```java
import components.queue.Queue;

/**
 * Layered implementation of secondary methods {@code parse} and
 * {@code parseBlock} for {@code Statement}.
 *
 * @author Gabe Azzarita and Ty Fredrick
 *
 */
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";

```

```
67          // discard "IF", syntax is already checked in assertions
68          tokens.dequeue();
69
70          // check for proper condition
71          String conditionString = tokens.dequeue();
72          Reporter.assertElseFatalError(Tokenizer.isCondition(conditionString),
73                  conditionString + ": Invalid condition.");
74          Condition ifCondition = parseCondition(conditionString);
75
76          // check THEN syntax
77          String thenStr = tokens.dequeue();
78          Reporter.assertElseFatalError(thenStr.equals("THEN"),
79                  "Recieved: " + thenStr + ", Expected: THEN.");
80
81          // create and parse ifBlock
82          Statement ifBlock = s.newInstance();
83          ifBlock.parseBlock(tokens);
84
85          /*
86           * parseBlock will parse ifBlock until it hits "END" or "ELSE" or
87           * "### END OF INPUT ###", so check if its ELSE to determine whether we
88           * have IF or IF_ELSE
89           */
90
91          if (tokens.front().equals("ELSE")) {
92
93              // discard "ELSE" then parse else block
94              tokens.dequeue();
95              Statement eBlock = s.newInstance();
96              eBlock.parseBlock(tokens);
97
98              s.assembleIfElse(ifCondition, ifBlock, eBlock);
99
100         } else {
101
102             s.assembleIf(ifCondition, ifBlock);
103
104         }
105
106         // check END and IF syntax
107         String end = tokens.dequeue();
108         Reporter.assertElseFatalError(end.equals("END"),
109                 "Recieved: " + end + ", Expected: END.");
110
111         String ifStr = tokens.dequeue();
112         Reporter.assertElseFatalError(ifStr.equals("IF"),
113                 "Recieved: " + ifStr + ", Expected: IF.");
114
115     }
116
117     /**
118      * Parses a WHILE statement from {@code tokens} into {@code s}.
119      *
120      * @param tokens
121      *            the input tokens
122      * @param s
123      *            the parsed statement
124      * @replaces s
125      * @updates tokens
```

```
126      * @requires <pre>
127      * [<"WHILE"> is a prefix of tokens]  and
128      *  [<Tokenizer.END_OF_INPUT> is a suffix of tokens]
129      * </pre>
130      * @ensures <pre>
131      * if [a while string is a proper prefix of #tokens] then
132      *  s = [WHILE Statement corresponding to while string at start of #tokens]  and
133      *  #tokens = [while string at start of #tokens] * tokens
134      * else
135      *  [reports an appropriate error message to the console and terminates client]
136      * </pre>
137      */
138     private static void parseWhile(Queue<String> tokens, Statement s) {
139         assert tokens != null : "Violation of: tokens is not null";
140         assert s != null : "Violation of: s is not null";
141         assert tokens.length() > 0 && tokens.front().equals("WHILE") : ""
142                 + "Violation of: <\"WHILE\"> is proper prefix of tokens";
143
144         // discard "WHILE", syntax checked in assertion
145         tokens.dequeue();
146
147         // check for proper while condition
148         String conditionString = tokens.dequeue();
149         Reporter.assertElseFatalError(Tokenizer.isCondition(conditionString),
150                 conditionString + ": Invalid condition.");
151         Condition whileCondition = parseCondition(conditionString);
152
153         // check DO syntax
154         String doStr = tokens.dequeue();
155         Reporter.assertElseFatalError(doStr.equals("DO"),
156                 "Recieved: " + doStr + ", Expected: DO.");
157
158         // create and parse whileBlock
159         Statement whileBlock = s.newInstance();
160         whileBlock.parseBlock(tokens);
161
162         // check END and WHILE syntax
163         String end = tokens.dequeue();
164         Reporter.assertElseFatalError(end.equals("END"),
165                 "Recieved: " + end + ", Expected: END.");
166
167         String whileStr = tokens.dequeue();
168         Reporter.assertElseFatalError(whileStr.equals("WHILE"),
169                 "Recieved: " + whileStr + ", Expected: WHILE.");
170
171         // assemble after checking closing syntax
172         s.assembleWhile(whileCondition, whileBlock);
173
174     }
175
176     /**
177      * Parses a CALL statement from {@code tokens} into {@code s}.
178      *
179      * @param tokens
180      *            the input tokens
181      * @param s
182      *            the parsed statement
183      * @replaces s
184      * @updates tokens
```

```
185      * @requires [identifier string is a proper prefix of tokens]
186      * @ensures <pre>
187      * s =
188      *   [CALL Statement corresponding to identifier string at start of #tokens]  and
189      *  #tokens = [identifier string at start of #tokens] * tokens
190      * </pre>
191      */
192     private static void parseCall(Queue<String> tokens, Statement s) {
193         assert tokens != null : "Violation of: tokens is not null";
194         assert s != null : "Violation of: s is not null";
195         assert tokens.length() > 0
196                 && Tokenizer.isIdentifier(tokens.front()) : ""
197                         + "Violation of: identifier string is proper prefix of
    tokens";
198
199         // no need to check syntax, already checked in assertion
200         String call = tokens.dequeue();
201         s.assembleCall(call);
202
203     }
204
205     /*
206      * Constructors ------------------------------------------------------------
207      */
208
209     /**
210      * No-argument constructor.
211      */
212     public Statement1Parse1() {
213         super();
214     }
215
216     /*
217      * Public methods ----------------------------------------------------------
218      */
219
220     @Override
221     public void parse(Queue<String> tokens) {
222         assert tokens != null : "Violation of: tokens is not null";
223         assert tokens.length() > 0 : ""
224                 + "Violation of: Tokenizer.END_OF_INPUT is a suffix of tokens";
225
226         String type = tokens.front();
227
228         // front token can either be WHILE, IF, or a simple call
229         if (type.equals("IF")) {
230             parseIf(tokens, this);
231         } else if (type.equals("WHILE")) {
232             parseWhile(tokens, this);
233         } else {
234             parseCall(tokens, this);
235         }
236
237     }
238
239     @Override
240     public void parseBlock(Queue<String> tokens) {
241         assert tokens != null : "Violation of: tokens is not null";
242         assert tokens.length() > 0 : ""
```

```
243                 + "Violation of: Tokenizer.END_OF_INPUT is a suffix of tokens";
244
245         Statement st = this.newInstance();
246
247         /*
248          * parse tokens until end of block or until front = "END", "ELSE", or
249          * "## END OF INPUT ###"
250          */
251
252         for (int i = 0; !tokens.front().equals("END")
253                 && !tokens.front().equals("ELSE")
254                 && !tokens.front().equals("### END OF INPUT ###"); i++) {
255
256             // parse tokens then add statement to block
257             st.parse(tokens);
258             this.addToBlock(i, st);
259         }
260     }
261
262     /*
263      * Main test method ----------------------------------------------------
264      */
265
266     /**
267      * Main method.
268      *
269      * @param args
270      *            the command line arguments
271      */
272     public static void main(String[] args) {
273         SimpleReader in = new SimpleReader1L();
274         SimpleWriter out = new SimpleWriter1L();
275         /*
276          * Get input file name
277          */
278         out.print("Enter valid BL statement(s) file name: ");
279         String fileName = in.nextLine();
280         /*
281          * Parse input file
282          */
283         out.println("*** Parsing input file ***");
284         Statement s = new Statement1Parse1();
285         SimpleReader file = new SimpleReader1L(fileName);
286         Queue<String> tokens = Tokenizer.tokens(file);
287         file.close();
288         s.parse(tokens); // replace with parseBlock to test other method
289         /*
290          * Pretty print the statement(s)
291          */
292         out.println("*** Pretty print of parsed statement(s) ***");
293         s.prettyPrint(out, 0);
294
295         in.close();
296         out.close();
297     }
298
299 }
300
```