

# Writing Self-testing Java Classes with SelfTest

Yoonsik Cheon

TR #14-31  
April 2014

**Keywords:** annotation; annotation processor; test case; unit test; Java; JUnit; SelfTest.

**1998 CR Categories:** D.2.3 [*Software Engineering*] Coding Tools and Techniques — Object-oriented programming; D.2.5 [*Software Engineering*] Testing and Debugging — Testing tools (e.g., data generators, coverage testing); D.3.4 [*Programming Languages*] Processors — Compilers, preprocessors.

Department of Computer Science  
The University of Texas at El Paso  
500 West University Avenue  
El Paso, Texas 79968-0518, U.S.A.

# Writing Self-testing Java Classes with SelfTest

Yoonsik Cheon

Department of Computer Science  
The University of Texas at El Paso  
El Paso, Texas, U.S.A.  
ycheon@utep.edu

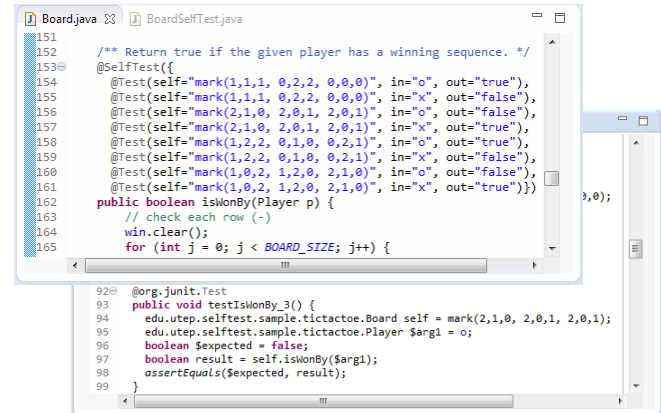
**Abstract**—This document provides a tutorial introduction to Java annotations called *SelfTest*. The *SelfTest* annotations allow one to annotate Java classes with test data, and the *SelfTest* annotation processor generates executable JUnit test classes from annotated Java classes by translating test cases to executable JUnit tests. The *SelfTest* annotations not only automate unit testing of Java classes significantly but also provides a step toward writing self-testing Java classes by embedding test data in source code for both compile and runtime processing.

## I. INTRODUCTION

Program testing is a practical means for improving and assuring the correctness of software. In Extreme Programming (XP) [1], for example, unit testing is viewed as integral part of programming. Tests are created before, during, and after code is written—often emphasized as “code a little, test a little, code a little, and test a little ...” [2]. The idea is to use regression testing as a practical means of supporting code refactoring. However, writing unit tests is a laborious, tedious and time-consuming task. One can use a framework that automates some of the details of running tests. One such a framework is JUnit, a simple yet practical testing framework for Java classes [2] [5]. It encourages a close integration of testing with development by allowing a test suite be built incrementally. However, even with frameworks like JUnit, writing and maintaining unit tests requires a great deal of time and effort. Separate test code must be written and maintained in synchrony with the code under development. This test code must be reviewed when the code changes and, if necessary, revised to reflect changes in the code. The difficulty and expense of writing test code are exacerbated during development, when the code being tested changes frequently. As a result, there is pressure to not write test code and to not test as frequently as might be optimal.

Annotations are compiler directives in that they provide data about a program that is not part of the program itself. In Java, annotations are markers which associate information with program constructs, but have no effect at run time [3, Chapter 9]; they have no direct effect on the operation of the code they annotate, but may affect the behavior of the compiler. When source code is compiled, annotations can be processed by compiler plug-ins called annotation processors. Annotation processors can produce informational messages or create additional Java source code files or resources, but they cannot modify the annotated code itself.

*SelfTest* is a small set of Java annotation types along with an annotation processor. Its annotations allow one to annotate Java classes with test data, and the annotation processor gener-



The screenshot shows two Java files in the Eclipse IDE. The top file, `BoardSelfTest.java`, contains `@SelfTest` annotations for various tic-tac-toe board states. Each annotation includes `in` and `out` parameters representing the board state and the expected result. The bottom file, `Board.java`, contains the `isWonBy` method that checks for a winning sequence. The generated JUnit test class, `testIsWonBy_3()`, is shown below, which uses `@org.junit.Test` to run the `isWonBy` method with the test data from the annotations and asserts the result.

Fig. 1. SelfTest annotations and generated JUnit tests on Eclipse

ates executable JUnit test classes from annotated Java classes by translating test cases to executable JUnit test methods (see Figure 1). *SelfTest* was created to help programmers focus more on interesting or challenging aspects of testing by freeing them from laborious, tedious, or time-consuming testing chores. For example, when one uses unit tests as a safety net for code refactoring, one has to test every method of a class. However, it is often the case that except for a few primary or core methods most methods are simple and straightforward to test, but one still needs to test all these methods by writing executable test code. Even for testing the primary methods, a more creative and interesting aspect is designing tests and test data, not writing low-level, executable test code and maintaining it. In short, *SelfTest* allows one to embed test data in source code for both compile and runtime processing. It not only automates unit testing of Java classes significantly by generating executable tests from annotations but also provides a step toward writing self-testing Java classes.

The remainder of this document is structured as follows. In Section II below we will describe a tic-tac-toe game that will be used as a running example throughout this document. In Section III we will write a series of *SelfTest* annotations to test tic-tac-toe classes, starting with simple annotations and moving to more advanced ones. In Section IV we will list and describe all *SelfTest* annotation types. In the appendix at the end of this document we will provide information about downloading *SelfTest* and configuring Java compilers [6] and Eclipse [4] to recognize *SelfTest* annotations.

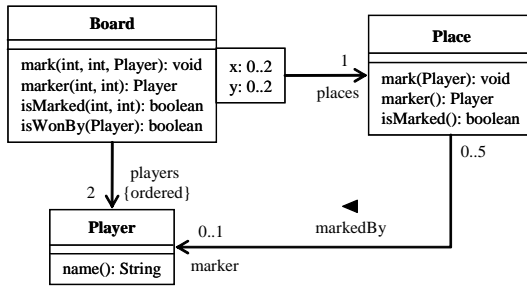


Fig. 2. A tic-tac-toe game

## II. A RUNNING EXAMPLE

We will use an implementation of the tic-tac-toe game as a running example. A tic-tac-toe game consists of nine places in a  $3 \times 3$  grid, and two players take turns to mark the places and win a game by marking three places in a horizontal, vertical, or diagonal row. We will be concerned with only the model classes of a tic-tac-toe game, shown in Figure 2. The class diagram depicts three model classes along with their primary operations and relationships. The association between Board and Place is a qualified association, an association that allows one to restrict the objects referred in an association using a key called a qualifier [7]. An optional qualifier at an association end, e.g., *x* and *y*, enables the indexing of many associations between classes. The qualifiers *x* and *y* denotes the column and row indices of a place in a board; e.g., `places[2,0]` denotes the place at the top right corner of a board.

In the next section we will annotate tic-tac-toe classes with `SelfTest` annotations to automate their unit testing by embedding test cases in source code. The embedded test cases are translated to executable JUnit tests by the `SelfTest` annotation processor. Since version 4, JUnit also uses annotations to identify methods called test methods that specify tests [5]. Figure 3 shows a JUnit test class that has two test methods, `testMark` and `testIsWonBy`. A test method uses JUnit framework methods such as `assertEquals` and `assertTrue` to check the expected result of code execution versus the actual result. It is common to share test data among test methods by sharing variables, e.g., fields like `board`, `p1`, and `p2`. These variables together define a test fixture, a fixed state of the program under test, and are often initialized by the so-called Before method (e.g., `setUp`) that is executed before each test to prepare a test environment.

## III. TESTING WITH SELFTEST

In this section we will write a series of `SelfTest` annotations to test the tic-tac-toe classes presented in the previous section. We will start with simple and most commonly-used annotations and move to more advanced annotations. The appendix at the end of this document contains information about downloading `SelfTest` and configuring Java compilers [6] and Eclipse [4] to recognize `SelfTest` annotations.

```
import org.junit.*;
import static org.junit.Assert.*;

public class BoardTest {
    private Board board;
    private Player p1 = new Player("O");
    private Player p2 = new Player("X");

    @Before public void setUp() {
        board = new Board();
    }

    @Test public void testMark() {
        board.mark(0, 1, p1);
        assertEquals(p1, board.marker(0, 1));
    }

    @Test public void testIsWonBy() {
        markBoard(1,0,2,
                2,1,0,
                0,0,1);
        assertTrue(board.isWonBy(p1));
        assertFalse(board.isWonBy(p2));
    }

    private void markBoard(int... marks) {
        // mark board to the given configuration
    }
}
```

Fig. 3. Sample JUnit tests

### A. Writing the First Self-testing Class

The first class to write and test is the `Player` class because it doesn't depend on any other classes. It will have just one constructor and one accessor method. The code listing below shows its skeleton code along with tests specified in `SelfTest` annotations.

```
1 import edu.utep.selftest.*;
2 public class Player {
3     @Test(in="\O", post="result.name().equals(\"O\")")
4     public Player(String name) { ... }
5
6     @Test(self="new Player(\"X\")", out="\X")
7     public String name() { ... }
8 }
```

The first line imports all `SelfTest` annotation types including the `Test` annotation used in this example. The `Test` annotation defines a single test case and is the most commonly-used annotation. A test case for a method or constructor is essentially a pair of sample inputs and the expected output for the inputs.

The `Test` annotation in line 3 defines a test case for the constructor. Its `in` element specifies argument values, and the `post` element specifies the condition, called a postcondition, that must be true upon termination of the method or constructor under test. A postcondition is a test oracle in that it determines the test outcome, a test success or failure. The pseudo variable `result` in the postcondition denotes the object that is initialized by the constructor. This test case tests that if the constructor is called with a value "O" then the name of the newly initialized player object should be "O".

The annotation in line 6 shows two new elements of the `Test` annotation. The `self` element specifies the implicit argument

for a non-static method—the receiver or the object under test. The *out* element specifies the expected return value for a non-void method; it is another way of writing a test oracle. This test case tests that if the *name* method is invoked on a player object, *new Player("X")* it should return a "X".

It would be instructive to see how these annotations are translated to JUnit tests. The code below shows the JUnit test class generated from the above annotations. As expected, each Test annotation is translated to a JUnit test method and a test oracle to an *assert* method call.

```
import static org.junit.Assert.*;
public class PlayerSelfTest {
    @org.junit.Test
    public void testPlayer() {
        String $arg1 = "O";
        Player result = new Player($arg1);
        assertTrue("post: result.name().equals(\"O\")",
            result.name().equals("O"));
    }

    @org.junit.Test
    public void testName() {
        Player self = new Player("X");
        String $expected = "X";
        result = self.name();
        assertEquals($expected, result);
    }
}
```

To summarize, the Test annotation defines a single test case by specifying sample inputs—including the receiver for a non-static method and the argument values—and the expected output. The expected output defines a test oracle and can be stated by writing postconditions or by specifying a return value. Each test case should define a test oracle, and the pseudo variables like *result* and *self* can be referred to when writing a test oracle.

### B. Defining Multiple Test Cases

It is common to define multiple test cases for a single method or constructor. For this, you can use the SelfTest annotation whose value is a list of Test values<sup>1</sup>. The following code snippet shows two test cases for the *isMarked* method of the Place class that checks if a place is marked.

```
@SelfTest({
    @Test(self="new Place(0, 0)", out="false"),
    @Test(self="new Place(0, 0); mark(new Player(\"O\"))", out="true")})
public boolean isMarked() { ... }
```

The first test case tests an unmarked place and the second a marked one. The second test case also shows how one can construct an object of a more complex state in a single expression. An expression of a reference type can be written in the form:  $e_0; m_1(e_1); m_2(e_2); \dots; m_n(e_n)$ . It denotes the object  $e_0$  after invoking on it all the methods  $m_i$ 's in the specified order; each  $m_i$  may mutate the object or change its state. The expression  $e_0$  defines an initial state of the object, and each

<sup>1</sup>A new annotation type SelfTest was introduced because Java doesn't allow more than one annotation of the same type; this was fixed in Java 8 released in 2014.

$m_i$  may change the object's state to bring it to a new state. For example, the state of the pseudo variable *self* defined in the second test case is the same as the one defined by the following Java statements.

```
Place self = new Place(0, 0);
self.mark(new Player("O"));
```

### C. Testing Exceptional Behavior

You can also test an exceptional behavior of a method or constructor by using the Test annotation. If the *mark* method of the Place class should throw an IllegalStateException when the given place is already marked, it can be tested using the following test case.

```
@Test(self="new Place(0, 0); mark(new Player(\"O\"))",
    in="new Player(\"X\")",
    err=IllegalStateException.class)
public void mark(Player p) { ... }
```

The *err* element specifies the exception that has to be thrown by the method or constructor under test when the specified test inputs are supplied. It is yet another way of writing a test oracle, as it determines the test outcome.

### D. Sharing Test Data

You often want to share test data among test cases. The SelfTest annotation provides ways to define variables that we call test variables that can be referred to when defining test cases, thus achieving sharing of test data. In fact, there are three different levels of sharing possible:

- sharing among test cases for all methods or constructors of a class
- sharing among test cases for a single method or constructor
- sharing in a single test case

The basic mechanism for sharing test data is to define a new test variable using the Var annotation, as shown below.

```
@Var(type=Place.class, name="p", value="new Place(1,2)")
public class Place {
    @Test(self="p", out="1")
    public int getX() { ... }

    @Test(self="p", out="2")
    public int getY() { ... }
}
```

The above Var annotation introduces a new test variable named *p* of type Place whose initial value is *new Place(1,2)*. A Var annotation like the above annotating a class introduces a test variable that can be referred to by all the test cases of the annotated class. Such a test variable is translated to a class field whose value is initialized by the so-called Before method (a.k.a. the setUp method) of a JUnit test class; that is, it defines a test fixture variable [2]. You can define more than one test fixture variable by using the SelfVar annotation whose value is a list of Var values; the order matters because a later one may be defined in terms of earlier ones.

If a `Var` annotation annotates a method or constructor, it introduces a test variable that can be referred to by all the test cases of the annotated method or constructor. Such a test variable is translated to a local variable of a JUnit test method, and its value is initialized before the method or constructor under test is called; it is a pre-state variable. For example, the following code snippet introduces two variables ( $o$  and  $x$ ) for use by test cases of the `isMarkedBy` method of the `Place` class that tests if a place is marked by a given player.

```
@SelfVar({
  @Var(type=Player.class, name="o", value="new Player(\"O\")"),
  @Var(type=Player.class, name="x", value="new Player(\"X\")")}
@Test({
  @Test(self="p", in="o", out="false"),
  @Test(self="p; mark(o)", in="o", out="true"),
  @Test(self="p; mark(o)", in="x", out="false")}
public boolean isMarkedBy(Player m) { ... }
```

It is possible for a local variable introduced by a `Var` annotation to shadow another test variable. If you change the name of variable  $o$  to  $p$  in this example, it will shadow the  $p$  test variable introduced by the `Var` annotation of the `Place` class (refer to the previous example). As in Java, you can refer to a shadowed variable by fully qualifying it, e.g., `this.p`.

You can also define variables for use in a single test case by specifying the `var` element of the `Test` annotation. The `var` element defines a list of `Var` values, as shown below. The following example also shows that you can define more than one postcondition; multiple postconditions are conjoined.

```
@Test(
  var={ @Var(type=int.class, name="x", value="1"),
        @Var(type=int.class, name="y", value="2")},
  in={"x", "y"},
  post={"result.getX() == x && result.getY() == y",
        "!result.isMarked()"}
public Place(int x, int y) { ... }
```

### E. Importing Classes for Writing Annotations

There are often cases when you need to use other classes only in annotations—i.e., not in the source code of the annotated classes themselves. You can import classes for use only in annotations by using the `SelfImport` annotation. The annotation takes a list of strings, each conforming to the import declaration syntax of Java, e.g., `java.util.Arrays` and `java.util.*`.

```
@SelfImport("java.util.Arrays")
public class MyListUtil {
  @Test(in="Arrays.asList(1,2,3)", out="6")
  public static int sum(List<Integer> l) { ... }
}
```

### F. Writing Helper Code

Perhaps, the most interesting method to test in our example is the `isWonBy` method of the `Board` class that checks if a given player has a winning row in a board. To test the method thoroughly, you will need to create `Board` objects of various configurations, e.g., with a winning horizontal, vertical, or diagonal row. One way to create objects of different states is

to use the extended expression syntax introduced earlier, e.g., `new Board(); mark(0,0,x); mark(1,0,x); mark(2,0,x)`. However, manually creating test data in such a way is time-consuming and even worse, test data are hard to read, understand and maintain.

If you are an experienced tester, however, you will instead write a helper method or class to ease the construction of test data. For example, you can easily write a helper method, say `mark`, that takes an encoding of a board configuration and constructs a `Board` object of the given configuration. However, one big question is how to include such helper code in your tests. Obviously, you don't want to mix it with regular code of the class under test because it is introduced solely for the testing purpose. One possibility would be to define a separate utility class hosting all such helper code and import it using the `SelfImport` annotation. This, perhaps, will be a recommended way if you have complicate or large amount of helper code.

However, if you have a small chunk of helper code, another possibility would be to include it directly in your annotations using the `SelfCode` annotation. The `SelfCode` annotation takes a chunk of Java code given as a list of strings and dumps it to the generated JUnit test class. The code is dumped at the member level, so it would better be member declarations like field declarations and method declarations. Figure 4 shows an example use of the `SelfCode` annotation. In line 1–3 we define two test variables named  $o$  and  $x$  to denote two players. A `SelfCode` annotation in lines 5–15 defines a helper method named `mark` that takes a sequence of numbers encoding a board configuration and creates a new board object of the given configuration. The argument values represents rows of a board with each number encoding the mark of a place, 0 for unmarked, 1 for marked by the player  $o$  and 2 for marked by the player  $x$ . In lines 19–27, the helper method is used to define test data for the `isWonBy` method.

One caveat of the `SelfCode` annotation is that the current annotation processor doesn't perform any checks (e.g., syntactic or static checks) on the code. If there are any errors in the code, they will show up only when the generated Junit class is compiled. Thus, the annotation will work better if the code is simple or you use an IDE like Eclipse [4] that supports incremental and automatic compilation so that you receive errors immediately. If you have more complicate code, you can write it initially as regular Java code—especially when you are still developing your tests—and once the code compiles or your tests are fully developed, you can turn it into an annotation.

## IV. SELFTEST ANNOTATIONS

In this section we briefly describe all the annotation types supported by `SelfTest`. These include `Test`, `SelfTest`, `Var`, `SelfVar`, `SelfImport`, and `SelfCode`.

### A. Test

This annotation defines a single test case for a method or constructor; use `SelfTest` annotation to define more than one test case (see Section IV-B below). The annotated method or

```

1  @SelfVar({
2  @Var(type=Player.class, name="o", value="new Player("\O\")),
3  @Var(type=Player.class, name="x", value="new Player("\X\"))})
4
5  @SelfCode(
6  "private Board mark(int... marks) {
7  Board board = new Board();
8  for (int i = 0; i < marks.length; i++) {
9  if (marks[i] != 0) {
10     int x = i % 3, y = i / 3;
11     mark(x, y, marks[i] == 1 ? this.o : this.x);
12  }
13  }
14  return board;
15  }")
16
17  public class Board {
18
19  @SelfTest({
20  @Test(self="mark(1,1,1, 0,2,2, 0,0,0)", in="o", out="true"),
21  @Test(self="mark(1,1,1, 0,2,2, 0,0,0)", in="x", out="false"),
22  @Test(self="mark(2,1,0, 2,0,1, 2,0,1)", in="o", out="false"),
23  @Test(self="mark(2,1,0, 2,0,1, 2,0,1)", in="x", out="true"),
24  @Test(self="mark(1,2,2, 0,1,0, 0,2,1)", in="o", out="true"),
25  @Test(self="mark(1,2,2, 0,1,0, 0,2,1)", in="x", out="false"),
26  @Test(self="mark(1,0,2, 1,2,0, 2,1,0)", in="o", out="false"),
27  @Test(self="mark(1,0,2, 1,2,0, 2,1,0)", in="x", out="true")})
28  public boolean isWonBy(Player p) { ... }
29  }

```

Fig. 4. SelfCode annotations

constructor should be concrete and non-private. The annotation has the following named elements.

- `Var[] var`: local variables whose scopes are the test case being defined (refer to Section IV-C below for the `Var` annotation type).
- `String self`: value for the implicit argument (receiver) for a non-static method under test.
- `String[] in`: values for the parameters of the method or constructor under test.
- `String out`: expected return value for a non-void method.
- `String[] post`: conditions that must be true upon termination of the method or constructor under test.
- `Class<? extends Throwable> err`: exception that must be thrown by the method or constructor under test.

All the elements are optional, and if no element is specified, an empty test method, a test method with no body, will be generated. For a non-empty test case, a test oracle—at least one of *out*, *post* and *err*—must be specified; *err* cannot be used together with *out* or *post*.

The value specified by the *self* element can be referred in annotations by using the pseudo variable *self*; similarly the return value of a non-void method or the object initialized by a constructor can be referred to by using the pseudo variable *result*.

### B. SelfTest

This annotation defines a list of test cases for a method or constructor. The annotated method or constructor should be concrete and non-private. The annotation has the following unnamed element.

- `Test[] value`: list of test cases (refer to Section IV-A for the `Test` annotation type).

Another use of this annotation is to indicate that a class should be processed for generating its test class; for this, the annotation should have an empty value and the annotated class must be concrete and non-private. Similarly, an annotation with an empty value attached to a method or constructor will trigger generation of an empty test method.

### C. Var

This annotation defines a variable that can be referred to in other annotations, allowing to share test values and data. It can annotate a class, method, or constructor, and has the following named elements.

- `Class<?> type`: type of the variable.
- `String name`: name of the variable.
- `String value`: value of the variable.
- `boolean isStatic`: true if the variable is static; the default is false.

A `Var` annotation annotating a class is translated to a class field, and thus it can be shared by all test cases for the class under test. A `Var` annotation annotating a method or constructor is translated to a local variable, and thus it can be shared only by the test cases for the annotated method or constructor. A local variable may shadow a field, and a shadowed field can be referred to by fully qualifying it, e.g., *this.x* and *TSelfTest.x*.

### D. SelfVar

This annotation defines a list of variables that can be referred to in other annotations, allowing to share test values and data. It can annotate a class, method, or constructor, and has the following unnamed element.

- `Var[] value`: list of variables (refer to Section IV-A for the `Var` annotation type).

Variables are translated to either fields or local variables, allowing different levels of sharing (refer to Section IV-A).

### E. SelfImport

This annotation defines classes to be imported by the generated test class. It annotates a class and has the following unnamed element.

- `String[] value`: names of classes to be imported.

The names should conform to those of Java import statements, e.g., *java.util.List* and *java.util.\**.

### F. SelfCode

This annotation defines a chunk of Java code to be dumped to the generated test class. It annotates a class and has the following unnamed element.

- `String[] value`: Java statements.

The code will be dumped to the generated test class at the member level (e.g., fields and methods) as given without being checked or processed.

## G. Reserved Names

The following two names are reserved and can be used in annotations.

- self: denote the receiver (a.k.a. the implicit argument) of a non-static method under test.
- result: denote the return value for a non-void method or the object just initialized for a constructor under test.

## V. SUMMARY

This document provided a tutorial introduction to SelfTest annotations that allow Java programmers to embed test cases in source code. The SelfTest annotation processor translates the embedded test cases to executable JUnit tests and thus can automate unit testing of Java classes significantly.

However, there are several shortcomings or limitations on the current SelfTest annotations and the annotation processor. One inherent limitation is that in Java only values of primitive and *java.lang.Class* types are allowed in annotations, and thus test values or data should be expressed as strings, affecting writability and readability of annotated test cases. And the current annotation processor doesn't perform much syntactic and static checks on these expressions written in strings; this, however, is an engineering challenge, not an inherent limitation. Expressiveness of annotations may be a concern, as test values or data must be written in a single expression. Although an extended form of expressions, e.g.,  $e_0; m_1(e_1); \dots; m_n(e_n)$ , and test variables may be used, it is often difficult to express complicate test values or data, e.g., complex composite objects, concisely in a single expression.

Another limitation is that the unit of testing is a method or constructor; each method or constructor is tested separately in isolation. It is impossible to test several methods together or protocol aspects of a class, e.g., allowed sequences of method calls. However, note that it is never the intention of SelfTest to completely replace human-based testing but to complement or even strengthen it by helping programmers focus more on interesting, challenging or creative aspects of testing. Nevertheless we expect that the SelfTest annotations makes testing more efficient by automating most of tedious, time-consuming, mundane testing chores and provides a step toward writing self-testing Java classes.

## REFERENCES

- [1] K. Beck. *Extreme Programming Explained*. Addison-Wesley, 2000.
- [2] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.
- [3] J. Gosling et al. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley, 2013.
- [4] Eclipse Foundation. Eclipse. <http://www.eclipse.org>. Date retrieved: March 26, 2014.
- [5] JUnit.org. Junit: A programmer-oriented testing framework for Java. <http://www.junit.org>. Date retrieved: March 26, 2014.
- [6] Oracle. Java Platform, Standard Edition. <http://www.oracle.com/technetwork/java/javase>. Date retrieved: March 26, 2014.
- [7] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, second edition, 204.

## APPENDIX

SelfTest is distributed as a single jar file and is available from: <http://www.cs.utep.edu/cheon/download/selftest/>.

## A. Configuring the javac Compiler

Since Java 6, annotation processing is integrated into the javac compiler, known as pluggable annotation processing [6]. The compiler automatically searches for annotation processors unless disabled with `-proc:none` option, and processors can also be specified explicitly with `-processor` option. Thus, all you have to do is to add the downloaded jar file to your CLASSPATH or specify it using `-classpath` (`-cp`) option; the SelfTest jar file is packaged in such a way that it triggers the Java compiler to search for the included annotation processor. You will also need to add the JUnit 4 jar file to your CLASSPATH because generated source code needs JUnit classes for its compilation. A shell script like below will be useful; replace colons with semicolons on Cygwin.

```
#!/bin/sh
LIB="libs"
CP="$CLASSPATH:$LIB/selftest.jar:$LIB/junit.jar"
exec javac -cp "$CP" "$@"
```

## B. Configuring Eclipse

You can configure an Eclipse project to use SelfTest annotations [4]. There may be several different ways to achieve this, but one way is to perform the following two steps; these steps are tested on Eclipse Juno (4.2.2) and Kepler (4.3.2) [4].

- 1) Add selftest.jar and JUnit 4 to the "Java Build Path." To do this, first click Project→Properties>Java Build Path and then select Libraries>Add External Jars... to add selftest.jar and Libraries>Add Library... to add JUnit 4. It may be a good idea to create a directory named libs in your project to store or import selftest.jar there first.
- 2) Enable annotation processing for your project. First, go to Project→Properties>Java Compiler>Annotation Processing and check all the check boxes, i.e., "Enable project specific settings", "Enable annotation processing", and "Enable processing in editor". It is also recommended to change the generated source directory to a name that does start with a dot so that it won't be hidden in the package explorer; the default is `.apt_generated`. And then configure the so-called factory path by expanding Annotation Processing and selecting Factory Path.
  - a) Check the "Enable project specific settings" box.
  - b) Click the "Add External Jars..." item and add the SelfTest jar file, selftest.jar.
  - c) Once added, select the newly added jar file and click the "Advanced..." item. From the Advanced Options dialog that shows contained annotation processors, select `edu.utep.selftest.MainProcessor`.

This configuration enables Eclipse to recognize SelfTest annotations by showing annotation errors in the built-in Java editor and generating/compiling a JUnit test class whenever annotated Java source code is saved by the editor.