```java
import components.queue.Queue;

/**
 * Layered implementation of secondary methods {@code parse} and
 * {@code parseBlock} for {@code Statement}.
 *
 * @author Qinuo Shi & Yiming Cheng
 *
 */
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";

        // TODO - fill in body
        tokens.dequeue();

        /*
         * If the BL format is not found, report an error here
         */
        String con = tokens.dequeue();
        Reporter.assertElseFatalError(Tokenizer.isCondition(con),
                "Cannot find CONDITION.");

        Condition ifCon = parseCondition(con);

        Reporter.assertElseFatalError(tokens.dequeue().equals("THEN"),
                "Cannot find THEN.");

        Statement tool = s.newInstance();
        tool.transferFrom(s);

        /*
         * Parse IF without either END or ELSE
         */
        while (!tokens.front().equals("END")
                && !tokens.front().equals("ELSE")) {
            tool.parseBlock(tokens);
        }

        /*
         * Parse IF with ELSE
         */
        if (tokens.front().equals("ELSE")) {
            tokens.dequeue();
            Statement elseBlock = s.newInstance();

            while (!tokens.front().equals("END")) {
                elseBlock.parseBlock(tokens);
            }
            /*
             * Assemble IF and ELSE
             */
            s.assembleIfElse(ifCon, tool, elseBlock);
        } else {
            /*
             * Assemble IF
             */
            s.assembleIf(ifCon, tool);
        }

        tokens.dequeue();

        /*
         * Reporting an error if IF cannot be found.
         */
        String testForIF = tokens.dequeue();
        Reporter.assertElseFatalError(testForIF.equals("IF"), "Cannot find IF");

    }

    /**
     * Parses a WHILE statement from {@code tokens} into {@code s}.
     *
     * @param tokens
     *            the input tokens
     * @param s
     *            the parsed statement
     * @replaces s
     * @updates tokens
     * @requires <pre>
     * [<"WHILE"> is a prefix of tokens]  and
     *  [<Tokenizer.END_OF_INPUT> is a suffix of tokens]
     * </pre>
     * @ensures <pre>
     * if [a while string is a proper prefix of #tokens] then
     *  s = [WHILE Statement corresponding to while string at start of #tokens]  and
     *  #tokens = [while string at start of #tokens] * tokens
     * else
     *  [reports an appropriate error message to the console and terminates client]
     * </pre>
     */
    private static void parseWhile(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("WHILE") : ""
                + "Violation of: <\"WHILE\"> is proper prefix of tokens";

        // TODO - fill in body
        tokens.dequeue();

        /*
         * If the BL format is not found, report an error here
         */
        String con = tokens.dequeue();
        Reporter.assertElseFatalError(Tokenizer.isCondition(con),
                "Cannot find CONDITION.");

        Condition whileCon = parseCondition(con);

        Reporter.assertElseFatalError(tokens.dequeue().equals("DO"),
                "Cannot find DO");

        /*
         * Parse WHILE
         */
        s.parseBlock(tokens);

        Statement tool = s.newInstance();
        tool.transferFrom(s);
        /*
         * Assemble WHILE
         */
        s.assembleWhile(whileCon, tool);

        /*
         * Report an error if END and WHILE cannot be found
         */
        Reporter.assertElseFatalError(tokens.dequeue().equals("END"),
                "Cannot find END");
        Reporter.assertElseFatalError(tokens.dequeue().equals("WHILE"),
                "Cannot find WHILE");
    }

    /**
     * Parses a CALL statement from {@code tokens} into {@code s}.
     *
     * @param tokens
     *            the input tokens
     * @param s
     *            the parsed statement
     * @replaces s
     * @updates tokens
     * @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";

        // TODO - fill in body
        s.assembleCall(tokens.dequeue());

    }

    /*
     * 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";

        // TODO - fill in body
```

```java
        /*
         * Run them in different method by identifying their kind
         */
        if (tokens.front().equals("IF")) {
            parseIf(tokens, this);
        } else if (tokens.front().equals("WHILE")) {
            parseWhile(tokens, this);
        } else {
            parseCall(tokens, this);
        }

    }

    @Override
    public void parseBlock(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";

        // TODO - fill in body
        /*
         * Block as a special condition, running here
         */
        while ((!tokens.front().equals("ELSE") && !tokens.front().equals("END")
                && !tokens.front().equals(Tokenizer.END_OF_INPUT))) {
            Statement tool = this.newInstance();
            tool.parse(tokens);
            this.addToBlock(this.lengthOfBlock(), tool);
            tool.clear();
        }

    }

    /*
     * Main test method -------------------------------------------------------
     */

    /**
     * Main method.
     *
     * @param args
     *            the command line arguments
     */
    public static void main(String[] args) {
        SimpleReader in = new SimpleReader1L();
        SimpleWriter out = new SimpleWriter1L();
        /*
         * Get input file name
         */
        out.print("Enter valid BL statement(s) file name: ");
        String fileName = in.nextLine();
        /*
         * Parse input file
         */
        out.println("*** Parsing input file ***");
        Statement s = new Statement1Parse1();
        SimpleReader file = new SimpleReader1L(fileName);
        Queue<String> tokens = Tokenizer.tokens(file);
        file.close();
        s.parse(tokens); // replace with parseBlock to test other method
        /*
         * Pretty print the statement(s)
         */
        out.println("*** Pretty print of parsed statement(s) ***");
        s.prettyPrint(out, 0);

        in.close();
        out.close();
    }

}

```