```
 1 import static org.junit.Assert.assertEquals;
12
13 /**
14  * JUnit test fixture for {@code Statement}'s constructor and
   kernel methods.
15  *
16  * @author Wayne Heym
17  * @author Put your name here
18  *
19  */
20 public abstract class StatementTest {
21
22     /**
23      * The name of a file containing a sequence of BL
   statements.
24      */
25     private static final String FILE_NAME_1 = "data/statement-
   sample1.bl";
26     private static final String FILE_NAME_2 = "data/statement-
   sample2.bl";
27     private static final String FILE_NAME_3 = "data/statement-
   sample3.bl";
28     private static final String FILE_NAME_4 = "data/statement-
   sample4.bl";
29     private static final String FILE_NAME_5 = "data/statement-
   sample5.bl";
30     private static final String FILE_NAME_6 = "data/statement-
   sample6.bl";
31     private static final String FILE_NAME_7 = "data/statement-
   sample7.bl";
32
33     // TODO - define file names for additional test inputs
34
35     /**
36      * Invokes the {@code Statement} constructor for the
   implementation under
37      * test and returns the result.
38      *
39      * @return the new statement
40      * @ensures constructor = compose((BLOCK, ?, ?), <>)
41      */
42     protected abstract Statement constructorTest();
43
44     /**
```

```
45      * Invokes the {@code Statement} constructor for the reference
46      * implementation and returns the result.
47      *
48      * @return the new statement
49      * @ensures constructor = compose((BLOCK, ?, ?), <>)
50      */
51     protected abstract Statement constructorRef();
52
53     /**
54      *
55      * Creates and returns a block {@code Statement}, of the type of the
56      * implementation under test, from the file with the given name.
57      *
58      * @param filename
59      *            the name of the file to be parsed for the sequence of
60      *            statements to go in the block statement
61      * @return the constructed block statement
62      * @ensures <pre>
63      * createFromFile = [the block statement containing the statements
64      * parsed from the file]
65      * </pre>
66      */
67     private Statement createFromFileTest(String filename) {
68         Statement s = this.constructorTest();
69         SimpleReader file = new SimpleReader1L(filename);
70         Queue<String> tokens = Tokenizer.tokens(file);
71         s.parseBlock(tokens);
72         file.close();
73         return s;
74     }
75
76     /**
77      *
78      * Creates and returns a block {@code Statement}, of the reference
79      * implementation type, from the file with the given name.
80      *
81      * @param filename
82      *            the name of the file to be parsed for the
```

```
   sequence of
 83      *            statements to go in the block statement
 84      * @return the constructed block statement
 85      * @ensures <pre>
 86      * createFromFile = [the block statement containing the
   statements
 87      * parsed from the file]
 88      * </pre>
 89      */
 90     private Statement createFromFileRef(String filename) {
 91         Statement s = this.constructorRef();
 92         SimpleReader file = new SimpleReader1L(filename);
 93         Queue<String> tokens = Tokenizer.tokens(file);
 94         s.parseBlock(tokens);
 95         file.close();
 96         return s;
 97     }
 98
 99     /**
100      * Test constructor.
101      */
102     @Test
103     public final void testConstructor() {
104         /*
105          * Setup
106          */
107         Statement sRef = this.constructorRef();
108
109         /*
110          * The call
111          */
112         Statement sTest = this.constructorTest();
113
114         /*
115          * Evaluation
116          */
117         assertEquals(sRef, sTest);
118     }
119
120     /**
121      * Test kind of a WHILE statement.
122      */
123     @Test
124     public final void testKindWhile() {
```

```
125         /*
126          * Setup
127          */
128         final int whilePos = 3;
129         Statement sourceTest =
    this.createFromFileTest(FILE_NAME_1);
130         Statement sourceRef =
    this.createFromFileRef(FILE_NAME_1);
131         Statement sTest = sourceTest.removeFromBlock(whilePos);
132         Statement sRef = sourceRef.removeFromBlock(whilePos);
133         Kind kRef = sRef.kind();
134
135         /*
136          * The call
137          */
138         Kind kTest = sTest.kind();
139
140         /*
141          * Evaluation
142          */
143         assertEquals(kRef, kTest);
144         assertEquals(sRef, sTest);
145     }
146
147     /**
148      * Test kind of a IF statement.
149      */
150     @Test
151     public final void testKindIF() {
152         /*
153          * Setup
154          */
155         final int ifPos = 1;
156         Statement sourceTest =
    this.createFromFileTest(FILE_NAME_1);
157         Statement sourceRef =
    this.createFromFileRef(FILE_NAME_1);
158         Statement sTest = sourceTest.removeFromBlock(ifPos);
159         Statement sRef = sourceRef.removeFromBlock(ifPos);
160         Kind kRef = sRef.kind();
161
162         /*
163          * The call
164          */
```

```
165         Kind kTest = sTest.kind();
166 
167         /*
168          * Evaluation
169          */
170         assertEquals(kRef, kTest);
171         assertEquals(sRef, sTest);
172     }
173 
174     /**
175      * Test kind of a IFELSE statement.
176      */
177     @Test
178     public final void testKindIFELSE() {
179         /*
180          * Setup
181          */
182         final int ifElsePos = 2;
183         Statement sourceTest =
    this.createFromFileTest(FILE_NAME_1);
184         Statement sourceRef =
    this.createFromFileRef(FILE_NAME_1);
185         Statement sTest =
    sourceTest.removeFromBlock(ifElsePos);
186         Statement sRef = sourceRef.removeFromBlock(ifElsePos);
187         Kind kRef = sRef.kind();
188 
189         /*
190          * The call
191          */
192         Kind kTest = sTest.kind();
193 
194         /*
195          * Evaluation
196          */
197         assertEquals(kRef, kTest);
198         assertEquals(sRef, sTest);
199     }
200 
201     /**
202      * Test kind of a Call statement.
203      */
204     @Test
205     public final void testKindCall() {
```

```
206         /*
207          * Setup
208          */
209         final int callPos = 0;
210         Statement sourceTest =
    this.createFromFileTest(FILE_NAME_1);
211         Statement sourceRef =
    this.createFromFileRef(FILE_NAME_1);
212         Statement sTest = sourceTest.removeFromBlock(callPos);
213         Statement sRef = sourceRef.removeFromBlock(callPos);
214         Kind kRef = sRef.kind();
215 
216         /*
217          * The call
218          */
219         Kind kTest = sTest.kind();
220 
221         /*
222          * Evaluation
223          */
224         assertEquals(kRef, kTest);
225         assertEquals(sRef, sTest);
226     }
227 
228     /**
229      * Test kind of a Block statement.
230      */
231     @Test
232     public final void testKindBlock() {
233         /*
234          * Setup
235          */
236         final int blockPos = 1;
237         Statement sourceTest =
    this.createFromFileTest(FILE_NAME_1);
238         Statement sourceRef =
    this.createFromFileRef(FILE_NAME_1);
239         Statement sTest = sourceTest.removeFromBlock(blockPos);
240         Statement sRef = sourceRef.removeFromBlock(blockPos);
241         Kind kRef = sRef.kind();
242 
243         /*
244          * The call
245          */
```

```
246         Kind kTest = sTest.kind();
247
248         /*
249          * Evaluation
250          */
251         assertEquals(kRef, kTest);
252         assertEquals(sRef, sTest);
253     }
254
255     /**
256      * Test addToBlock at an interior position routine.
257      */
258     @Test
259     public final void testAddToBlockInterior1() {
260         /*
261          * Setup
262          */
263         Statement sTest = this.createFromFileTest(FILE_NAME_1);
264         Statement sRef = this.createFromFileRef(FILE_NAME_1);
265         Statement emptyBlock = sRef.newInstance();
266         Statement nestedTest = sTest.removeFromBlock(1);
267         Statement nestedRef = sRef.removeFromBlock(1);
268         sRef.addToBlock(2, nestedRef);
269
270         /*
271          * The call
272          */
273         sTest.addToBlock(2, nestedTest);
274
275         /*
276          * Evaluation
277          */
278         assertEquals(emptyBlock, nestedTest);
279         assertEquals(sRef, sTest);
280     }
281
282     /**
283      * Test addToBlock at an interior position edge.
284      */
285     @Test
286     public final void testAddToBlockInterior2() {
287         /*
288          * Setup
289          */
```

```
290         Statement sTest = this.createFromFileTest(FILE_NAME_2);
291         Statement sRef = this.createFromFileRef(FILE_NAME_2);
292         Statement emptyBlock = sRef.newInstance();
293         Statement nestedTest = sTest.removeFromBlock(0);
294         Statement nestedRef = sRef.removeFromBlock(0);
295         sRef.addToBlock(1, nestedRef);
296 
297         /*
298          * The call
299          */
300         sTest.addToBlock(1, nestedTest);
301 
302         /*
303          * Evaluation
304          */
305         assertEquals(emptyBlock, nestedTest);
306         assertEquals(sRef, sTest);
307     }
308 
309     /**
310      * Test addToBlock at an exterior position challenging.
311      */
312     @Test
313     public final void testAddToBlockExterior3() {
314         /*
315          * Setup
316          */
317         Statement sTest = this.createFromFileTest(FILE_NAME_2);
318         Statement sRef = this.createFromFileRef(FILE_NAME_2);
319         Statement emptyBlock = sRef.newInstance();
320         Statement nestedTest = sTest.removeFromBlock(1);
321         Statement nestedRef = sRef.removeFromBlock(1);
322         sRef.addToBlock(0, nestedRef);
323 
324         /*
325          * The call
326          */
327         sTest.addToBlock(0, nestedTest);
328 
329         /*
330          * Evaluation
331          */
332         assertEquals(emptyBlock, nestedTest);
333         assertEquals(sRef, sTest);
```

```
334     }
335
336     /**
337      * Test removeFromBlock at the front leaving a non-empty
    block behind.
338      */
339     @Test
340     public final void testRemoveFromBlockFrontLeavingNonEmpty()
    {
341         /*
342          * Setup
343          */
344         Statement sTest = this.createFromFileTest(FILE_NAME_1);
345         Statement sRef = this.createFromFileRef(FILE_NAME_1);
346         Statement nestedRef = sRef.removeFromBlock(0);
347
348         /*
349          * The call
350          */
351         Statement nestedTest = sTest.removeFromBlock(0);
352
353         /*
354          * Evaluation
355          */
356         assertEquals(sRef, sTest);
357         assertEquals(nestedRef, nestedTest);
358     }
359
360     /**
361      * Test removeFromBlock at the front leaving a empty block
    behind.
362      */
363     @Test
364     public final void testRemoveFromBlockFrontLeavingEmpty() {
365         /*
366          * Setup
367          */
368         Statement sTest = this.createFromFileTest(FILE_NAME_4);
369         Statement sRef = this.createFromFileRef(FILE_NAME_4);
370         Statement nestedRef = sRef.removeFromBlock(0);
371
372         /*
373          * The call
374          */
```

```
375         Statement nestedTest = sTest.removeFromBlock(0);
376 
377         /*
378          * Evaluation
379          */
380         assertEquals(sRef, sTest);
381         assertEquals(nestedRef, nestedTest);
382     }
383 
384     /**
385      * Test removeFromBlock at the back leaving a non-empty
    block behind.
386      */
387     @Test
388     public final void testRemoveFromBlockBackLeavingNonEmpty()
    {
389         /*
390          * Setup
391          */
392         Statement sTest = this.createFromFileTest(FILE_NAME_2);
393         Statement sRef = this.createFromFileRef(FILE_NAME_2);
394         Statement nestedRef = sRef.removeFromBlock(1);
395 
396         /*
397          * The call
398          */
399         Statement nestedTest = sTest.removeFromBlock(1);
400 
401         /*
402          * Evaluation
403          */
404         assertEquals(sRef, sTest);
405         assertEquals(nestedRef, nestedTest);
406     }
407 
408     /**
409      * Test lengthOfBlock, greater than zero.
410      */
411     @Test
412     public final void testLengthOfBlockNonEmpty() {
413         /*
414          * Setup
415          */
416         Statement sTest = this.createFromFileTest(FILE_NAME_1);
```

```
417         Statement sRef = this.createFromFileRef(FILE_NAME_1);
418         int lengthRef = sRef.lengthOfBlock();
419 
420         /*
421          * The call
422          */
423         int lengthTest = sTest.lengthOfBlock();
424 
425         /*
426          * Evaluation
427          */
428         assertEquals(lengthRef, lengthTest);
429         assertEquals(sRef, sTest);
430     }
431 
432     /**
433      * Test lengthOfBlock, with least amount of blocks.
434      */
435     @Test
436     public final void testLengthOfBlockZero() {
437         /*
438          * Setup
439          */
440         Statement sTest = this.createFromFileTest(FILE_NAME_3);
441         Statement sRef = this.createFromFileRef(FILE_NAME_3);
442         int lengthRef = sRef.lengthOfBlock();
443 
444         /*
445          * The call
446          */
447         int lengthTest = sTest.lengthOfBlock();
448 
449         /*
450          * Evaluation
451          */
452         assertEquals(lengthRef, lengthTest);
453         assertEquals(sRef, sTest);
454     }
455 
456     /**
457      * Test assembleIf.
458      */
459     @Test
460     public final void testAssembleIf() {
```

```
461         /*
462          * Setup
463          */
464         Statement blockTest =
    this.createFromFileTest(FILE_NAME_1);
465         Statement blockRef =
    this.createFromFileRef(FILE_NAME_1);
466         Statement emptyBlock = blockRef.newInstance();
467         Statement sourceTest = blockTest.removeFromBlock(1);
468         Statement sRef = blockRef.removeFromBlock(1);
469         Statement nestedTest = sourceTest.newInstance();
470         Condition c = sourceTest.disassembleIf(nestedTest);
471         Statement sTest = sourceTest.newInstance();
472
473         /*
474          * The call
475          */
476         sTest.assembleIf(c, nestedTest);
477
478         /*
479          * Evaluation
480          */
481         assertEquals(emptyBlock, nestedTest);
482         assertEquals(sRef, sTest);
483     }
484
485     /**
486      * Test assembleIf with boolean.
487      */
488     @Test
489     public final void testAssembleIf2() {
490         /*
491          * Setup
492          */
493         Statement blockTest =
    this.createFromFileTest(FILE_NAME_3);
494         Statement blockRef =
    this.createFromFileRef(FILE_NAME_3);
495         Statement emptyBlock = blockRef.newInstance();
496         Statement sourceTest = blockTest.removeFromBlock(0);
497         Statement sRef = blockRef.removeFromBlock(0);
498         Statement nestedTest = sourceTest.newInstance();
499         Condition c = sourceTest.disassembleIf(nestedTest);
500         Statement sTest = sourceTest.newInstance();
```

```
501
502         /*
503          * The call
504          */
505         sTest.assembleIf(c, nestedTest);
506
507         /*
508          * Evaluation
509          */
510         assertEquals(emptyBlock, nestedTest);
511         assertEquals(sRef, sTest);
512     }
513
514     /**
515      * Test disassembleIf.
516      */
517     @Test
518     public final void testDisassembleIf() {
519         /*
520          * Setup
521          */
522         Statement blockTest =
    this.createFromFileTest(FILE_NAME_1);
523         Statement blockRef =
    this.createFromFileRef(FILE_NAME_1);
524         Statement sTest = blockTest.removeFromBlock(1);
525         Statement sRef = blockRef.removeFromBlock(1);
526         Statement nestedTest = sTest.newInstance();
527         Statement nestedRef = sRef.newInstance();
528         Condition cRef = sRef.disassembleIf(nestedRef);
529
530         /*
531          * The call
532          */
533         Condition cTest = sTest.disassembleIf(nestedTest);
534
535         /*
536          * Evaluation
537          */
538         assertEquals(nestedRef, nestedTest);
539         assertEquals(sRef, sTest);
540         assertEquals(cRef, cTest);
541     }
542
```

```
543     /**
544      * Test disassembleIf with boolean.
545      */
546     @Test
547     public final void testDisassembleIf2() {
548         /*
549          * Setup
550          */
551         Statement blockTest =
    this.createFromFileTest(FILE_NAME_4);
552         Statement blockRef =
    this.createFromFileRef(FILE_NAME_4);
553         Statement sTest = blockTest.removeFromBlock(0);
554         Statement sRef = blockRef.removeFromBlock(0);
555         Statement nestedTest = sTest.newInstance();
556         Statement nestedRef = sRef.newInstance();
557         Condition cRef = sRef.disassembleIf(nestedRef);
558
559         /*
560          * The call
561          */
562         Condition cTest = sTest.disassembleIf(nestedTest);
563
564         /*
565          * Evaluation
566          */
567         assertEquals(nestedRef, nestedTest);
568         assertEquals(sRef, sTest);
569         assertEquals(cRef, cTest);
570     }
571
572     /**
573      * Test assembleIfElse.
574      */
575     @Test
576     public final void testAssembleIfElse() {
577         /*
578          * Setup
579          */
580         final int ifElsePos = 2;
581         Statement blockTest =
    this.createFromFileTest(FILE_NAME_1);
582         Statement blockRef =
    this.createFromFileRef(FILE_NAME_1);
```

```
583         Statement emptyBlock = blockRef.newInstance();
584         Statement sourceTest =
    blockTest.removeFromBlock(ifElsePos);
585         Statement sRef = blockRef.removeFromBlock(ifElsePos);
586         Statement thenBlockTest = sourceTest.newInstance();
587         Statement elseBlockTest = sourceTest.newInstance();
588         Condition cTest =
    sourceTest.disassembleIfElse(thenBlockTest,
589                 elseBlockTest);
590         Statement sTest = blockTest.newInstance();
591
592         /*
593          * The call
594          */
595         sTest.assembleIfElse(cTest, thenBlockTest,
    elseBlockTest);
596
597         /*
598          * Evaluation
599          */
600         assertEquals(emptyBlock, thenBlockTest);
601         assertEquals(emptyBlock, elseBlockTest);
602         assertEquals(sRef, sTest);
603     }
604
605     /**
606      * Test assembleIfElse with boolean.
607      */
608     @Test
609     public final void testAssembleIfElse2() {
610         /*
611          * Setup
612          */
613         final int ifElsePos = 0;
614         Statement blockTest =
    this.createFromFileTest(FILE_NAME_5);
615         Statement blockRef =
    this.createFromFileRef(FILE_NAME_5);
616         Statement emptyBlock = blockRef.newInstance();
617         Statement sourceTest =
    blockTest.removeFromBlock(ifElsePos);
618         Statement sRef = blockRef.removeFromBlock(ifElsePos);
619         Statement thenBlockTest = sourceTest.newInstance();
620         Statement elseBlockTest = sourceTest.newInstance();
```

```
621         Condition cTest =
    sourceTest.disassembleIfElse(thenBlockTest,
622                 elseBlockTest);
623         Statement sTest = blockTest.newInstance();
624
625         /*
626          * The call
627          */
628         sTest.assembleIfElse(cTest, thenBlockTest,
    elseBlockTest);
629
630         /*
631          * Evaluation
632          */
633         assertEquals(emptyBlock, thenBlockTest);
634         assertEquals(emptyBlock, elseBlockTest);
635         assertEquals(sRef, sTest);
636     }
637
638     /**
639      * Test disassembleIfElse.
640      */
641     @Test
642     public final void testDisassembleIfElse() {
643         /*
644          * Setup
645          */
646         final int ifElsePos = 2;
647         Statement blockTest =
    this.createFromFileTest(FILE_NAME_1);
648         Statement blockRef =
    this.createFromFileRef(FILE_NAME_1);
649         Statement sTest = blockTest.removeFromBlock(ifElsePos);
650         Statement sRef = blockRef.removeFromBlock(ifElsePos);
651         Statement thenBlockTest = sTest.newInstance();
652         Statement elseBlockTest = sTest.newInstance();
653         Statement thenBlockRef = sRef.newInstance();
654         Statement elseBlockRef = sRef.newInstance();
655         Condition cRef = sRef.disassembleIfElse(thenBlockRef,
    elseBlockRef);
656
657         /*
658          * The call
659          */
```

```
660         Condition cTest =
    sTest.disassembleIfElse(thenBlockTest, elseBlockTest);
661
662         /*
663          * Evaluation
664          */
665         assertEquals(cRef, cTest);
666         assertEquals(thenBlockRef, thenBlockTest);
667         assertEquals(elseBlockRef, elseBlockTest);
668         assertEquals(sRef, sTest);
669     }
670
671     /**
672      * Test disassembleIfElse with boolean.
673      */
674     @Test
675     public final void testDisassembleIfElse2() {
676         /*
677          * Setup
678          */
679         final int ifElsePos = 0;
680         Statement blockTest =
    this.createFromFileTest(FILE_NAME_5);
681         Statement blockRef =
    this.createFromFileRef(FILE_NAME_5);
682         Statement sTest = blockTest.removeFromBlock(ifElsePos);
683         Statement sRef = blockRef.removeFromBlock(ifElsePos);
684         Statement thenBlockTest = sTest.newInstance();
685         Statement elseBlockTest = sTest.newInstance();
686         Statement thenBlockRef = sRef.newInstance();
687         Statement elseBlockRef = sRef.newInstance();
688         Condition cRef = sRef.disassembleIfElse(thenBlockRef,
    elseBlockRef);
689
690         /*
691          * The call
692          */
693         Condition cTest =
    sTest.disassembleIfElse(thenBlockTest, elseBlockTest);
694
695         /*
696          * Evaluation
697          */
698         assertEquals(cRef, cTest);
```

```
699         assertEquals(thenBlockRef, thenBlockTest);
700         assertEquals(elseBlockRef, elseBlockTest);
701         assertEquals(sRef, sTest);
702     }
703 
704     /**
705      * Test assembleWhile.
706      */
707     @Test
708     public final void testAssembleWhile() {
709         /*
710          * Setup
711          */
712         Statement blockTest =
    this.createFromFileTest(FILE_NAME_1);
713         Statement blockRef =
    this.createFromFileRef(FILE_NAME_1);
714         Statement emptyBlock = blockRef.newInstance();
715         Statement sourceTest = blockTest.removeFromBlock(1);
716         Statement sourceRef = blockRef.removeFromBlock(1);
717         Statement nestedTest = sourceTest.newInstance();
718         Statement nestedRef = sourceRef.newInstance();
719         Condition cTest = sourceTest.disassembleIf(nestedTest);
720         Condition cRef = sourceRef.disassembleIf(nestedRef);
721         Statement sRef = sourceRef.newInstance();
722         sRef.assembleWhile(cRef, nestedRef);
723         Statement sTest = sourceTest.newInstance();
724 
725         /*
726          * The call
727          */
728         sTest.assembleWhile(cTest, nestedTest);
729 
730         /*
731          * Evaluation
732          */
733         assertEquals(emptyBlock, nestedTest);
734         assertEquals(sRef, sTest);
735     }
736 
737     /**
738      * Test assembleWhile with boolean.
739      */
740     @Test
```

```
741     public final void testAssembleWhile2() {
742         /*
743          * Setup
744          */
745         Statement blockTest =
    this.createFromFileTest(FILE_NAME_4);
746         Statement blockRef =
    this.createFromFileRef(FILE_NAME_4);
747         Statement emptyBlock = blockRef.newInstance();
748         Statement sourceTest = blockTest.removeFromBlock(0);
749         Statement sourceRef = blockRef.removeFromBlock(0);
750         Statement nestedTest = sourceTest.newInstance();
751         Statement nestedRef = sourceRef.newInstance();
752         Condition cTest = sourceTest.disassembleIf(nestedTest);
753         Condition cRef = sourceRef.disassembleIf(nestedRef);
754         Statement sRef = sourceRef.newInstance();
755         sRef.assembleWhile(cRef, nestedRef);
756         Statement sTest = sourceTest.newInstance();
757
758         /*
759          * The call
760          */
761         sTest.assembleWhile(cTest, nestedTest);
762
763         /*
764          * Evaluation
765          */
766         assertEquals(emptyBlock, nestedTest);
767         assertEquals(sRef, sTest);
768     }
769
770     /**
771      * Test disassembleWhile.
772      */
773     @Test
774     public final void testDisassembleWhile() {
775         /*
776          * Setup
777          */
778         final int whilePos = 3;
779         Statement blockTest =
    this.createFromFileTest(FILE_NAME_1);
780         Statement blockRef =
    this.createFromFileRef(FILE_NAME_1);
```

```
781         Statement sTest = blockTest.removeFromBlock(whilePos);
782         Statement sRef = blockRef.removeFromBlock(whilePos);
783         Statement nestedTest = sTest.newInstance();
784         Statement nestedRef = sRef.newInstance();
785         Condition cRef = sRef.disassembleWhile(nestedRef);
786 
787         /*
788          * The call
789          */
790         Condition cTest = sTest.disassembleWhile(nestedTest);
791 
792         /*
793          * Evaluation
794          */
795         assertEquals(nestedRef, nestedTest);
796         assertEquals(sRef, sTest);
797         assertEquals(cRef, cTest);
798     }
799 
800     /**
801      * Test disassembleWhile with boolean.
802      */
803     @Test
804     public final void testDisassembleWhile2() {
805         /*
806          * Setup
807          */
808         final int whilePos = 0;
809         Statement blockTest =
    this.createFromFileTest(FILE_NAME_6);
810         Statement blockRef =
    this.createFromFileRef(FILE_NAME_6);
811         Statement sTest = blockTest.removeFromBlock(whilePos);
812         Statement sRef = blockRef.removeFromBlock(whilePos);
813         Statement nestedTest = sTest.newInstance();
814         Statement nestedRef = sRef.newInstance();
815         Condition cRef = sRef.disassembleWhile(nestedRef);
816 
817         /*
818          * The call
819          */
820         Condition cTest = sTest.disassembleWhile(nestedTest);
821 
822         /*
```

```
823          * Evaluation
824          */
825         assertEquals(nestedRef, nestedTest);
826         assertEquals(sRef, sTest);
827         assertEquals(cRef, cTest);
828     }
829 
830     /**
831      * Test assembleCall.
832      */
833     @Test
834     public final void testAssembleCall() {
835         /*
836          * Setup
837          */
838         Statement sRef = this.constructorRef().newInstance();
839         Statement sTest = this.constructorTest().newInstance();
840 
841         String name = "look-for-something";
842         sRef.assembleCall(name);
843 
844         /*
845          * The call
846          */
847         sTest.assembleCall(name);
848 
849         /*
850          * Evaluation
851          */
852         assertEquals(sRef, sTest);
853     }
854 
855     /**
856      * Test assembleCall with a dash(challenging).
857      */
858     @Test
859     public final void testAssembleCall2() {
860         /*
861          * Setup
862          */
863         Statement sRef = this.constructorRef().newInstance();
864         Statement sTest = this.constructorTest().newInstance();
865 
866         String name = "ONE-";
```

```
867         sRef.assembleCall(name);
868 
869         /*
870          * The call
871          */
872         sTest.assembleCall(name);
873 
874         /*
875          * Evaluation
876          */
877         assertEquals(sRef, sTest);
878     }
879 
880     /**
881      * Test disassembleCall.
882      */
883     @Test
884     public final void testDisassembleCall() {
885         /*
886          * Setup
887          */
888         Statement blockTest =
    this.createFromFileTest(FILE_NAME_1);
889         Statement blockRef =
    this.createFromFileRef(FILE_NAME_1);
890         Statement sTest = blockTest.removeFromBlock(0);
891         Statement sRef = blockRef.removeFromBlock(0);
892         String nRef = sRef.disassembleCall();
893 
894         /*
895          * The call
896          */
897         String nTest = sTest.disassembleCall();
898         System.out.println(nTest);
899 
900         /*
901          * Evaluation
902          */
903         assertEquals(sRef, sTest);
904         assertEquals(nRef, nTest);
905     }
906 
907     /**
908      * Test disassembleCall with one line.
```

```
909      */
910     @Test
911     public final void testDisassembleCall2() {
912         /*
913          * Setup
914          */
915         Statement blockTest =
    this.createFromFileTest(FILE_NAME_7);
916         Statement blockRef =
    this.createFromFileRef(FILE_NAME_7);
917         Statement sTest = blockTest.removeFromBlock(0);
918         Statement sRef = blockRef.removeFromBlock(0);
919         String nRef = sRef.disassembleCall();
920 
921         /*
922          * The call
923          */
924         String nTest = sTest.disassembleCall();
925 
926         /*
927          * Evaluation
928          */
929         assertEquals(sRef, sTest);
930         assertEquals(nRef, nTest);
931     }
932 
933     // TODO - provide additional test cases to thoroughly test
    StatementKernel
934 
935 }
936 
```