```java
import components.program.Program;

/**
 * Program to test method to interpret a BugsWorld virtual machine program.
 *
 * @author Gage Farmer
 *
 */
public final class BugsWorldVMInterpreter {

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

    /**
     * BugsWorld possible cell states.
     */
    enum CellState {
        EMPTY, WALL, FRIEND, ENEMY;
    }

    /**
     * Private constructor so this utility class cannot be instantiated.
     */
    private BugsWorldVMInterpreter() {
    }

    /**
     * Gets a file name from the user and loads a BL compiled program from the
     * corresponding file and returns an array containing the compiled program.
     *
     * @param in
     *            the input stream
     * @param out
     *            the output stream
     * @return the compiled BL program loaded from a file specified by the user
     * @updates in.content
     * @updates out.content
     * @requires in.is_open and out.is_open
     * @ensures <pre>
     * [prompts the user to enter a file name, inputs it, and loads a
     *  compiled BL program from the corresponding file and returns the
     *  compiled program]
     * </pre>
     */
    private static int[] loadProgram(SimpleReader in, SimpleWriter out) {
        int[] cp;
        out.print("Enter compiled BL program file name: ");
        String fileName = in.nextLine();
        SimpleReader file = new SimpleReader1L(fileName);
        int length = file.nextInteger();
        cp = new int[length];
        for (int i = 0; i < length; i++) {
            cp[i] = file.nextInteger();
        }
        file.close();
        return cp;
    }

```

```
67      /**
68       * Returns whether the given integer is the byte code of a BugsWorld virtual
69       * machine primitive instruction (MOVE, TURNLEFT, TURNRIGHT, INFECT, SKIP,
70       * HALT).
71       *
72       * @param byteCode
73       *            the integer to be checked
74       * @return true if {@code byteCode} is the byte code of a primitive
75       *         instruction or false otherwise
76       * @ensures <pre>
77       * isPrimitiveInstructionByteCode =
78       *  [true iff byteCode is the byte code of a primitive instruction]
79       * </pre>
80       */
81      private static boolean isPrimitiveInstructionByteCode(int byteCode) {
82          return (byteCode == Instruction.MOVE.byteCode())
83                  || (byteCode == Instruction.TURNLEFT.byteCode())
84                  || (byteCode == Instruction.TURNRIGHT.byteCode())
85                  || (byteCode == Instruction.INFECT.byteCode())
86                  || (byteCode == Instruction.SKIP.byteCode())
87                  || (byteCode == Instruction.HALT.byteCode());
88      }
89
90      /**
91       * Returns the value of the condition in the given conditional jump
92       * {@code condJump} given what the bug sees {@code wbs}. Note that if
93       * {@code condJump} is the byte code for the conditional jump
94       * JUMP_IF_NOT_condition, the value returned is the value of the "condition"
95       * part of the jump instruction.
96       *
97       * @param wbs
98       *            the {@code CellState} indicating what the bug sees
99       * @param condJump
100      *            the byte code of a conditional jump
101      * @return the value of the conditional jump condition
102      * @requires [condJump is the byte code of a conditional jump]
103      * @ensures <pre>
104      * conditionalJumpCondition =
105      *  [the value of the condition of condJump given what the bug sees wbs]
106      * </pre>
107      */
108     private static boolean conditionalJumpCondition(CellState wbs,
109             int condJump) {
110         final double half = 0.5;
111         boolean answer = true;
112         if (condJump == Instruction.JUMP_IF_NOT_NEXT_IS_EMPTY.byteCode()) {
113             answer = (wbs == CellState.EMPTY);
114         } else if (condJump == Instruction.JUMP_IF_NOT_NEXT_IS_NOT_EMPTY
115                 .byteCode()) {
116             answer = (wbs != CellState.EMPTY);
117         } else if (condJump == Instruction.JUMP_IF_NOT_NEXT_IS_WALL
118                 .byteCode()) {
119             answer = (wbs == CellState.WALL);
120         } else if (condJump == Instruction.JUMP_IF_NOT_NEXT_IS_NOT_WALL
121                 .byteCode()) {
122             answer = (wbs != CellState.WALL);
123         } else if (condJump == Instruction.JUMP_IF_NOT_NEXT_IS_FRIEND
124                 .byteCode()) {
125             answer = (wbs == CellState.FRIEND);
```

```
126         } else if (condJump == Instruction.JUMP_IF_NOT_NEXT_IS_NOT_FRIEND
127                 .byteCode()) {
128             answer = (wbs != CellState.FRIEND);
129         } else if (condJump == Instruction.JUMP_IF_NOT_NEXT_IS_ENEMY
130                 .byteCode()) {
131             answer = (wbs == CellState.ENEMY);
132         } else if (condJump == Instruction.JUMP_IF_NOT_NEXT_IS_NOT_ENEMY
133                 .byteCode()) {
134             answer = (wbs != CellState.ENEMY);
135         } else if (condJump == Instruction.JUMP_IF_NOT_RANDOM.byteCode()) {
136             answer = (Math.random() < half);
137         } else if (condJump == Instruction.JUMP_IF_NOT_TRUE.byteCode()) {
138             answer = true;
139         } else {
140             assert false : "Violation of: condJump is a conditional jump byte code";
141         }
142         return answer;
143     }
144 
145     /**
146      * Checks whether the given location {@code loc} is the location of an
147      * instruction byte code in the given program {@code cp}.
148      *
149      * @param cp
150      *            the compiler program
151      * @param loc
152      *            the location to check
153      * @return true iff {@code loc} is the address of an instruction byte code
154      * @requires [cp is a valid compiled BL program]
155      * @ensures <pre>
156      * isValidInstructionLocation =
157      *  [true iff loc is the address of an instruction byte code in cp]
158      * </pre>
159      */
160     private static boolean isValidInstructionLocation(int[] cp, int loc) {
161         boolean found = false;
162         int pos = 0;
163         while ((pos < cp.length) && !found) {
164             if (pos == loc) {
165                 found = true;
166             } else {
167                 if (!isPrimitiveInstructionByteCode(cp[pos])) {
168                     /*
169                      * It must be a jump instruction, increment pos one extra
170                      * time
171                      */
172                     pos++;
173                 }
174                 pos++;
175             }
176         }
177         return found;
178     }
179 
180     /*
181      * Public members -----------------------------------------------------
182      */
183 
184     /**
```

```
185      * Returns the location of the next primitive instruction to execute in
186      * compiled program {@code cp} given what the bug sees {@code wbs} and
187      * starting from location {@code pc}.
188      *
189      * @param cp
190      *            the compiled program
191      * @param wbs
192      *            the {@code CellState} indicating what the bug sees
193      * @param pc
194      *            the program counter
195      * @return the location of the next primitive instruction to execute
196      * @requires <pre>
197      * [cp is a valid compiled BL program]  and
198      * 0 <= pc < cp.length  and
199      * [pc is the location of an instruction byte code in cp, that is, pc
200      *  cannot be the location of an address]
201      * </pre>
202      * @ensures <pre>
203      * [return the address of the next primitive instruction that
204      *  should be executed in program cp given what the bug sees wbs and
205      *  starting execution at address pc in program cp]
206      * </pre>
207      */
208     public static int nextPrimitiveInstructionAddress(int[] cp, CellState wbs,
209             int pc) {
210         assert cp != null : "Violation of: cp is not null";
211         assert wbs != null : "Violation of: wbs is not null";
212         assert cp.length > 0 : "Violation of: cp is a valid compiled BL program";
213         assert 0 <= pc : "Violation of: 0 <= pc";
214         assert pc < cp.length : "Violation of: pc < cp.length";
215         assert isValidInstructionLocation(cp, pc) : ""
216                 + "Violation of: pc is the location of an instruction byte code in
    cp";
217
218         switch (cp[pc]) {
219             case 6:
220                 pc = cp[pc + 1];
221                 break;
222
223             case 7:
224                 if (wbs != CellState.EMPTY) {
225                     pc = cp[pc + 1];
226                 } else {
227                     pc++;
228                 }
229                 break;
230
231             case 8:
232                 if (wbs == CellState.EMPTY) {
233                     pc = cp[pc + 1];
234                 } else {
235                     pc++;
236                 }
237                 break;
238
239             case 9:
240                 if (wbs != CellState.WALL) {
241                     pc = cp[pc + 1];
242                 } else {
```

```
243                     pc++;
244                 }
245                 break;
246
247             case 10:
248                 if (wbs == CellState.WALL) {
249                     pc = cp[pc + 1];
250                 } else {
251                     pc++;
252                 }
253                 break;
254
255             case 11:
256                 if (wbs != CellState.FRIEND) {
257                     pc = cp[pc + 1];
258                 } else {
259                     pc++;
260                 }
261                 break;
262
263             case 12:
264                 if (wbs == CellState.FRIEND) {
265                     pc = cp[pc + 1];
266                 } else {
267                     pc++;
268                 }
269                 break;
270
271             case 13:
272                 if (wbs != CellState.ENEMY) {
273                     pc = cp[pc + 1];
274                 } else {
275                     pc++;
276                 }
277                 break;
278
279             case 14:
280                 if (wbs == CellState.ENEMY) {
281                     pc = cp[pc + 1];
282                 } else {
283                     pc++;
284                 }
285                 break;
286
287             case 15:
288                 if (Math.random() >= .5) {
289                     pc = cp[pc + 1];
290                 } else {
291                     pc++;
292                 }
293                 break;
294
295             case 16:
296                 if (!conditionalJumpCondition(wbs, cp[pc])) {
297                     pc = cp[pc + 1];
298                 } else {
299                     pc++;
300                 }
301                 break;
```

```
302
303             default:
304                 pc++;
305                 break;
306         }
307
308         // This line added just to make the program compilable.
309         return pc;
310     }
311
312     /**
313      * Main method.
314      *
315      * @param args
316      *            the command line arguments
317      */
318     public static void main(String[] args) {
319         SimpleReader in = new SimpleReader1L();
320         SimpleWriter out = new SimpleWriter1L();
321
322         /*
323          * Load compiled BL program
324          */
325         int[] cp = loadProgram(in, out);
326
327         int pc = 0;
328         out.println();
329         out.println("Enter program counter outside the [0," + cp.length
330                 + ") range to quit.");
331         /*
332          * Output disassembled program with marked address
333          */
334         out.println();
335         Program1.disassembleProgram(out, cp, pc);
336         while (true) {
337             /*
338              * Input new program counter
339              */
340             out.println();
341             out.print("Enter program counter (Enter => pc = " + pc + "): ");
342             String input = in.nextLine();
343             if (input.length() > 0 && !FormatChecker.canParseInt(input)) {
344                 out.println("Program counter must be a number in the [0,"
345                         + cp.length + ") range");
346                 continue;
347             } else if (FormatChecker.canParseInt(input)) {
348                 int pcCandidate = Integer.parseInt(input);
349                 if (pcCandidate < 0 || pcCandidate >= cp.length) {
350                     break;
351                 } else if (!isValidInstructionLocation(cp, pcCandidate)) {
352                     out.println("Program counter must be the location of an "
353                             + "instruction byte code in the program");
354                     continue;
355                 } else {
356                     pc = pcCandidate;
357                 }
358             }
359             /*
360              * Input what bug sees and convert to CellState value
```

```
361             */
362             out.print("Enter what bug sees "
363                     + "(EMPTY=0, WALL=1, FRIEND=2, ENEMY=3): ");
364             String wbs = in.nextLine();
365             CellState cs;
366             switch (wbs) {
367                 case "0": {
368                     cs = CellState.EMPTY;
369                     break;
370                 }
371                 case "1": {
372                     cs = CellState.WALL;
373                     break;
374                 }
375                 case "2": {
376                     cs = CellState.FRIEND;
377                     break;
378                 }
379                 case "3": {
380                     cs = CellState.ENEMY;
381                     break;
382                 }
383                 default: {
384                     out.println("What bug sees must be a number"
385                             + " in the [0,3] range");
386                     continue;
387                 }
388             }
389             /*
390              * Interpret program to find next primitive instruction
391              */
392             pc = nextPrimitiveInstructionAddress(cp, cs, pc);
393             out.println();
394             out.println("  Next primitive instruction: "
395                     + Program.Instruction.values()[cp[pc]].toString()
396                     + " at address " + pc);
397             /*
398              * Output disassembled program with marked address
399              */
400             out.println();
401             Program1.disassembleProgram(out, cp, pc);
402             /*
403              * Increment program counter pc to make progress
404              */
405             pc++;
406         }
407         out.println("Goodbye!");
408 
409         in.close();
410         out.close();
411     }
412 
413 }
414 
```