priori which method will be invoked. How can that be? Well, imagine that the program simply asks the user for which method to invoke on object `foo`. This sounds odd, but there are times where this is close to what we want. (In fact, one can come up with examples of this in the game of Homework 7.)

Assume we have a class `Input` with a static method `getInput`. We might want to try something like this:

```
  Foo foo = new Foo ();
  String in = Input.getInput();
  foo.in();
```

but that clearly doesn't work, as it tries to invoke method `in` on object `foo`, not the method named after the string contained in variable `in`. There's no easy way around that.

Enter reflection. Here's how to reproduce the above simple invocation of `bar` with reflection. The reflection classes are in package `java.lang.reflect`, so you will need to import its content. First, get a representation of the source code for class `Foo`:

```
  Class<Foo> cls = Class.forName("Foo");
```

This returns an object of type `Class` which represents the source code of class `Foo`, given as a string argument. Next, we create a new instance of the class:

```
  Object foo = cls.newInstance();
```

This is essentially equivalent to performing a `new`, except that it does it indirectly, through the representation of the class we got our hands on earlier. Now that we have an instance, let's invoke method `bar` on it. First, we need to get a representation of the method to invoke by querying the source code of the class.

```
  Method method = cls.getMethod("bar",null);
```

This gives us back a representation of the method `bar` in class `Foo`. We can now invoke that method on object `foo`:

```
  method.invoke(foo,null);
```

There. Invoking `foo.bar()` ,the long way around.

What's interesting is that both the class name and the method name to invoke are just strings. Meaning that we can actually use anything that evaluates to a string in their place. Thus, we can solve our problem above:

```
  Class cls = Class.forName("Foo");
  Object foo = cls.newInstance();
  String in = Input.getInput();
  Method method = cls.getMethod(in,null);
  method.invoke(foo,null);
```

Voilà. Of course, if the user inputs a method name that is actually not implemented by class `Foo`, the call to `cls.getMethod()` will fail with an exception.

(4) Assertions

Let's go back to testing now. As I said earlier, the focus of the course has been on blackbox unit testing, and JUnit does provide an infrastructure for helping.

However, there are times when whitebox testing, that is, testing where the tests or checks are put *inside* the class to be tested, is useful, and more natural.

Java does provide some help in doing that.

An *assertion* takes the following form:

`assert` *BooleanExpression*;

or

`assert` *BooleanExpression* : *ValueExpression*;

When Java encounters an `assert` during execution, it does one of two things, depending on whether or not assertion checking is turned on or off. (It is off by default.)

- If assertion checking is turned on, it will evaluate the *BooleanExpression* in the assertion; if it evaluates to true, then execution proceeds to whatever statement follows the assertion; if it evaluates to false, then an exception `AssertError` is thrown (with a value given by *ValueExpression*, if one is provided).
- If assertion checking is turned off, then the assertion is skipped altogether.

Thus, assertions provide a way to put in "sanity checks" within your own code, that you can enable by running the code with assertion checking turned on, with the property that if you run the code with assertion checking turned off, it behaves just like you never had put in any assertions in the first place. (There is no runtime penalty, either.)

To execute a Java program with assertion checking turned on, use:

```
java -ea Program
```

(assuming `Program.class` is the compiled class containing the `main` method).

I will refer you to the course web page for a particularly good tutorial about assertions, from the Java documentation. It describes cases in which assertions are useful.