Fall, 2004 Name: ______________________________

Com S 541 — Programming Languages 1

Test on Aspect-Oriented Languages and AspectJ

This test has 10 questions and pages numbered 1 through 13.

Special Directions for this Test

This test is open book and notes.
If you need more space, use the back of a page. Note when you do that on the front.
This exam is timed. We will not grade your exam if you try to take more than the time allowed. Therefore, before you begin, please take a moment to look over the entire exam so that you can budget your time.
Clarity is important; if your diagrams or programs are sloppy and hard to read, you will lose points.
Correct syntax also makes a difference for programming questions.
When you write AspectJ code on this test, you may use anything in the Java standard libraries, or in the packages `org.aspectj.lang` or `org.aspectj.lang.reflect` without writing it in your test. You are encouraged to define classes, interfaces, fields, and methods not specifically asked for if they are useful to your programming; if they are not in the standard libraries or the packages named above, please write them into your test.

Base Program

In the first several problems we will use the following code as a base program. The base program contains two packages. The first package is `geom`, which consists of three classes: `Point`, `ColoredPoint`, and `Line`.

```
package geom;

/** Two-dimensional, cartesian points.
 */
public class Point {
    /** The x coordinate. */
    private int x = 0;
    /** The y coordinate. */
    private  int y = 0;

    /** Initialize this point to the origin. */
    public Point() { }

    /** Initialize this point to the given coordinates. */
    public Point(int xc, int yc) {
      x = xc;
      y = yc;
    }

    /** Return this point's x coordinate. */
    public /*@ pure @*/ int getX() {
      return x;
    }

    /** Return this point's y coordinate. */
    public /*@ pure @*/ int getY() {
      return y;
    }

    /** Move this point's x coordinate by the given amount. */
    public void moveX(int dx) {
      x += dx;
    }

    /** Move this point's y coordinate by the given amount. */
    public void moveY(int dy) {
      y += dy;
    }

    public boolean equals(Object o) {
        Point p = (Point) o;
        return p.x == x && p.y == y;
    }

    public int hashCode() {
        return x + y;
    }
}
```

```
package geom;

import java.awt.Color;

/** Colored Points. */
public class ColoredPoint extends Point {

    /** This point's color attribute. */
    private Color c;

    /** Initialize this ColoredPoint. */
    public ColoredPoint() {
        c = Color.BLACK;
    }

    /** Initialize this colored point. */
    public ColoredPoint(int xc, int yc, Color c) {
        super(xc, yc);
        this.c = c;
    }

    /**Returns the color of this point. */
    public Color getColor() {
        return c;
    }

    /** Set the color of this point to the argument. */
    public void setC(Color c) {
        this.c = c;
    }
}
```

```
package geom;

/** Lines in 2-dimensional space.
 */
public class Line {

    /** The starting point. */
    private Point start;
    /** The ending point. */
    private Point end;

    /** Initialize this line to go from the given
     * start to the given end point. */
    public Line(Point start, Point end) {
        this.start = start;
        this.end = end;
    }

    /** Return the end point of this line. */
    public Point getEnd() {
        return end;
    }
    /** Return the starting point of this line. */
    public Point getStart() {
        return start;
    }
    public boolean equals(Object o) {
        Line ln = (Line) o;
        return ln.start.equals(start)
            && ln.end.equals(end);
    }

    public int hashCode() {
        return start.hashCode() + end.hashCode();
    }
}
```

The second package is `another`, which has two classes that are designed for testing purposes and don't have any practical use.

```
package another;

/** For testing the equals method advice. */
public class MyType {

    /** Always return true. */
    public boolean equals(Object o) {
        return true;
    }

    public int hashcode() { return 0; }
}
```

```
package another;

/** For testing the equals method. */
public class SubType extends MyType {
}
```

1. (10 points) In AspectJ, without changing the base program's code (see the previous pages), write an aspect `answers.LineConstructorPre` that checks that both arguments to the constructor of `geom.Line` are not null, and throws a `java.lang.AssertionError` exception otherwise.

   For example, with the required aspect, each of the following new object construction expressions would throw this exception when executed.

   ```
   new Line(null, new Point());
   new Line(new Point(), null);
   new Line(null, null);
   ```

   However, the expression `new Line(new Point(), new Point(5,41))` would terminate normally.

2. (15 points) Without changing the base program code or any aspects written previously, write an aspect, `answers.EqualsNullEnforcer`. This aspect is to automate the otherwise tedious job of writing code to check that, whenever the method `equals(Object)` is executed with an argument of `null`, it returns false.

   For example, when compiled together with your aspect, the following expressions should all return `false`:

```
new Point(3,4).equals(null));
new Point().equals(null));
new ColoredPoint().equals(null));
new Line(new Point(), new Point(5, 4)).equals(null));
new MyType().equals(null));
```

   However, the behavior of the `equals` method on other arguments should not be changed by your aspect. Your aspect should work for all types, not just the base program types given above.

3. (15 points) Without changing the base program or other aspects, write an aspect, `answers.EqualsEqOptimizer` that automates the otherwise tedious job of writing code to check that, whenever the method `equals(Object)` is called with an argument that is the same as (i.e., is `==` to) the receiver, it should return true.

   For example, when compiled together with your aspect, the following JUnit test code should pass but not actually call the `equals` methods:

   ```
   Point p = new Point();
   assertTrue(p.equals(p));

   Line ln = new Line(new Point(), new Point(5, 4));
   assertTrue(ln.equals(ln));
   ```

   However, the behavior of the `equals` method on other arguments should not be changed by your aspect. Your aspect should also work for all types, not just `Point` and `Line`.

4. (5 points) Without changing the base program, write an aspect `answers.Glue` to make each type in the package `geom` be a subtype of the following interface, `contracts.EqualsSameTypeOnly`.

```
package contracts;

/** This interface marks classes for which
 * equals should return true only when the argument's
 * dynamic type is the same as the receiver's class.
 */
public interface EqualsSameTypeOnly {}
```

(This interface will also be used in the following problem.)

5. (15 points) In this problem, you will write an aspect, `answers.EqualsSameTypeOptimizer`. The advice you are to write should automate the otherwise tedious job of writing code to check that, whenever the method `equals(Object)` of some type T is called with an argument of of dynamic type S, where $S \neq T$, it returns false. (Hint: in Java, the expression `o.getClass()` returns the dynamic type of `o` as an object of type `java.lang.Class`. You can use `==` to compare these singleton class objects.) However, the advice you write should only apply to types that are subtypes of the type, `contracts.EqualsSameTypeOnly`.

   For example, suppose that `ColoredPoint` is a subclass of `Point`. Then when it and the rest of the base program are compiled together with your aspect, the following expressions should all return `false`:

   ```
   new geom.Point().equals(new geom.ColoredPoint());
   new geom.Point().equals(new geom.Line(new geom.Point(), new geom.Point(5, 4)));
   new geom.Line(new geom.Point(), new geom.Point(5, 4)).equals(new geom.Point());
   ```

   However, the behavior of the `equals` method on other arguments should not be changed by your aspect. Furthermore, since the types in the package `another` do not implement `contracts.EqualsSameTypeOnly`, the expression

   ```
   new another.MyType().equals(new another.SubType())
   ```

   would still return true.

6. (5 points) In Java, one cannot call the `getClass()` method on `null`. So there is a potential problem when combining the advice in `answers.EqualsSameTypeOptimizer` (in the previous problem) and `answers.EqualsNullEnforcer` (in problem 2), since the check for null must be done before the check for the same type. Without changing the code of the base program or any aspects, write an aspect `answers.FixOrder` that ensures that the advice in `answers.EqualsNullEnforcer` executes before the advice in `answers.EqualsSameTypeOptimizer`.

7. Suppose we added `around` advice to Haskell.

   (a) (5 points) Describe one use for around advice in AspectJ that that would still work well in Haskell. Explain briefly or give a short example.

   (b) (5 points) Describe one use for around advice in AspectJ that would not work well in Haskell. Explain briefly or give a short example.

8. The the new version of Java, 1.5, like C#, has an attributes feature. *Attributes* allow one to statically add user-defined modifiers to declarations of fields, methods, and classes. For example, one might designate a method as an `@atomic` method for purposes of concurrency control, state that a method was `@pure` in the sense that it should have no side-effects, or state that a method was a `@test` method, to be used by a testing tool. In Java, these attributes can be processed by various tools, and can be queried at run-time (using Java's reflection mechanism).

   (a) (5 points) Would it be useful to extend AspectJ's `declare` syntax for static introductions to allow one to add attributes to existing code? That is, for the new version of Java, would it be useful to add to AspectJ the following syntax?

   AspectMemberDecl ::= `declare attributes :` *Pattern* `:` *AttributeList* `;`
   Pattern ::= *FieldPattern* | *MethodPattern* | *ConstructorPattern* | *TypePattern*

   Explain briefly.

   (b) (5 points) Would it be useful to extend AspectJ's *ModifersPattern* syntax to match attributes? (The current *ModifersPattern* syntax is used for matching modifiers in other patterns.) Explain briefly.

9. (5 points) What expressive power would AspectJ lose if the "`this`" pointcut description were eliminated from the language?

10. (10 points) This is a question about AspectJ's type system. Recall that Java ensures that each type of checked exception that may be thrown by an expression (or statement) in a method is either handled with a surrounding `try-catch` statement, or is a subtype of an exception type declared in the method's header.

    Consider `before` advice of the form:

```
before() throws EOFException
  :  calls(void foo() throws FileNotFoundException) {
    throw new EOFException();
}
```

In Java `EOFException` and `FileNotFoundException` are both checked exceptions, but neither is a subtype of the other. Let us assume that there is only one method `foo()` in the program, and that it's header is

```
  public void foo() throws FileNotFoundException
```

(and thus it may only throw exceptions of type `FileNotFoundException` (or a subtype) when called.

Ignoring what AspectJ actually does with this advice, should this be considered type correct or a type error? Explain.