Homework 4: Aspect-Oriented Programming and AspectJ

Due: Thursday, November 6, 2003.

This homework should all be done individually. Its purpose is to help you explore aspect-oriented programming and the design of AspectJ.

Don't hesitate to contact the staff if you are not clear about what to do.

See the syllabus for readings for this homework, which include the article by Kiczales, *et al.* we passed out, as well as the material on the `aspectj.org` website. The course web page has material on running Java, and the AspectJ compiler is installed already on the department Linux machines. It's called `ajc`; you can also get a copy for your home machine from `aspectj.org`. It's handy to use AspectJ from the Eclipse IDE; this should also be installed already on the department Linux machines.

1. To begin copy the directory

   ```
   /home/course/cs541/public/homework/aspectjhw
   ```

   to your home directory. To work with the following problems, you must have the jar files for JUnit and AspectJ's runtime in your `CLASSPATH`, as well as the directory that is just above `aspectjhw` (i.e., your home directory if you copied it there directly). For example, you can have on Linux, with the bash shell:

   ```
   CLASSPATH="$HOME:/opt/junit/junit.jar:/opt/aspectj/lib/aspectjrt.jar:."
   export CLASSPATH
   ```

   (You can put this in your .bashrc file if you wish, but you may have other Java projects that need other `CLASSPATH` settings.)

   If you use tcsh or csh as your shell (again on Linux), you should instead of the above execute something like:

   ```
   setenv CLASSPATH "$HOME:/opt/junit/junit.jar:/opt/aspectj/lib/aspectjrt.jar:."
   ```

   On a Windows machine, replace the colons (`:`) in the `CLASSPATH` above by semicolons (`;`). Also, if you're not using bash, set the `CLASSPATH` environment variable from the control panel. (If you don't have JUnit at home, you will need to download that first from `http://www.junit.org`.)

   If you use Eclipse, just include the junit jar file named in the `CLASSPATH` above in your Java Build Path's Libraries path, and work in an AspectJ project. See `http://www.eclipse.org/ajdt/` for more information about using Eclipse with AspectJ, and in particular, look at the post-installation instructions.

   To test that everything is working before you start, compile, with `ajc`, the `Fibonacci.java` and `FibonacciTest.java` files, and run `aspectjhw.FobonacciTest`. If you are using the command line on Linux, this is done as follows. First change to the directory immediately above `asjpectjhw` (e.g., `cd ..` if you are currently in that directory), then do:

   ```
   ajc aspectjhw/Fibonacci.java aspectjhw/FibonacciTest.java
   java aspectjhw.FibonacciTest
   ```

   If all goes well, then you should see output like the following.

```
.
Time: 0.015

OK (1 tests)
```

In Eclipse, you should see a green progress bar for the JUnit test if you run FibonacciTest as a JUnit test (or the above output if you run it as an application).

If that doesn't work, carefully review the instructions above, and make sure that everything is installed correctly. If you're still having trouble, see the staff for help.

2. (15 points) In this problem you will write a simple development aspect for tracing some Java code. In AspectJ, write an aspect, `aspectjhw.FibTracing`, which you should put in a file `aspectjhw/FibTracing.java`. Define a single named pointcut, `fibpc`, that matches all executions of methods named `fib*`, and two pieces of advice. The first piece of advice should be `before` advice, that prints out "`calling `", the name of the called method (use methods defined on `thisJoinPoint` to get the name — look in API AspectJ defines for this), "`(`", and the argument passed to the call, "`)`", and a newline. You can use `System.out.println` to do this. For example, your code would print "`calling fib(3)`" followed by a newline. The second piece of advice should be `after returning` advice that prints out the result returned by the call, "` == `", the name of the called method (again use `thisJoinPoint` to get the name), "`(`", and the argument passed to the call, "`)`", and a newline. For example, your code would print "`2 == fib(3)`" followed by a newline.

   Now compile your aspect together with `Fibonacci.java` and `FibonacciTest.java`, as follows (on Linux).

   ```
   ajc aspectjhw/FibTracing.java aspectjhw/Fibonacci.java aspectjhw/FibonacciTest.java
   ```

   The tests should pass and the output should also include your tracing information, which starts as follows:

   ```
   calling fib(0)
   0 == fib(0)
   calling fib(1)
   1 == fib(1)
   calling fib(2)
   calling fib(1)
   1 == fib(1)
   calling fib(0)
   0 == fib(0)
   1 == fib(2)
   calling fib(3)
   calling fib(2)
   calling fib(1)
   1 == fib(1)
   calling fib(0)
   0 == fib(0)
   1 == fib(2)
   calling fib(1)
   1 == fib(1)
   2 == fib(3)
   calling fib(8)
   ...
   ```

   Make a printout of your aspect and the testing output, and hand those in for this problem.

3. (15 points) This problem modifies the solution of the previous problem, so you need to make a printout for the previous problem first before continuing.

   In this problem, change the `FibTracing.java` aspect so that it prints a display that indents matching calls and returns by the same amount, and so that a recursive call is nested one space more than the call that originated it. Thus when you compile and run your new tracing output with the Fibonacci test, your output should start like:

```
calling fib(0)
0 == fib(0)
calling fib(1)
1 == fib(1)
calling fib(2)
 calling fib(1)
 1 == fib(1)
 calling fib(0)
 0 == fib(0)
1 == fib(2)
calling fib(3)
 calling fib(2)
  calling fib(1)
  1 == fib(1)
  calling fib(0)
  0 == fib(0)
 1 == fib(2)
 calling fib(1)
 1 == fib(1)
2 == fib(3)
calling fib(8)
...
```

   Be sure to modularize your code so that you don't write the same thing several times.

   Make a printout of your aspect and the testing output, and hand those in for this problem.

4. (15 points) Without changing the `aspectjhw.FibTracing` aspect you wrote in the previous problems or any other existing code in `aspectjhw.Fibonacci` or `aspectjhw.FibonacciTest`, write an aspect, `aspectjhw.BlankLines` that makes the output have a blank line after each successive outermost call to `Fibonacci.fib` returns. (Hint: use `cflowbelow`.) That is the output should start as follows.

```
calling fib(0)
0 == fib(0)

calling fib(1)
1 == fib(1)

calling fib(2)
 calling fib(1)
 1 == fib(1)
 calling fib(0)
 0 == fib(0)
1 == fib(2)

calling fib(3)
 calling fib(2)
```

```
  calling fib(1)
  1 == fib(1)
  calling fib(0)
  0 == fib(0)
 1 == fib(2)
 calling fib(1)
 1 == fib(1)
2 == fib(3)

calling fib(8)
...
```

Make a printout of your aspect and the testing output, and hand those in for this problem.

5. (15 points) Without changing the aspects you wrote in the previous problems or any other existing code, write an aspect, `aspectjhw.FibTiming` that prints out the number of milliseconds taken by each top-level (outermost) call to `Fibonacci.fib` before the blank line printed by the `aspectjhw.BlankLines`. (Hint: use `around` advice and Java's `System.currentTimeMillis()` method.) Your output should thus start as follows.

```
calling fib(0)
0 == fib(0)
Time: 10 ms

calling fib(1)
1 == fib(1)
Time: 0 ms

calling fib(2)
 calling fib(1)
 1 == fib(1)
 calling fib(0)
 0 == fib(0)
1 == fib(2)
Time: 10 ms

calling fib(3)
 calling fib(2)
  calling fib(1)
  1 == fib(1)
  calling fib(0)
  0 == fib(0)
 1 == fib(2)
 calling fib(1)
 1 == fib(1)
2 == fib(3)
Time: 0 ms

calling fib(8)
...
```

Of course the times you actually see will vary depending on the version of Java you use, the operating system, the machine, etc.

Make a printout of your aspect and the testing output, and hand those in for this problem.

6. (15 points) Without changing the aspects you wrote in the previous problems or any other existing code, write an aspect, `aspectjhw.FibMemo` that keeps a table relating arguments and results of `Fibonacci.fib`, and uses these to avoid recomputation previously-computed values for `Fibonacci.fib`. (Hint: use a `java.util.Hashtable`.)

   When you test, the output should be as follows.

```
calling fib(0)
0 == fib(0)
Time: 20 ms

calling fib(1)
1 == fib(1)
Time: 0 ms

calling fib(2)
1 == fib(2)
Time: 0 ms

calling fib(3)
2 == fib(3)
Time: 0 ms

calling fib(8)
21 == fib(8)
Time: 0 ms
```

   Make a printout of your aspect and the testing output, and hand those in for this problem.

7. (15 points) You may notice that the test in the above program run noticeably faster than before. We should be able to use the memoized version to compute the Fibonacci function of moderately-sized arguments.

   Without changing the aspects you wrote in the previous problems or any other existing code, write an aspect `aspectjhw.FibLargerTest` that adds another test method, `testLargerArgs` to the class, `aspectjhw.FibonacciTest`, and with body

```
assertTrue(Fibonacci.fib(30) > Fibonacci.fib(20));
assertTrue(Fibonacci.fib(40) > Fibonacci.fib(30));
assertTrue(Fibonacci.fib(50) > Fibonacci.fib(40));
assertTrue(Fibonacci.fib(70) > Fibonacci.fib(50));
```

   Make a printout of your aspect and the testing output, and hand those in for this problem.