COMP6209 Assignment Instructions

Module	COMP6209: Automated Code Generation			**Lecturer**	*Julian Rathke*
Assignment	Coursework 1: AspectJ			**Weight**	*20%*
Deadline	15*/03/2017*	**Feedback**	16*/04/2017*	**Effort**	30 *hours*

Instructions

The goal of this coursework is to use AspectJ in order to profile Java programs. It consists of two parts. The first part is concerned with *qualitative* properties and aims at constructing the dynamic call graph of a program. The second part considers *quantitative* properties, collecting the execution time of certain methods.

Part 1: Dynamic call-graph construction

We define the call graph of a Java program as a directed graph whose nodes are method signatures. There is a node for each method that is called, and an edge from node *mi* to node *mj* if method *mj* is called within the definition of method *mi*. We say that a call graph is *dynamic* if it is built at runtime, when nodes and edges are added during program execution.

Write an aspect that constructs the dynamic call graph of a Java program restricted to all public methods that take one argument `int`, return an `int`, and are defined in any class within the package called `q1` or any of the packages within the `q1` hierarchy.

For instance, consider the class:

```
package q1;
public class A {
  public int foo(int a) {
    bar();
    return 0;
  }
  private void bar() {
    baz(4);
  }
  public int baz(int a) {
    return a + a;
  }
}
```

The dynamic call graph upon invoking `A.foo` would have no edges because `A.foo` does not invoke any other method of interest within its definition. Instead, with the class

```
package q1;
public class B {
  public int foo(int a) {
    bar(1);
    return 0;
  }
  public int bar(int b) {
    return baz(b);
  }
  public int baz(int a) {
    return a + a;
  }
}
```

the dynamic call graph upon invoking `B.foo` would have two edges

`B.foo(int) -> B.bar(int)` and `B.bar(int) -> B.baz(int).`

For any two nodes, there is at most one edge between them. There may be nodes in the graph that have no edges connecting them at all.

After the execution of the program, the aspect should produce the following output:

1. A file in comma-separated value format named `q1-nodes.csv` with the list of nodes of the dynamic call graph (one node per line).
2. A file in comma-separated value format named `q1-edges.csv` where each line consists of a pair *source method, target method* which represents an edge in the call graph.

You may prefer to use the symbols —> as a separator rather than commas. It is fine to do this.

Organise your code in a package named `q1.<yourUsername>`.

Part 2: Refined call-graph construction

Write an aspect that refines your solution to Part 1 (where package name q1 is replaced throughout by q2) by **not providing** an arc from *mi* to *mj* if *mj* if *mj* throws an exception instance of `java.lang.Exception` when called within the definition of *mi.*

Edges should still be included for any method call that terminates normally, even if a later call to the same method throws an exception.

Organise your code in a package named `q2.<yourUsername>`. Much of this should be similar to Part 1 with a small amount of extra code.

Part 3: Runtime profiling

Write an aspect to obtain an input/output summary of the methods identified in Part 1 (where package name q1 is replaced throughout by q3).

1. For each method, the aspect should compute a histogram of the frequency of the values of the `int` parameter passed as an input as well as of the returned `int` value.
   At the end of the program execution, this information should be saved to files named

`<method signature>-hist.csv` (i.e., one file for each method) using a comma separated values format. Each histogram should have three columns: the value of interest, how many times it has been used as an input, how many times it has been returned as a result.

2. The aspect should compute the *frequency of failures*, defined as the percentage of times that the method does not return an `int` due to throwing an exception.
   This information should be outputted to a file named `failures.csv` in comma-separated values format, where each line provides the failure frequency of a method.
3. Finally, for each method the aspect should compute the average execution time and its standard deviation across all its invocations.
   This information should be outputted to a file named `runtimes.csv` in comma-separated values format (one line for each method).

Organise your code in a package named `q3.<yourUsername>`. It is okay to use three separate aspects for each of these tasks.

Submission

Please submit well-documented AspectJ source code. **Do not** submit any Java source files. We will test your aspects on our own class files.

Please submit using the Handin system (http://handin.ecs.soton.ac.uk) by 4pm on the due date.

Relevant Learning Outcomes

1. *Aspect-oriented programming techniques*

Marking Scheme

Criterion	**Description**	**Outcomes**	**Total**
Part 1	*AspectJ code for call-graph construction*	*1*	*8 marks*
Part 2	*AspectJ code for refined call-graph construction*	*1*	*4 marks*
Part 3	*AspectJ code for execution runtime profiling*	*1*	*8 marks*

Late submissions will be penalised at 10% per working day. No work can be accepted after feedback has been given. Please note the University regulations regarding academic integrity.