CSE 2231 – Software 2: Software Development and Design

Professor: Rob LaTour

Project #7

Program and Statement Kernel Implementations / Implementation of Program and Statement Kernels

---

Date of Submission: March 24th, 2023

The Ohio State University

College of Engineering

Columbus, Ohio

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

import org.junit.Test;

import components.queue.Queue;
import components.simplereader.SimpleReader;
import components.simplereader.SimpleReader1L;
import components.statement.Statement;
import components.statement.StatementKernel.Condition;
import components.statement.StatementKernel.Kind;
import components.utilities.Tokenizer;

/**
 * JUnit test fixture for {@code Statement}'s constructor and kernel methods.
 *
 * @author Wayne Heym (heym.1@osu.edu)
 * @author Danny Kan (kan.74@osu.edu)
 * @author Jatin Mamtani (mamtani.6@osu.edu)
 *
 */
public abstract class StatementTest {

    /**
     * The name of a file containing a sequence of BL statements.
     */
    private static final String STATEMENT_SAMPLE = "data/statement-sample.bl";
    /**
     * The name of a file containing a sequence of BL statements.
     */
    private static final String STATEMENT_TEST_1 = "data/statement-test1.bl";
    /**
     * The name of a file containing a sequence of BL statements.
```

```
 */
private static final String STATEMENT_TEST_2 = "data/statement-test2.bl";
/**
 * The name of a file containing a sequence of BL statements.
 */
private static final String STATEMENT_TEST_3 = "data/statement-test3.bl";

/**
 * Invokes the {@code Statement} constructor for the implementation under
 * test and returns the result.
 *
 * @return the new statement
 * @ensures constructor = compose((BLOCK, ?, ?), <>)
 */
protected abstract Statement constructorTest();

/**
 * Invokes the {@code Statement} constructor for the reference
 * implementation and returns the result.
 *
 * @return the new statement
 * @ensures constructor = compose((BLOCK, ?, ?), <>)
 */
protected abstract Statement constructorRef();

/**
 *
 * Creates and returns a block {@code Statement}, of the type of the
 * implementation under test, from the file with the given name.
 *
 * @param filename
 *            the name of the file to be parsed for the sequence of
```

```
     *            statements to go in the block statement
     * @return the constructed block statement
     * @ensures <pre>
     * createFromFile = [the block statement containing the statements
     * parsed from the file]
     * </pre>
     */
    private Statement createFromFileTest(String filename) {
        Statement s = this.constructorTest();
        SimpleReader file = new SimpleReader1L(filename);
        Queue<String> tokens = Tokenizer.tokens(file);
        s.parseBlock(tokens);
        file.close();
        return s;
    }

    /**
     *
     * Creates and returns a block {@code Statement}, of the reference
     * implementation type, from the file with the given name.
     *
     * @param filename
     *            the name of the file to be parsed for the sequence of
     *            statements to go in the block statement
     * @return the constructed block statement
     * @ensures <pre>
     * createFromFile = [the block statement containing the statements
     * parsed from the file]
     * </pre>
     */
    private Statement createFromFileRef(String filename) {
        Statement s = this.constructorRef();
```

```
    SimpleReader file = new SimpleReader1L(filename);
    Queue<String> tokens = Tokenizer.tokens(file);
    s.parseBlock(tokens);
    file.close();
    return s;
}

/**
 * Test constructor.
 */
@Test
public final void testConstructor() {
    /*
     * Setup
     */
    Statement sRef = this.constructorRef();

    /*
     * The call
     */
    Statement sTest = this.constructorTest();

    /*
     * Evaluation
     */
    assertEquals(sRef, sTest);
}

/**
 * Test kind of a WHILE statement.
 */
@Test
```

```
public final void testKindWhile() {
    /*
     * Setup
     */
    final int whilePos = 3;
    Statement sourceTest = this.createFromFileTest(STATEMENT_SAMPLE);
    Statement sourceRef = this.createFromFileRef(STATEMENT_SAMPLE);
    Statement sTest = sourceTest.removeFromBlock(whilePos);
    Statement sRef = sourceRef.removeFromBlock(whilePos);
    Kind kRef = sRef.kind();

    /*
     * The call
     */
    Kind kTest = sTest.kind();

    /*
     * Evaluation
     */
    assertEquals(kRef, kTest);
    assertEquals(sRef, sTest);
}

/**
 * Test kind of a WHILE statement.
 */
@Test
public final void testKindWhileStatementTest1() {
    final int whilePos = 2;
    Statement sourceTest = this.createFromFileTest(STATEMENT_TEST_1);
    Statement sourceRef = this.createFromFileRef(STATEMENT_TEST_1);
    Statement sTest = sourceTest.removeFromBlock(whilePos);
```

```
    Statement sRef = sourceRef.removeFromBlock(whilePos);
    Kind kRef = sRef.kind();
    Kind kTest = sTest.kind();
    assertEquals(kRef, kTest);
    assertEquals(sRef, sTest);
}

/**
 * Test kind of a WHILE statement.
 */
@Test
public final void testKindWhileStatementTest2() {
    final int whilePos = 0;
    Statement sourceTest = this.createFromFileTest(STATEMENT_TEST_2);
    Statement sourceRef = this.createFromFileRef(STATEMENT_TEST_2);
    Statement sTest = sourceTest.removeFromBlock(whilePos);
    Statement sRef = sourceRef.removeFromBlock(whilePos);
    Kind kRef = sRef.kind();
    Kind kTest = sTest.kind();
    assertEquals(kRef, kTest);
    assertEquals(sRef, sTest);
}

/**
 * Test kind of a WHILE statement.
 */
@Test
public final void testKindWhileStatementTest3() {
    final int whilePos = 2;
    Statement sourceTest = this.createFromFileTest(STATEMENT_TEST_3);
    Statement sourceRef = this.createFromFileRef(STATEMENT_TEST_3);
    Statement sTest = sourceTest.removeFromBlock(whilePos);
```

```
    Statement sRef = sourceRef.removeFromBlock(whilePos);
    Kind kRef = sRef.kind();
    Kind kTest = sTest.kind();
    assertEquals(kRef, kTest);
    assertEquals(sRef, sTest);
}

/**
 * Test addToBlock at an interior position.
 */
@Test
public final void testAddToBlockInterior() {
    /*
     * Setup
     */
    Statement sTest = this.createFromFileTest(STATEMENT_SAMPLE);
    Statement sRef = this.createFromFileRef(STATEMENT_SAMPLE);
    Statement emptyBlock = sRef.newInstance();
    Statement nestedTest = sTest.removeFromBlock(1);
    Statement nestedRef = sRef.removeFromBlock(1);
    sRef.addToBlock(2, nestedRef);

    /*
     * The call
     */
    sTest.addToBlock(2, nestedTest);

    /*
     * Evaluation
     */
    assertEquals(emptyBlock, nestedTest);
    assertEquals(sRef, sTest);
```

```
}

/**
 * Test addToBlock at an exterior position.
 */
@Test
public final void testAddToBlockExteriorStatementTest1() {
    Statement sTest = this.createFromFileTest(STATEMENT_TEST_1);
    Statement sRef = this.createFromFileRef(STATEMENT_TEST_1);
    Statement emptyBlock = sRef.newInstance();
    Statement nestedTest = sTest.removeFromBlock(1);
    Statement nestedRef = sRef.removeFromBlock(1);
    sRef.addToBlock(2, nestedRef);
    sTest.addToBlock(2, nestedTest);
    assertEquals(emptyBlock, nestedTest);
    assertEquals(sRef, sTest);
}

/**
 * Test addToBlock at an interior position.
 */
@Test
public final void testAddToBlockInteriorStatementTest3() {
    Statement sTest = this.createFromFileTest(STATEMENT_TEST_3);
    Statement sRef = this.createFromFileRef(STATEMENT_TEST_3);
    Statement emptyBlock = sRef.newInstance();
    Statement nestedTest = sTest.removeFromBlock(1);
    Statement nestedRef = sRef.removeFromBlock(1);
    sRef.addToBlock(2, nestedRef);
    sTest.addToBlock(2, nestedTest);
    assertEquals(emptyBlock, nestedTest);
    assertEquals(sRef, sTest);
```

```
}

/**
 * Test removeFromBlock at the front leaving a non-empty block behind.
 */
@Test
public final void testRemoveFromBlockFrontLeavingNonEmpty() {
    /*
     * Setup
     */
    Statement sTest = this.createFromFileTest(STATEMENT_SAMPLE);
    Statement sRef = this.createFromFileRef(STATEMENT_SAMPLE);
    Statement nestedRef = sRef.removeFromBlock(0);

    /*
     * The call
     */
    Statement nestedTest = sTest.removeFromBlock(0);

    /*
     * Evaluation
     */
    assertEquals(sRef, sTest);
    assertEquals(nestedRef, nestedTest);
}

/**
 * Test removeFromBlock at the front leaving a non-empty block behind.
 */
@Test
public final void testRemoveFromBlockFrontLeavingNonEmptyStatementTest1() {
    Statement sTest = this.createFromFileTest(STATEMENT_TEST_1);
```

```
    Statement sRef = this.createFromFileRef(STATEMENT_TEST_1);
    Statement nestedRef = sRef.removeFromBlock(0);
    Statement nestedTest = sTest.removeFromBlock(0);
    assertEquals(sRef, sTest);
    assertEquals(nestedRef, nestedTest);
}

/**
 * Test removeFromBlock at the front leaving a non-empty block behind.
 */
@Test
public final void testRemoveFromBlockRearLeavingNonEmptyStatementTest1() {
    Statement sTest = this.createFromFileTest(STATEMENT_TEST_1);
    Statement sRef = this.createFromFileRef(STATEMENT_TEST_1);
    Statement nestedRef = sRef.removeFromBlock(1);
    Statement nestedTest = sTest.removeFromBlock(1);
    assertEquals(sRef, sTest);
    assertEquals(nestedRef, nestedTest);
}

/**
 * Test removeFromBlock at the front leaving an empty block behind.
 */
@Test
public final void testRemoveFromBlockFrontLeavingEmptyStatementTest2() {
    Statement sTest = this.createFromFileTest(STATEMENT_TEST_2);
    Statement sRef = this.createFromFileRef(STATEMENT_TEST_2);
    Statement nestedRef = sRef.removeFromBlock(0);
    Statement nestedTest = sTest.removeFromBlock(0);
    assertEquals(sRef, sTest);
    assertEquals(nestedRef, nestedTest);
}
```

```
/**
 * Test removeFromBlock at the front leaving a non-empty block behind.
 */
@Test
public final void testRemoveFromBlockFrontLeavingNonEmptyStatementTest3() {
    Statement sTest = this.createFromFileTest(STATEMENT_TEST_3);
    Statement sRef = this.createFromFileRef(STATEMENT_TEST_3);
    Statement nestedRef = sRef.removeFromBlock(0);
    Statement nestedTest = sTest.removeFromBlock(0);
    assertEquals(sRef, sTest);
    assertEquals(nestedRef, nestedTest);
}

/**
 * Test removeFromBlock at the front leaving a non-empty block behind.
 */
@Test
public final void testRemoveFromBlockRearLeavingNonEmptyStatementTest3() {
    Statement sTest = this.createFromFileTest(STATEMENT_TEST_3);
    Statement sRef = this.createFromFileRef(STATEMENT_TEST_3);
    Statement nestedRef = sRef.removeFromBlock(0);
    Statement nestedTest = sTest.removeFromBlock(0);
    assertEquals(sRef, sTest);
    assertEquals(nestedRef, nestedTest);
}

/**
 * Test lengthOfBlock, greater than zero.
 */
@Test
public final void testLengthOfBlockNonEmpty() {
```

```
    /*
     * Setup
     */
    Statement sTest = this.createFromFileTest(STATEMENT_SAMPLE);
    Statement sRef = this.createFromFileRef(STATEMENT_SAMPLE);
    int lengthRef = sRef.lengthOfBlock();

    /*
     * The call
     */
    int lengthTest = sTest.lengthOfBlock();

    /*
     * Evaluation
     */
    assertEquals(lengthRef, lengthTest);
    assertEquals(sRef, sTest);
}

/**
 * Test lengthOfBlock, greater than zero.
 */
@Test
public final void testLengthOfBlockNonEmptyStatementTest1() {
    Statement sTest = this.createFromFileTest(STATEMENT_TEST_1);
    Statement sRef = this.createFromFileRef(STATEMENT_TEST_1);
    int lengthRef = sRef.lengthOfBlock();
    int lengthTest = sTest.lengthOfBlock();
    assertEquals(lengthRef, lengthTest);
    assertEquals(sRef, sTest);
}
```

```
/**
 * Test lengthOfBlock, greater than zero.
 */
@Test
public final void testLengthOfBlockNonEmptyStatementTest2() {
    Statement sTest = this.createFromFileTest(STATEMENT_TEST_2);
    Statement sRef = this.createFromFileRef(STATEMENT_TEST_2);
    int lengthRef = sRef.lengthOfBlock();
    int lengthTest = sTest.lengthOfBlock();
    assertEquals(lengthRef, lengthTest);
    assertEquals(sRef, sTest);
}

/**
 * Test lengthOfBlock, greater than zero.
 */
@Test
public final void testLengthOfBlockNonEmptyStatementTest3() {
    Statement sTest = this.createFromFileTest(STATEMENT_TEST_3);
    Statement sRef = this.createFromFileRef(STATEMENT_TEST_3);
    int lengthRef = sRef.lengthOfBlock();
    int lengthTest = sTest.lengthOfBlock();
    assertEquals(lengthRef, lengthTest);
    assertEquals(sRef, sTest);
}

/**
 * Test assembleIf.
 */
@Test
public final void testAssembleIf() {
    /*
```

```
     * Setup
     */
    Statement blockTest = this.createFromFileTest(STATEMENT_SAMPLE);
    Statement blockRef = this.createFromFileRef(STATEMENT_SAMPLE);
    Statement emptyBlock = blockRef.newInstance();
    Statement sourceTest = blockTest.removeFromBlock(1);
    Statement sRef = blockRef.removeFromBlock(1);
    Statement nestedTest = sourceTest.newInstance();
    Condition c = sourceTest.disassembleIf(nestedTest);
    Statement sTest = sourceTest.newInstance();

    /*
     * The call
     */
    sTest.assembleIf(c, nestedTest);

    /*
     * Evaluation
     */
    assertEquals(emptyBlock, nestedTest);
    assertEquals(sRef, sTest);
}

/**
 * Test assembleIf.
 */
@Test
public final void testAssembleIfStatementTest1() {
    Statement blockTest = this.createFromFileTest(STATEMENT_TEST_1);
    Statement blockRef = this.createFromFileRef(STATEMENT_TEST_1);
    Statement emptyBlock = blockRef.newInstance();
    Statement sourceTest = blockTest.removeFromBlock(1);
```

```
    Statement sRef = blockRef.removeFromBlock(1);
    Statement nestedTest = sourceTest.newInstance();
    Condition c = sourceTest.disassembleIf(nestedTest);
    Statement sTest = sourceTest.newInstance();
    sTest.assembleIf(c, nestedTest);
    assertEquals(emptyBlock, nestedTest);
    assertEquals(sRef, sTest);
}

/**
 * Test assembleIf.
 */
@Test
public final void testAssembleIfStatementTest3() {
    Statement blockTest = this.createFromFileTest(STATEMENT_TEST_3);
    Statement blockRef = this.createFromFileRef(STATEMENT_TEST_3);
    Statement emptyBlock = blockRef.newInstance();
    Statement sourceTest = blockTest.removeFromBlock(0);
    Statement sRef = blockRef.removeFromBlock(0);
    Statement nestedTest = sourceTest.newInstance();
    Condition c = sourceTest.disassembleIf(nestedTest);
    Statement sTest = sourceTest.newInstance();
    sTest.assembleIf(c, nestedTest);
    assertEquals(emptyBlock, nestedTest);
    assertEquals(sRef, sTest);
}

/**
 * Test disassembleIf.
 */
@Test
public final void testDisassembleIf() {
```

```
    /*
     * Setup
     */
    Statement blockTest = this.createFromFileTest(STATEMENT_SAMPLE);
    Statement blockRef = this.createFromFileRef(STATEMENT_SAMPLE);
    Statement sTest = blockTest.removeFromBlock(1);
    Statement sRef = blockRef.removeFromBlock(1);
    Statement nestedTest = sTest.newInstance();
    Statement nestedRef = sRef.newInstance();
    Condition cRef = sRef.disassembleIf(nestedRef);

    /*
     * The call
     */
    Condition cTest = sTest.disassembleIf(nestedTest);

    /*
     * Evaluation
     */
    assertEquals(nestedRef, nestedTest);
    assertEquals(sRef, sTest);
    assertEquals(cRef, cTest);
}

/**
 * Test disassembleIf.
 */
@Test
public final void testDisassembleIfStatementTest1() {
    Statement blockTest = this.createFromFileTest(STATEMENT_TEST_1);
    Statement blockRef = this.createFromFileRef(STATEMENT_TEST_1);
    Statement sTest = blockTest.removeFromBlock(1);
```

```
    Statement sRef = blockRef.removeFromBlock(1);
    Statement nestedTest = sTest.newInstance();
    Statement nestedRef = sRef.newInstance();
    Condition cRef = sRef.disassembleIf(nestedRef);
    Condition cTest = sTest.disassembleIf(nestedTest);
    assertEquals(nestedRef, nestedTest);
    assertEquals(sRef, sTest);
    assertEquals(cRef, cTest);
}

/**
 * Test disassembleIf.
 */
@Test
public final void testDisassembleIfStatementTest3() {
    Statement blockTest = this.createFromFileTest(STATEMENT_TEST_3);
    Statement blockRef = this.createFromFileRef(STATEMENT_TEST_3);
    Statement sTest = blockTest.removeFromBlock(0);
    Statement sRef = blockRef.removeFromBlock(0);
    Statement nestedTest = sTest.newInstance();
    Statement nestedRef = sRef.newInstance();
    Condition cRef = sRef.disassembleIf(nestedRef);
    Condition cTest = sTest.disassembleIf(nestedTest);
    assertEquals(nestedRef, nestedTest);
    assertEquals(sRef, sTest);
    assertEquals(cRef, cTest);
}

/**
 * Test assembleIfElse.
 */
@Test
```

```
public final void testAssembleIfElse() {
    /*
     * Setup
     */
    final int ifElsePos = 2;
    Statement blockTest = this.createFromFileTest(STATEMENT_SAMPLE);
    Statement blockRef = this.createFromFileRef(STATEMENT_SAMPLE);
    Statement emptyBlock = blockRef.newInstance();
    Statement sourceTest = blockTest.removeFromBlock(ifElsePos);
    Statement sRef = blockRef.removeFromBlock(ifElsePos);
    Statement thenBlockTest = sourceTest.newInstance();
    Statement elseBlockTest = sourceTest.newInstance();
    Condition cTest = sourceTest.disassembleIfElse(thenBlockTest,
            elseBlockTest);
    Statement sTest = blockTest.newInstance();

    /*
     * The call
     */
    sTest.assembleIfElse(cTest, thenBlockTest, elseBlockTest);

    /*
     * Evaluation
     */
    assertEquals(emptyBlock, thenBlockTest);
    assertEquals(emptyBlock, elseBlockTest);
    assertEquals(sRef, sTest);
}

/**
 * Test assembleIfElse.
 */
```

```
@Test
public final void testAssembleIfElseStatementTest2() {
    final int ifElsePos = 0;
    Statement blockTest = this.createFromFileTest(STATEMENT_TEST_2);
    Statement blockRef = this.createFromFileRef(STATEMENT_TEST_2);
    Statement emptyBlock = blockRef.newInstance();
    Statement sourceTest = blockTest.removeFromBlock(ifElsePos);
    Statement sRef = blockRef.removeFromBlock(ifElsePos);
    Statement thenBlockTest = sourceTest.newInstance();
    Statement elseBlockTest = sourceTest.newInstance();
    Condition cTest = sourceTest.disassembleIfElse(thenBlockTest,
            elseBlockTest);
    Statement sTest = blockTest.newInstance();
    sTest.assembleIfElse(cTest, thenBlockTest, elseBlockTest);
    assertEquals(emptyBlock, thenBlockTest);
    assertEquals(emptyBlock, elseBlockTest);
    assertEquals(sRef, sTest);
}

/**
 * Test assembleIfElse.
 */
@Test
public final void testAssembleIfElseStatementTest3() {
    final int ifElsePos = 2;
    Statement blockTest = this.createFromFileTest(STATEMENT_TEST_3);
    Statement blockRef = this.createFromFileRef(STATEMENT_TEST_3);
    Statement emptyBlock = blockRef.newInstance();
    Statement sourceTest = blockTest.removeFromBlock(ifElsePos);
    Statement sRef = blockRef.removeFromBlock(ifElsePos);
    Statement thenBlockTest = sourceTest.newInstance();
    Statement elseBlockTest = sourceTest.newInstance();
```

```
    Condition cTest = sourceTest.disassembleIfElse(thenBlockTest,
            elseBlockTest);
    Statement sTest = blockTest.newInstance();
    sTest.assembleIfElse(cTest, thenBlockTest, elseBlockTest);
    assertEquals(emptyBlock, thenBlockTest);
    assertEquals(emptyBlock, elseBlockTest);
    assertEquals(sRef, sTest);
}

/**
 * Test disassembleIfElse.
 */
@Test
public final void testDisassembleIfElse() {
    /*
     * Setup
     */
    final int ifElsePos = 2;
    Statement blockTest = this.createFromFileTest(STATEMENT_SAMPLE);
    Statement blockRef = this.createFromFileRef(STATEMENT_SAMPLE);
    Statement sTest = blockTest.removeFromBlock(ifElsePos);
    Statement sRef = blockRef.removeFromBlock(ifElsePos);
    Statement thenBlockTest = sTest.newInstance();
    Statement elseBlockTest = sTest.newInstance();
    Statement thenBlockRef = sRef.newInstance();
    Statement elseBlockRef = sRef.newInstance();
    Condition cRef = sRef.disassembleIfElse(thenBlockRef, elseBlockRef);

    /*
     * The call
     */
    Condition cTest = sTest.disassembleIfElse(thenBlockTest, elseBlockTest);
```

```
    /*
     * Evaluation
     */
    assertEquals(cRef, cTest);
    assertEquals(thenBlockRef, thenBlockTest);
    assertEquals(elseBlockRef, elseBlockTest);
    assertEquals(sRef, sTest);
}

/**
 * Test disassembleIfElse.
 */
@Test
public final void testDisassembleIfElseStatementTest2() {
    final int ifElsePos = 0;
    Statement blockTest = this.createFromFileTest(STATEMENT_TEST_2);
    Statement blockRef = this.createFromFileRef(STATEMENT_TEST_2);
    Statement sTest = blockTest.removeFromBlock(ifElsePos);
    Statement sRef = blockRef.removeFromBlock(ifElsePos);
    Statement thenBlockTest = sTest.newInstance();
    Statement elseBlockTest = sTest.newInstance();
    Statement thenBlockRef = sRef.newInstance();
    Statement elseBlockRef = sRef.newInstance();
    Condition cRef = sRef.disassembleIfElse(thenBlockRef, elseBlockRef);
    Condition cTest = sTest.disassembleIfElse(thenBlockTest, elseBlockTest);
    assertEquals(cRef, cTest);
    assertEquals(thenBlockRef, thenBlockTest);
    assertEquals(elseBlockRef, elseBlockTest);
    assertEquals(sRef, sTest);
}
```

```
/**
 * Test disassembleIfElse.
 */
@Test
public final void testDisassembleIfElseStatementTest3() {
    final int ifElsePos = 2;
    Statement blockTest = this.createFromFileTest(STATEMENT_TEST_3);
    Statement blockRef = this.createFromFileRef(STATEMENT_TEST_3);
    Statement sTest = blockTest.removeFromBlock(ifElsePos);
    Statement sRef = blockRef.removeFromBlock(ifElsePos);
    Statement thenBlockTest = sTest.newInstance();
    Statement elseBlockTest = sTest.newInstance();
    Statement thenBlockRef = sRef.newInstance();
    Statement elseBlockRef = sRef.newInstance();
    Condition cRef = sRef.disassembleIfElse(thenBlockRef, elseBlockRef);
    Condition cTest = sTest.disassembleIfElse(thenBlockTest, elseBlockTest);
    assertEquals(cRef, cTest);
    assertEquals(thenBlockRef, thenBlockTest);
    assertEquals(elseBlockRef, elseBlockTest);
    assertEquals(sRef, sTest);
}

/**
 * Test assembleWhile.
 */
@Test
public final void testAssembleWhile() {
    /*
     * Setup
     */
    Statement blockTest = this.createFromFileTest(STATEMENT_SAMPLE);
    Statement blockRef = this.createFromFileRef(STATEMENT_SAMPLE);
```

```
    Statement emptyBlock = blockRef.newInstance();
    Statement sourceTest = blockTest.removeFromBlock(1);
    Statement sourceRef = blockRef.removeFromBlock(1);
    Statement nestedTest = sourceTest.newInstance();
    Statement nestedRef = sourceRef.newInstance();
    Condition cTest = sourceTest.disassembleIf(nestedTest);
    Condition cRef = sourceRef.disassembleIf(nestedRef);
    Statement sRef = sourceRef.newInstance();
    sRef.assembleWhile(cRef, nestedRef);
    Statement sTest = sourceTest.newInstance();

    /*
     * The call
     */
    sTest.assembleWhile(cTest, nestedTest);

    /*
     * Evaluation
     */
    assertEquals(emptyBlock, nestedTest);
    assertEquals(sRef, sTest);
}

/**
 * Test assembleWhile.
 */
@Test
public final void testAssembleWhileStatementTest1() {
    Statement blockTest = this.createFromFileTest(STATEMENT_TEST_1);
    Statement blockRef = this.createFromFileRef(STATEMENT_TEST_1);
    Statement emptyBlock = blockRef.newInstance();
    Statement sourceTest = blockTest.removeFromBlock(1);
```

```
    Statement sourceRef = blockRef.removeFromBlock(1);
    Statement nestedTest = sourceTest.newInstance();
    Statement nestedRef = sourceRef.newInstance();
    Condition cTest = sourceTest.disassembleIf(nestedTest);
    Condition cRef = sourceRef.disassembleIf(nestedRef);
    Statement sRef = sourceRef.newInstance();
    sRef.assembleWhile(cRef, nestedRef);
    Statement sTest = sourceTest.newInstance();
    sTest.assembleWhile(cTest, nestedTest);
    assertEquals(emptyBlock, nestedTest);
    assertEquals(sRef, sTest);
}

/**
 * Test assembleWhile.
 */
@Test
public final void testAssembleWhileStatementTest3() {
    Statement blockTest = this.createFromFileTest(STATEMENT_TEST_3);
    Statement blockRef = this.createFromFileRef(STATEMENT_TEST_3);
    Statement emptyBlock = blockRef.newInstance();
    Statement sourceTest = blockTest.removeFromBlock(0);
    Statement sourceRef = blockRef.removeFromBlock(0);
    Statement nestedTest = sourceTest.newInstance();
    Statement nestedRef = sourceRef.newInstance();
    Condition cTest = sourceTest.disassembleIf(nestedTest);
    Condition cRef = sourceRef.disassembleIf(nestedRef);
    Statement sRef = sourceRef.newInstance();
    sRef.assembleWhile(cRef, nestedRef);
    Statement sTest = sourceTest.newInstance();
    sTest.assembleWhile(cTest, nestedTest);
    assertEquals(emptyBlock, nestedTest);
```

```
    assertEquals(sRef, sTest);
}

/**
 * Test disassembleWhile.
 */
@Test
public final void testDisassembleWhile() {
    /*
     * Setup
     */
    final int whilePos = 3;
    Statement blockTest = this.createFromFileTest(STATEMENT_SAMPLE);
    Statement blockRef = this.createFromFileRef(STATEMENT_SAMPLE);
    Statement sTest = blockTest.removeFromBlock(whilePos);
    Statement sRef = blockRef.removeFromBlock(whilePos);
    Statement nestedTest = sTest.newInstance();
    Statement nestedRef = sRef.newInstance();
    Condition cRef = sRef.disassembleWhile(nestedRef);

    /*
     * The call
     */
    Condition cTest = sTest.disassembleWhile(nestedTest);

    /*
     * Evaluation
     */
    assertEquals(nestedRef, nestedTest);
    assertEquals(sRef, sTest);
    assertEquals(cRef, cTest);
}
```

```
/**
 * Test disassembleWhile.
 */
@Test
public final void testDisassembleWhileStatementTest3() {
    final int whilePos = 1;
    Statement blockTest = this.createFromFileTest(STATEMENT_TEST_3);
    Statement blockRef = this.createFromFileRef(STATEMENT_TEST_3);
    Statement sTest = blockTest.removeFromBlock(whilePos);
    Statement sRef = blockRef.removeFromBlock(whilePos);
    Statement nestedTest = sTest.newInstance();
    Statement nestedRef = sRef.newInstance();
    Condition cRef = sRef.disassembleWhile(nestedRef);
    Condition cTest = sTest.disassembleWhile(nestedTest);
    assertEquals(nestedRef, nestedTest);
    assertEquals(sRef, sTest);
    assertEquals(cRef, cTest);
}

/**
 * Test assembleCall.
 */
@Test
public final void testAssembleCall() {
    /*
     * Setup
     */
    Statement sRef = this.constructorRef().newInstance();
    Statement sTest = this.constructorTest().newInstance();

    String name = "look-for-something";
```

```
    sRef.assembleCall(name);

    /*
     * The call
     */
    sTest.assembleCall(name);

    /*
     * Evaluation
     */
    assertEquals(sRef, sTest);
}

/**
 * Test assembleCall.
 */
@Test
public final void testAssembleCallV1() {
    Statement sRef = this.constructorRef().newInstance();
    Statement sTest = this.constructorTest().newInstance();
    String name = "something-goes-here";
    sRef.assembleCall(name);
    sTest.assembleCall(name);
    assertEquals(sRef, sTest);
}

/**
 * Test assembleCall.
 */
@Test
public final void testAssembleCallV2() {
    Statement sRef = this.constructorRef().newInstance();
```

```
    Statement sTest = this.constructorTest().newInstance();
    String name = "the-ohio-state-university";
    sRef.assembleCall(name);
    sTest.assembleCall(name);
    assertEquals(sRef, sTest);
}

/**
 * Test assembleCall.
 */
@Test
public final void testAssembleCallV3() {
    Statement sRef = this.constructorRef().newInstance();
    Statement sTest = this.constructorTest().newInstance();
    String name = "computer-science";
    sRef.assembleCall(name);
    sTest.assembleCall(name);
    assertEquals(sRef, sTest);
}

/**
 * Test disassembleCall.
 */
@Test
public final void testDisassembleCall() {
    /*
     * Setup
     */
    Statement blockTest = this.createFromFileTest(STATEMENT_SAMPLE);
    Statement blockRef = this.createFromFileRef(STATEMENT_SAMPLE);
    Statement sTest = blockTest.removeFromBlock(0);
    Statement sRef = blockRef.removeFromBlock(0);
```

```
        String nRef = sRef.disassembleCall();

        /*
         * The call
         */
        String nTest = sTest.disassembleCall();

        /*
         * Evaluation
         */
        assertEquals(sRef, sTest);
        assertEquals(nRef, nTest);
    }

    /**
     * Test disassembleCall.
     */
    @Test
    public final void testDisassembleCallStatementTest1() {
        Statement blockTest = this.createFromFileTest(STATEMENT_TEST_1);
        Statement blockRef = this.createFromFileRef(STATEMENT_TEST_1);
        Statement sTest = blockTest.removeFromBlock(0);
        Statement sRef = blockRef.removeFromBlock(0);
        String nRef = sRef.disassembleCall();
        String nTest = sTest.disassembleCall();
        assertEquals(sRef, sTest);
        assertEquals(nRef, nTest);
    }

}
```