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

/**
 * JUnit test fixture for {@code Statement}'s constructor and kernel methods.
 *
 * @author Wayne Heym
 * @author Zhuoyang Li
 *
 */
public abstract class StatementTest {

    /**
     * The name of a file containing a sequence of BL statements.
     */
    private static final String FILE_NAME_1 = "data/statement-sample.bl";

    // TODO - define file names for additional test inputs
    /**
     * The name of a file containing a sequence of BL statements.
     */
    private static final String FILE_NAME_2 = "data/statement-sampleTest.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();

```

```
51      /**
52       *
53       * Creates and returns a block {@code Statement}, of the
    type of the
54       * implementation under test, from the file with the
    given name.
55       *
56       * @param filename
57       *            the name of the file to be parsed for the
    sequence of
58       *            statements to go in the block statement
59       * @return the constructed block statement
60       * @ensures <pre>
61       * createFromFile = [the block statement containing the
    statements
62       * parsed from the file]
63       * </pre>
64       */
65      private Statement createFromFileTest(String filename) {
66          Statement s = this.constructorTest();
67          SimpleReader file = new SimpleReader1L(filename);
68          Queue<String> tokens = Tokenizer.tokens(file);
69          s.parseBlock(tokens);
70          file.close();
71          return s;
72      }
73
74      /**
75       *
76       * Creates and returns a block {@code Statement}, of the
    reference
77       * implementation type, from the file with the given
    name.
78       *
79       * @param filename
80       *            the name of the file to be parsed for the
    sequence of
81       *            statements to go in the block statement
82       * @return the constructed block statement
83       * @ensures <pre>
84       * createFromFile = [the block statement containing the
    statements
85       * parsed from the file]
86       * </pre>
87       */
88      private Statement createFromFileRef(String filename) {
89          Statement s = this.constructorRef();
```

```
 90          SimpleReader file = new SimpleReader1L(filename);
 91          Queue<String> tokens = Tokenizer.tokens(file);
 92          s.parseBlock(tokens);
 93          file.close();
 94          return s;
 95      }
 96
 97      /**
 98       * Test constructor.
 99       */
100      @Test
101      public final void testConstructor() {
102          /*
103           * Setup
104           */
105          Statement sRef = this.constructorRef();
106
107          /*
108           * The call
109           */
110          Statement sTest = this.constructorTest();
111
112          /*
113           * Evaluation
114           */
115          assertEquals(sRef, sTest);
116      }
117
118      /**
119       * Test kind of a WHILE statement.
120       */
121      @Test
122      public final void testKindWhile() {
123          /*
124           * Setup
125           */
126          final int whilePos = 3;
127          Statement sourceTest =
    this.createFromFileTest(FILE_NAME_1);
128          Statement sourceRef =
    this.createFromFileRef(FILE_NAME_1);
129          Statement sTest =
    sourceTest.removeFromBlock(whilePos);
130          Statement sRef = sourceRef.removeFromBlock(whilePos);
131          Kind kRef = sRef.kind();
132
133          /*
```

```
134          * The call
135          */
136         Kind kTest = sTest.kind();
137 
138         /*
139          * Evaluation
140          */
141         assertEquals(kRef, kTest);
142         assertEquals(sRef, sTest);
143     }
144 
145     /**
146      * Test addToBlock at an interior position.
147      */
148     @Test
149     public final void testAddToBlockInterior() {
150         /*
151          * Setup
152          */
153         Statement sTest =
    this.createFromFileTest(FILE_NAME_1);
154         Statement sRef = this.createFromFileRef(FILE_NAME_1);
155         Statement emptyBlock = sRef.newInstance();
156         Statement nestedTest = sTest.removeFromBlock(1);
157         Statement nestedRef = sRef.removeFromBlock(1);
158         sRef.addToBlock(2, nestedRef);
159 
160         /*
161          * The call
162          */
163         sTest.addToBlock(2, nestedTest);
164 
165         /*
166          * Evaluation
167          */
168         assertEquals(emptyBlock, nestedTest);
169         assertEquals(sRef, sTest);
170     }
171 
172     /**
173      * Test removeFromBlock at the front leaving a non-empty
    block behind.
174      */
175     @Test
176     public final void
    testRemoveFromBlockFrontLeavingNonEmpty() {
177         /*
```

```
178          * Setup
179          */
180         Statement sTest =
    this.createFromFileTest(FILE_NAME_1);
181         Statement sRef = this.createFromFileRef(FILE_NAME_1);
182         Statement nestedRef = sRef.removeFromBlock(0);
183 
184         /*
185          * The call
186          */
187         Statement nestedTest = sTest.removeFromBlock(0);
188 
189         /*
190          * Evaluation
191          */
192         assertEquals(sRef, sTest);
193         assertEquals(nestedRef, nestedTest);
194     }
195 
196     /**
197      * Test lengthOfBlock, greater than zero.
198      */
199     @Test
200     public final void testLengthOfBlockNonEmpty() {
201         /*
202          * Setup
203          */
204         Statement sTest =
    this.createFromFileTest(FILE_NAME_1);
205         Statement sRef = this.createFromFileRef(FILE_NAME_1);
206         int lengthRef = sRef.lengthOfBlock();
207 
208         /*
209          * The call
210          */
211         int lengthTest = sTest.lengthOfBlock();
212 
213         /*
214          * Evaluation
215          */
216         assertEquals(lengthRef, lengthTest);
217         assertEquals(sRef, sTest);
218     }
219 
220     /**
221      * Test assembleIf.
222      */
```

```
223     @Test
224     public final void testAssembleIf() {
225         /*
226          * Setup
227          */
228         Statement blockTest =
    this.createFromFileTest(FILE_NAME_1);
229         Statement blockRef =
    this.createFromFileRef(FILE_NAME_1);
230         Statement emptyBlock = blockRef.newInstance();
231         Statement sourceTest = blockTest.removeFromBlock(1);
232         Statement sRef = blockRef.removeFromBlock(1);
233         Statement nestedTest = sourceTest.newInstance();
234         Condition c = sourceTest.disassembleIf(nestedTest);
235         Statement sTest = sourceTest.newInstance();
236 
237         /*
238          * The call
239          */
240         sTest.assembleIf(c, nestedTest);
241 
242         /*
243          * Evaluation
244          */
245         assertEquals(emptyBlock, nestedTest);
246         assertEquals(sRef, sTest);
247     }
248 
249     /**
250      * Test disassembleIf.
251      */
252     @Test
253     public final void testDisassembleIf() {
254         /*
255          * Setup
256          */
257         Statement blockTest =
    this.createFromFileTest(FILE_NAME_1);
258         Statement blockRef =
    this.createFromFileRef(FILE_NAME_1);
259         Statement sTest = blockTest.removeFromBlock(1);
260         Statement sRef = blockRef.removeFromBlock(1);
261         Statement nestedTest = sTest.newInstance();
262         Statement nestedRef = sRef.newInstance();
263         Condition cRef = sRef.disassembleIf(nestedRef);
264 
265         /*
```

```
266          * The call
267          */
268         Condition cTest = sTest.disassembleIf(nestedTest);
269 
270         /*
271          * Evaluation
272          */
273         assertEquals(nestedRef, nestedTest);
274         assertEquals(sRef, sTest);
275         assertEquals(cRef, cTest);
276     }
277 
278     /**
279      * Test assembleIfElse.
280      */
281     @Test
282     public final void testAssembleIfElse() {
283         /*
284          * Setup
285          */
286         final int ifElsePos = 2;
287         Statement blockTest =
    this.createFromFileTest(FILE_NAME_1);
288         Statement blockRef =
    this.createFromFileRef(FILE_NAME_1);
289         Statement emptyBlock = blockRef.newInstance();
290         Statement sourceTest =
    blockTest.removeFromBlock(ifElsePos);
291         Statement sRef = blockRef.removeFromBlock(ifElsePos);
292         Statement thenBlockTest = sourceTest.newInstance();
293         Statement elseBlockTest = sourceTest.newInstance();
294         Condition cTest =
    sourceTest.disassembleIfElse(thenBlockTest,
295                 elseBlockTest);
296         Statement sTest = blockTest.newInstance();
297 
298         /*
299          * The call
300          */
301         sTest.assembleIfElse(cTest, thenBlockTest,
    elseBlockTest);
302 
303         /*
304          * Evaluation
305          */
306         assertEquals(emptyBlock, thenBlockTest);
307         assertEquals(emptyBlock, elseBlockTest);
```

```
308         assertEquals(sRef, sTest);
309     }
310 
311     /**
312      * Test disassembleIfElse.
313      */
314     @Test
315     public final void testDisassembleIfElse() {
316         /*
317          * Setup
318          */
319         final int ifElsePos = 2;
320         Statement blockTest =
    this.createFromFileTest(FILE_NAME_1);
321         Statement blockRef =
    this.createFromFileRef(FILE_NAME_1);
322         Statement sTest =
    blockTest.removeFromBlock(ifElsePos);
323         Statement sRef = blockRef.removeFromBlock(ifElsePos);
324         Statement thenBlockTest = sTest.newInstance();
325         Statement elseBlockTest = sTest.newInstance();
326         Statement thenBlockRef = sRef.newInstance();
327         Statement elseBlockRef = sRef.newInstance();
328         Condition cRef = sRef.disassembleIfElse(thenBlockRef,
    elseBlockRef);
329 
330         /*
331          * The call
332          */
333         Condition cTest =
    sTest.disassembleIfElse(thenBlockTest, elseBlockTest);
334 
335         /*
336          * Evaluation
337          */
338         assertEquals(cRef, cTest);
339         assertEquals(thenBlockRef, thenBlockTest);
340         assertEquals(elseBlockRef, elseBlockTest);
341         assertEquals(sRef, sTest);
342     }
343 
344     /**
345      * Test assembleWhile.
346      */
347     @Test
348     public final void testAssembleWhile() {
349         /*
```

```
350         * Setup
351         */
352        Statement blockTest =
    this.createFromFileTest(FILE_NAME_1);
353        Statement blockRef =
    this.createFromFileRef(FILE_NAME_1);
354        Statement emptyBlock = blockRef.newInstance();
355        Statement sourceTest = blockTest.removeFromBlock(1);
356        Statement sourceRef = blockRef.removeFromBlock(1);
357        Statement nestedTest = sourceTest.newInstance();
358        Statement nestedRef = sourceRef.newInstance();
359        Condition cTest =
    sourceTest.disassembleIf(nestedTest);
360        Condition cRef = sourceRef.disassembleIf(nestedRef);
361        Statement sRef = sourceRef.newInstance();
362        sRef.assembleWhile(cRef, nestedRef);
363        Statement sTest = sourceTest.newInstance();
364
365        /*
366         * The call
367         */
368        sTest.assembleWhile(cTest, nestedTest);
369
370        /*
371         * Evaluation
372         */
373        assertEquals(emptyBlock, nestedTest);
374        assertEquals(sRef, sTest);
375    }
376
377    /**
378     * Test disassembleWhile.
379     */
380    @Test
381    public final void testDisassembleWhile() {
382        /*
383         * Setup
384         */
385        final int whilePos = 3;
386        Statement blockTest =
    this.createFromFileTest(FILE_NAME_1);
387        Statement blockRef =
    this.createFromFileRef(FILE_NAME_1);
388        Statement sTest =
    blockTest.removeFromBlock(whilePos);
389        Statement sRef = blockRef.removeFromBlock(whilePos);
390        Statement nestedTest = sTest.newInstance();
```

```
391         Statement nestedRef = sRef.newInstance();
392         Condition cRef = sRef.disassembleWhile(nestedRef);
393 
394         /*
395          * The call
396          */
397         Condition cTest = sTest.disassembleWhile(nestedTest);
398 
399         /*
400          * Evaluation
401          */
402         assertEquals(nestedRef, nestedTest);
403         assertEquals(sRef, sTest);
404         assertEquals(cRef, cTest);
405     }
406 
407     /**
408      * Test assembleCall.
409      */
410     @Test
411     public final void testAssembleCall() {
412         /*
413          * Setup
414          */
415         Statement sRef = this.constructorRef().newInstance();
416         Statement sTest =
    this.constructorTest().newInstance();
417 
418         String name = "look-for-something";
419         sRef.assembleCall(name);
420 
421         /*
422          * The call
423          */
424         sTest.assembleCall(name);
425 
426         /*
427          * Evaluation
428          */
429         assertEquals(sRef, sTest);
430     }
431 
432     /**
433      * Test disassembleCall.
434      */
435     @Test
436     public final void testDisassembleCall() {
```

```
437         /*
438          * Setup
439          */
440         Statement blockTest =
    this.createFromFileTest(FILE_NAME_1);
441         Statement blockRef =
    this.createFromFileRef(FILE_NAME_1);
442         Statement sTest = blockTest.removeFromBlock(0);
443         Statement sRef = blockRef.removeFromBlock(0);
444         String nRef = sRef.disassembleCall();
445 
446         /*
447          * The call
448          */
449         String nTest = sTest.disassembleCall();
450 
451         /*
452          * Evaluation
453          */
454         assertEquals(sRef, sTest);
455         assertEquals(nRef, nTest);
456     }
457 
458     // TODO - provide additional test cases to thoroughly
    test StatementKernel
459 
460     /**
461      * Test kind of a BLOCK statement.
462      */
463     @Test
464     public final void testKindBlock() {
465         /*
466          * Setup
467          */
468         Statement sourceTest =
    this.createFromFileTest(FILE_NAME_2);
469         Statement sourceRef =
    this.createFromFileRef(FILE_NAME_2);
470         Kind kRef = sourceRef.kind();
471 
472         /*
473          * The call
474          */
475         Kind kTest = sourceTest.kind();
476 
477         /*
478          * Evaluation
```

```
479          */
480         assertEquals(kRef, kTest);
481         assertEquals(sourceRef, sourceTest);
482     }
483 
484     /**
485      * Test addToBlock at an interior position.
486      */
487 
488     @Test
489     public final void testAddToBlockInterior2() {
490         /*
491          * Setup
492          */
493         Statement sTest =
    this.createFromFileTest(FILE_NAME_2);
494         Statement sRef = this.createFromFileRef(FILE_NAME_2);
495         Statement emptyBlock = sRef.newInstance();
496         Statement nestedTest = sTest.removeFromBlock(1);
497         Statement nestedRef = sRef.removeFromBlock(1);
498         sRef.addToBlock(2, nestedRef);
499 
500         /*
501          * The call
502          */
503         sTest.addToBlock(2, nestedTest);
504 
505         /*
506          * Evaluation
507          */
508         assertEquals(emptyBlock, nestedTest);
509         assertEquals(sRef, sTest);
510     }
511 
512     /**
513      * Test removeFromBlock at the front leaving a non-empty
    block behind.
514      */
515     @Test
516     public final void
    testRemoveFromBlockFrontLeavingNonEmpty2() {
517         /*
518          * Setup
519          */
520         Statement sTest =
    this.createFromFileTest(FILE_NAME_2);
521         Statement sRef = this.createFromFileRef(FILE_NAME_2);
```

```
522         Statement nestedRef = sRef.removeFromBlock(0);
523 
524         /*
525          * The call
526          */
527         Statement nestedTest = sTest.removeFromBlock(0);
528 
529         /*
530          * Evaluation
531          */
532         assertEquals(sRef, sTest);
533         assertEquals(nestedRef, nestedTest);
534     }
535 
536     /**
537      * Test lengthOfBlock, greater than zero.
538      */
539     @Test
540     public final void testLengthOfBlockNonEmpty2() {
541         /*
542          * Setup
543          */
544         Statement sTest =
    this.createFromFileTest(FILE_NAME_2);
545         Statement sRef = this.createFromFileRef(FILE_NAME_2);
546         int lengthRef = sRef.lengthOfBlock();
547 
548         /*
549          * The call
550          */
551         int lengthTest = sTest.lengthOfBlock();
552 
553         /*
554          * Evaluation
555          */
556         assertEquals(lengthRef, lengthTest);
557         assertEquals(sRef, sTest);
558     }
559 
560     /**
561      * Test assembleIf.
562      */
563     @Test
564     public final void testAssembleIf2() {
565         /*
566          * Setup
567          */
```

```
568         Statement blockTest =
    this.createFromFileTest(FILE_NAME_2);
569         Statement blockRef =
    this.createFromFileRef(FILE_NAME_2);
570         Statement emptyBlock = blockRef.newInstance();
571         Statement sourceTest = blockTest.removeFromBlock(1);
572         Statement sRef = blockRef.removeFromBlock(1);
573         Statement nestedTest = sourceTest.newInstance();
574         Condition c = sourceTest.disassembleIf(nestedTest);
575         Statement sTest = sourceTest.newInstance();
576
577         /*
578          * The call
579          */
580         sTest.assembleIf(c, nestedTest);
581
582         /*
583          * Evaluation
584          */
585         assertEquals(emptyBlock, nestedTest);
586         assertEquals(sRef, sTest);
587     }
588
589     /**
590      * Test disassembleIf.
591      */
592     @Test
593     public final void testDisassembleIf2() {
594         /*
595          * Setup
596          */
597         Statement blockTest =
    this.createFromFileTest(FILE_NAME_2);
598         Statement blockRef =
    this.createFromFileRef(FILE_NAME_2);
599         Statement sTest = blockTest.removeFromBlock(1);
600         Statement sRef = blockRef.removeFromBlock(1);
601         Statement nestedTest = sTest.newInstance();
602         Statement nestedRef = sRef.newInstance();
603         Condition cRef = sRef.disassembleIf(nestedRef);
604
605         /*
606          * The call
607          */
608         Condition cTest = sTest.disassembleIf(nestedTest);
609
610         /*
```

```
611          * Evaluation
612          */
613         assertEquals(nestedRef, nestedTest);
614         assertEquals(sRef, sTest);
615         assertEquals(cRef, cTest);
616     }
617
618     /**
619      * Test assembleIfElse.
620      */
621     @Test
622     public final void testAssembleIfElse2() {
623         /*
624          * Setup
625          */
626         final int ifElsePos = 2;
627         Statement blockTest =
    this.createFromFileTest(FILE_NAME_2);
628         Statement blockRef =
    this.createFromFileRef(FILE_NAME_2);
629         Statement emptyBlock = blockRef.newInstance();
630         Statement sourceTest =
    blockTest.removeFromBlock(ifElsePos);
631         Statement sRef = blockRef.removeFromBlock(ifElsePos);
632         Statement thenBlockTest = sourceTest.newInstance();
633         Statement elseBlockTest = sourceTest.newInstance();
634         Condition cTest =
    sourceTest.disassembleIfElse(thenBlockTest,
635                 elseBlockTest);
636         Statement sTest = blockTest.newInstance();
637
638         /*
639          * The call
640          */
641         sTest.assembleIfElse(cTest, thenBlockTest,
    elseBlockTest);
642
643         /*
644          * Evaluation
645          */
646         assertEquals(emptyBlock, thenBlockTest);
647         assertEquals(emptyBlock, elseBlockTest);
648         assertEquals(sRef, sTest);
649     }
650
651     /**
652      * Test disassembleIfElse.
```

```
653      */
654     @Test
655     public final void testDisassembleIfElse2() {
656         /*
657          * Setup
658          */
659         final int ifElsePos = 2;
660         Statement blockTest =
    this.createFromFileTest(FILE_NAME_2);
661         Statement blockRef =
    this.createFromFileRef(FILE_NAME_2);
662         Statement sTest =
    blockTest.removeFromBlock(ifElsePos);
663         Statement sRef = blockRef.removeFromBlock(ifElsePos);
664         Statement thenBlockTest = sTest.newInstance();
665         Statement elseBlockTest = sTest.newInstance();
666         Statement thenBlockRef = sRef.newInstance();
667         Statement elseBlockRef = sRef.newInstance();
668         Condition cRef = sRef.disassembleIfElse(thenBlockRef,
    elseBlockRef);
669 
670         /*
671          * The call
672          */
673         Condition cTest =
    sTest.disassembleIfElse(thenBlockTest, elseBlockTest);
674 
675         /*
676          * Evaluation
677          */
678         assertEquals(cRef, cTest);
679         assertEquals(thenBlockRef, thenBlockTest);
680         assertEquals(elseBlockRef, elseBlockTest);
681         assertEquals(sRef, sTest);
682     }
683 
684     /**
685      * Test assembleWhile.
686      */
687     @Test
688     public final void testAssembleWhile2() {
689         /*
690          * Setup
691          */
692         Statement blockTest =
    this.createFromFileTest(FILE_NAME_2);
693         Statement blockRef =
```

```
    this.createFromFileRef(FILE_NAME_2);
694         Statement emptyBlock = blockRef.newInstance();
695         Statement sourceTest = blockTest.removeFromBlock(1);
696         Statement sourceRef = blockRef.removeFromBlock(1);
697         Statement nestedTest = sourceTest.newInstance();
698         Statement nestedRef = sourceRef.newInstance();
699         Condition cTest =
    sourceTest.disassembleIf(nestedTest);
700         Condition cRef = sourceRef.disassembleIf(nestedRef);
701         Statement sRef = sourceRef.newInstance();
702         sRef.assembleWhile(cRef, nestedRef);
703         Statement sTest = sourceTest.newInstance();
704
705         /*
706          * The call
707          */
708         sTest.assembleWhile(cTest, nestedTest);
709
710         /*
711          * Evaluation
712          */
713         assertEquals(emptyBlock, nestedTest);
714         assertEquals(sRef, sTest);
715     }
716
717     /**
718      * Test disassembleWhile.
719      */
720     @Test
721     public final void testDisassembleWhile2() {
722         /*
723          * Setup
724          */
725         final int whilePos = 3;
726         Statement blockTest =
    this.createFromFileTest(FILE_NAME_2);
727         Statement blockRef =
    this.createFromFileRef(FILE_NAME_2);
728         Statement sTest =
    blockTest.removeFromBlock(whilePos);
729         Statement sRef = blockRef.removeFromBlock(whilePos);
730         Statement nestedTest = sTest.newInstance();
731         Statement nestedRef = sRef.newInstance();
732         Condition cRef = sRef.disassembleWhile(nestedRef);
733
734         /*
735          * The call
```

```
736          */
737         Condition cTest = sTest.disassembleWhile(nestedTest);
738 
739         /*
740          * Evaluation
741          */
742         assertEquals(nestedRef, nestedTest);
743         assertEquals(sRef, sTest);
744         assertEquals(cRef, cTest);
745     }
746 
747     /**
748      * Test assembleCall.
749      */
750     @Test
751     public final void testAssembleCall2() {
752         /*
753          * Setup
754          */
755         Statement sRef = this.constructorRef().newInstance();
756         Statement sTest =
    this.constructorTest().newInstance();
757 
758         String name = "look-for-something";
759         sRef.assembleCall(name);
760 
761         /*
762          * The call
763          */
764         sTest.assembleCall(name);
765 
766         /*
767          * Evaluation
768          */
769         assertEquals(sRef, sTest);
770     }
771 
772     /**
773      * Test disassembleCall.
774      */
775     @Test
776     public final void testDisassembleCall2() {
777         /*
778          * Setup
779          */
780         Statement blockTest =
    this.createFromFileTest(FILE_NAME_2);
```

```
781         Statement blockRef =
    this.createFromFileRef(FILE_NAME_2);
782         Statement sTest = blockTest.removeFromBlock(0);
783         Statement sRef = blockRef.removeFromBlock(0);
784         String nRef = sRef.disassembleCall();
785 
786         /*
787          * The call
788          */
789         String nTest = sTest.disassembleCall();
790 
791         /*
792          * Evaluation
793          */
794         assertEquals(sRef, sTest);
795         assertEquals(nRef, nTest);
796     }
797 
798     /**
799      * Test assembleIfElse.
800      */
801     @Test
802     public final void testAssembleIfElse3() {
803         /*
804          * Setup
805          */
806         final int ifElsePos = 2;
807         Statement blockTest =
    this.createFromFileTest(FILE_NAME_2);
808         Statement blockRef =
    this.createFromFileRef(FILE_NAME_2);
809         Statement emptyBlock = blockRef.newInstance();
810         Statement sourceTest =
    blockTest.removeFromBlock(ifElsePos);
811         Statement sRef = blockRef.removeFromBlock(ifElsePos);
812         Statement thenBlockTest = sourceTest.newInstance();
813         Statement elseBlockTest = sourceTest.newInstance();
814         Condition cTest =
    sourceTest.disassembleIfElse(thenBlockTest,
815                 elseBlockTest);
816         Statement sTest = blockTest.newInstance();
817 
818         /*
819          * The call
820          */
821         sTest.assembleIfElse(cTest, thenBlockTest,
    elseBlockTest);
```

```
822
823         /*
824          * Evaluation
825          */
826         assertEquals(emptyBlock, thenBlockTest);
827         assertEquals(emptyBlock, elseBlockTest);
828         assertEquals(sRef, sTest);
829     }
830
831 }
832
```