```java
import components.map.Map;

/**
 * {@code Program} represented the obvious way with implementations of primary
 * methods.
 *
 * @convention [$this.name is an IDENTIFIER] and [$this.context is a CONTEXT]
 *             and [$this.body is a BLOCK statement]
 * @correspondence this = ($this.name, $this.context, $this.body)
 *
 * @author Robert Frenken
 * @author Bennett Palmer
 *
 */
public class Program2 extends ProgramSecondary {

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

    /**
     * The program name.
     */
    private String name;

    /**
     * The program context.
     */
    private Map<String, Statement> context;

    /**
     * The program body.
     */
    private Statement body;

    /**
     * Reports whether all the names of instructions in {@code c} are valid
     * IDENTIFIERs.
     *
     * @param c
     *            the context to check
     * @return true if all instruction names are identifiers; false otherwise
     * @ensures <pre>
     * allIdentifiers =
     *   [all the names of instructions in c are valid IDENTIFIERs]
     * </pre>
     */
    private static boolean allIdentifiers(Map<String, Statement> c) {
        for (Map.Pair<String, Statement> pair : c) {
            if (!Tokenizer.isIdentifier(pair.key())) {
                return false;
            }
        }
        return true;
    }

    /**
```

```
65      * Reports whether no instruction name in {@code c} is the name of a
66      * primitive instruction.
67      *
68      * @param c
69      *            the context to check
70      * @return true if no instruction name is the name of a primitive
71      *         instruction; false otherwise
72      * @ensures <pre>
73      * noPrimitiveInstructions =
74      *   [no instruction name in c is the name of a primitive instruction]
75      * </pre>
76      */
77     private static boolean noPrimitiveInstructions(Map<String, Statement> c) {
78         return !c.hasKey("move") && !c.hasKey("turnleft")
79                 && !c.hasKey("turnright") && !c.hasKey("infect")
80                 && !c.hasKey("skip");
81     }
82
83     /**
84      * Reports whether all the bodies of instructions in {@code c} are BLOCK
85      * statements.
86      *
87      * @param c
88      *            the context to check
89      * @return true if all instruction bodies are BLOCK statements; false
90      *         otherwise
91      * @ensures <pre>
92      * allBlocks =
93      *   [all the bodies of instructions in c are BLOCK statements]
94      * </pre>
95      */
96     private static boolean allBlocks(Map<String, Statement> c) {
97         for (Map.Pair<String, Statement> pair : c) {
98             if (pair.value().kind() != Kind.BLOCK) {
99                 return false;
100            }
101        }
102        return true;
103    }
104
105    /**
106     * Creator of initial representation.
107     */
108    private void createNewRep() {
109
110        this.name = "Unnamed";
111        this.context = new Map1L<>();
112        this.body = new Statement1();
113
114    }
115
116    /*
117     * Constructors ------------------------------------------------------------
118     */
119
120    /**
121     * No-argument constructor.
```

```
122      */
123     public Program2() {
124         this.createNewRep();
125     }
126
127     /*
128      * Standard methods ---------------------------------------------------
129      */
130
131     @Override
132     public final Program newInstance() {
133         try {
134             return this.getClass().getConstructor().newInstance();
135         } catch (ReflectiveOperationException e) {
136             throw new AssertionError(
137                     "Cannot construct object of type " + this.getClass());
138         }
139     }
140
141     @Override
142     public final void clear() {
143         this.createNewRep();
144     }
145
146     @Override
147     public final void transferFrom(Program source) {
148         assert source != null : "Violation of: source is not null";
149         assert source != this : "Violation of: source is not this";
150         assert source instanceof Program2 : ""
151                 + "Violation of: source is of dynamic type Program2";
152         /*
153          * This cast cannot fail since the assert above would have stopped
154          * execution in that case: source must be of dynamic type Program2.
155          */
156         Program2 localSource = (Program2) source;
157         this.name = localSource.name;
158         this.context = localSource.context;
159         this.body = localSource.body;
160         localSource.createNewRep();
161     }
162
163     /*
164      * Kernel methods -----------------------------------------------------
165      */
166
167     @Override
168     public final void setName(String n) {
169         assert n != null : "Violation of: n is not null";
170         assert Tokenizer.isIdentifier(n) : ""
171                 + "Violation of: n is a valid IDENTIFIER";
172
173         this.name = n;
174
175     }
176
177     @Override
178     public final String name() {
```

```
179
180         return this.name;
181     }
182
183     @Override
184     public final Map<String, Statement> newContext() {
185
186         Map<String, Statement> map = this.context.newInstance();
187
188         return map;
189     }
190
191     @Override
192     public final void swapContext(Map<String, Statement> c) {
193         assert c != null : "Violation of: c is not null";
194         assert c instanceof Map1L<?, ?> : "Violation of: c is a Map1L<?, ?>";
195         assert allIdentifiers(
196                 c) : "Violation of: names in c are valid IDENTIFIERs";
197         assert noPrimitiveInstructions(c) : ""
198                 + "Violation of: names in c do not match the names"
199                 + " of primitive instructions in the BL language";
200         assert allBlocks(c) : "Violation of: bodies in c"
201                 + " are all BLOCK statements";
202
203         Map<String, Statement> map = this.context.newInstance();
204         // map now has value of c
205         map.transferFrom(c);
206         // c now has value of map
207         c.transferFrom(this.context);
208         // context now has value of c (currently variable map), swapping complete
209         this.context.transferFrom(map);
210
211     }
212
213     @Override
214     public final Statement newBody() {
215
216         Statement body = this.body.newInstance();
217
218         return body;
219     }
220
221     @Override
222     public final void swapBody(Statement b) {
223         assert b != null : "Violation of: b is not null";
224         assert b instanceof Statement1 : "Violation of: b is a Statement1";
225         assert b.kind() == Kind.BLOCK : "Violation of: b is a BLOCK statement";
226
227         Statement transferBody = this.body.newInstance();
228
229         transferBody.transferFrom(b);
230
231         b.transferFrom(this.body);
232
233         this.body.transferFrom(transferBody);
234
235     }
```

```
236
237 }
238
```