Homework 4: Concepts of Aspect-Oriented Programming and Aspect Smalltalk

Due: Tuesday, November 12, 2002.

All but the last problem on this homework should be done individually. The purpose of these problems is help you better understand the concepts behind aspect-oriented programming in AspectJ.

The last problem on this homework can either be done individually or in teams. Its purpose is to design an aspect-oriented version of Smalltalk.

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 two articles 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`.

1. (15 points) Describe, in general terms, the kinds of changes that aspects written in AspectJ can make to a Java program. That is, what kinds of things can you do in AspectJ that you can't easily do in Java itself. Give the AspectJ syntax for each such change by showing a brief example of each.

2. (20 points) An aspect written in AspectJ can be seen as a kind of edit of a program. Indeed the compiler `ajc` has a switch, `-preprocess`, that can be used to look at the source code of what it would normally generate in a `.class` file. So one could imagine that instead of using AspectJ directly one could use editing scripts (e.g., in Perl) to edit a Java program instead of AspectJ. Discuss the advantages of using a programming language, such as AspectJ instead of such editing scripts. To be comprehensive, in your answer be sure to treat each of the kind of changes you described in the previous problem.

3. (20 points) For each of the following, describe the advantages and disadvantages of simplifying AspectJ by removing the named feature. (Another way to think of this is what is the purpose and importance of having the feature in the language.)

   (a) Abstract pointcuts and abstract aspects.

   (b) Around advice.

   (c) Both `call` and `execution` joinpoints.

   (d) The `cflow` and `cflowbelow` joinpoints.

4. (30 points) This problem concerns type checking and AspectJ's `around` advice.

   (a) Describe the problem of type checking `around` advice. (I.e., how could you get a type error in `around` advice that passes the AspectJ type checker?)

   (b) How could you change AspectJ, e.g., by restricting some of its features, so that it would be possible to have sound static type checking of AspectJ's around advice? Would this be a good idea?

   The following problem can be done in a team if you wish.

5. Design, on paper, a set of changes to Smalltalk that would support aspect-oriented programming. We'll call this language Aspect Smalltalk. Note that there are a couple of designs for an aspect-oriented version of Smalltalk on the web. For purposes of this exercise, I think it is best to ignore these, but if you look at them, be sure to credit any ideas, wording, etc. that you use appropriately.

(a) (40 points) Describe the goals and objectives for your design by explaining what kinds of changes you want to support. That is, describe the kinds of changes to Smalltalk you would like to automate and why you want to automate them.

(b) (30 points) Give grammars that define the microsyntax (lexical grammar) and syntax of Aspect Smalltalk. You only need to give productions that are different from those in Smalltalk. Use the nonterminal names in the back of the Goldberg and Robson's book (*Smalltalk-80*) as your source for the Smalltalk grammar. But don't draw railroad charts like that yourself, instead use the usual conventions of context-free grammars (as shown in class, for example).

(c) (100 points) For each grammar production that you add to Smalltalk, give a description of the semantics (meaning) of that production along with a few simple examples of its use. These semantics can be written in clear English, although you should also use mathematical notation whenever helpful. Make whatever technical definitions you find helpful, but label these clearly as definitions.

(d) (25 points, extra credit) Explain how would you go about evaluating the quality of your language design, assuming you had sufficient time and resources.

(e) (75 points, extra credit) If you have an interest, implement part of your langauge design in Squeak. Give examples of how it works and hand in a printout of your source code.