```java
import components.queue.Queue;

/**
 * Layered implementation of secondary methods {@code parse} and
 * {@code parseBlock} for {@code Statement}.
 *
 * @author Put your name here
 *
 */
public final class Statement1Parse1 extends Statement1 {

    /*
     * Private members --------------------------------------------------------
     */

    /**
     * Converts {@code c} into the corresponding {@code Condition}.
     *
     * @param c
     *            the condition to convert
     * @return the {@code Condition} corresponding to {@code c}
     * @requires [c is a condition string]
     * @ensures parseCondition = [Condition corresponding to c]
     */
    private static Condition parseCondition(String c) {
        assert c != null : "Violation of: c is not null";
        assert Tokenizer.isCondition(c) : "Violation of: c is a condition string";
        return Condition.valueOf(c.replace('-', '_').toUpperCase());
    }

    /**
     * Parses an IF or IF_ELSE statement from {@code tokens} into {@code s}.
     *
     * @param tokens
     *            the input tokens
     * @param s
     *            the parsed statement
     * @replaces s
     * @updates tokens
     * @requires <pre>
     * [<"IF"> is a prefix of tokens]  and
     *  [<Tokenizer.END_OF_INPUT> is a suffix of tokens]
     * </pre>
     * @ensures <pre>
     * if [an if string is a proper prefix of #tokens] then
     *  s = [IF or IF_ELSE Statement corresponding to if string at start of #tokens]
and
     *  #tokens = [if string at start of #tokens] * tokens
     * else
     *  [reports an appropriate error message to the console and terminates client]
     * </pre>
     */
    private static void parseIf(Queue<String> tokens, Statement s) {
        assert tokens != null : "Violation of: tokens is not null";
        assert s != null : "Violation of: s is not null";
        assert tokens.length() > 0 && tokens.front().equals("IF")
                : "" + "Violation of: <\"IF\"> is proper prefix of tokens";
```

```
 65
 66          // if and else statement variables
 67          Statement ifBlock = s.newInstance();
 68          Statement elseBlock = s.newInstance();
 69
 70          // check for if
 71          Reporter.assertElseFatalError(tokens.dequeue().equals("IF"), "Incorrect
   syntax.");
 72
 73          // parse condition
 74          String cString = tokens.dequeue();
 75          Reporter.assertElseFatalError(Tokenizer.isCondition(cString),
 76                  "No condition found.");
 77          Condition c = parseCondition(cString);
 78
 79          // check for then
 80          Reporter.assertElseFatalError(tokens.dequeue().equals("THEN"),
 81                  "THEN token not found.");
 82
 83          // parse block
 84          ifBlock.parseBlock(tokens);
 85
 86          // check for else
 87          if (tokens.front().equals("ELSE")) {
 88              // parse else
 89              tokens.dequeue();
 90              elseBlock.parseBlock(tokens);
 91
 92              // assemble if else
 93              s.assembleIfElse(c, ifBlock, elseBlock);
 94          } else {
 95              // assemble if
 96              s.assembleIf(c, ifBlock);
 97          }
 98
 99          // check for end
100          Reporter.assertElseFatalError(tokens.dequeue().equals("END"),
101                  "END token not found.");
102
103          // check for IF
104          Reporter.assertElseFatalError(tokens.dequeue().equals("IF"),
105                  "Closing IF token not found.");
106
107      }
108
109      /**
110       * Parses a WHILE statement from {@code tokens} into {@code s}.
111       *
112       * @param tokens
113       *            the input tokens
114       * @param s
115       *            the parsed statement
116       * @replaces s
117       * @updates tokens
118       * @requires <pre>
119       * [<"WHILE"> is a prefix of tokens]  and
120       *  [<Tokenizer.END_OF_INPUT> is a suffix of tokens]
```

```
121      * </pre>
122      * @ensures <pre>
123      * if [a while string is a proper prefix of #tokens] then
124      *  s = [WHILE Statement corresponding to while string at start of #tokens]  and
125      *  #tokens = [while string at start of #tokens] * tokens
126      * else
127      *  [reports an appropriate error message to the console and terminates client]
128      * </pre>
129      */
130     private static void parseWhile(Queue<String> tokens, Statement s) {
131         assert tokens != null : "Violation of: tokens is not null";
132         assert s != null : "Violation of: s is not null";
133         assert tokens.length() > 0 && tokens.front().equals("WHILE")
134                 : "" + "Violation of: <\"WHILE\"> is proper prefix of tokens";
135
136         // new statement
137         Statement whileBlock = s.newInstance();
138
139         // check for if
140         Reporter.assertElseFatalError(tokens.dequeue().equals("WHILE"),
141                 "WHILE token not found.");
142
143         // parse condition
144         String cString = tokens.dequeue();
145         Reporter.assertElseFatalError(Tokenizer.isCondition(cString),
146                 "No condition found.");
147         Condition c = parseCondition(cString);
148
149         // check for then
150         Reporter.assertElseFatalError(tokens.dequeue().equals("DO"), "Incorrect
    syntax.");
151
152         // parse block
153         whileBlock.parseBlock(tokens);
154
155         // check for end
156         Reporter.assertElseFatalError(tokens.dequeue().equals("END"),
157                 "END token not found.");
158
159         // check for while
160         Reporter.assertElseFatalError(tokens.dequeue().equals("WHILE"),
161                 "Cloding WHILE token not found.");
162
163         // assemble while
164         s.assembleWhile(c, whileBlock);
165
166     }
167
168     /**
169      * Parses a CALL statement from {@code tokens} into {@code s}.
170      *
171      * @param tokens
172      *            the input tokens
173      * @param s
174      *            the parsed statement
175      * @replaces s
176      * @updates tokens
```

```java
     * @requires [identifier string is a proper prefix of tokens]
     * @ensures <pre>
     * s =
     *   [CALL Statement corresponding to identifier string at start of #tokens]  and
     *  #tokens = [identifier string at start of #tokens] * tokens
     * </pre>
     */
    private static void parseCall(Queue<String> tokens, Statement s) {
        assert tokens != null : "Violation of: tokens is not null";
        assert s != null : "Violation of: s is not null";
        assert tokens.length() > 0 && Tokenizer.isIdentifier(tokens.front())
                : "" + "Violation of: identifier string is proper prefix of tokens";

        // get call
        String call = tokens.dequeue();
        Reporter.assertElseFatalError(Tokenizer.isIdentifier(call), "Invalid call.");

        // assemble call
        s.assembleCall(call);

    }

    /*
     * Constructors ------------------------------------------------------------
     */

    /**
     * No-argument constructor.
     */
    public Statement1Parse1() {
        super();
    }

    /*
     * Public methods ----------------------------------------------------------
     */

    @Override
    public void parse(Queue<String> tokens) {
        assert tokens != null : "Violation of: tokens is not null";
        assert tokens.length() > 0
                : "" + "Violation of: Tokenizer.END_OF_INPUT is a suffix of tokens";

        if (tokens.front().equals("WHILE")) {
            parseWhile(tokens, this);
        } else if (tokens.front().equals("IF")) {
            parseIf(tokens, this);
        } else {
            // must be identifier if none of the above, considering body for
parseBlock
            parseCall(tokens, this);
        }

    }

    @Override
    public void parseBlock(Queue<String> tokens) {
```

```
233         assert tokens != null : "Violation of: tokens is not null";
234         assert tokens.length() > 0
235                 : "" + "Violation of: Tokenizer.END_OF_INPUT is a suffix of tokens";
236
237         Statement tempBlock = this.newInstance();
238
239         while (tokens.front().equals("WHILE") || tokens.front().equals("IF")
240                 || Tokenizer.isIdentifier(tokens.front())) {
241             Statement tempStatement = this.newInstance();
242
243             // parse statement
244             tempStatement.parse(tokens);
245
246             // add to block
247             tempBlock.addToBlock(tempBlock.lengthOfBlock(), tempStatement);
248         }
249
250         this.transferFrom(tempBlock);
251
252     }
253
254     /*
255      * Main test method ----------------------------------------------------
256      */
257
258     /**
259      * Main method.
260      *
261      * @param args
262      *            the command line arguments
263      */
264     public static void main(String[] args) {
265         SimpleReader in = new SimpleReader1L();
266         SimpleWriter out = new SimpleWriter1L();
267         /*
268          * Get input file name
269          */
270         out.print("Enter valid BL statement(s) file name: ");
271         String fileName = in.nextLine();
272         /*
273          * Parse input file
274          */
275         out.println("*** Parsing input file ***");
276         Statement s = new Statement1Parse1();
277         SimpleReader file = new SimpleReader1L(fileName);
278         Queue<String> tokens = Tokenizer.tokens(file);
279         file.close();
280         s.parse(tokens); // replace with parseBlock to test other method
281         /*
282          * Pretty print the statement(s)
283          */
284         out.println("*** Pretty print of parsed statement(s) ***");
285         s.prettyPrint(out, 0);
286
287         in.close();
288         out.close();
289     }
```

```
290
291 }
292
```