```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 Gabe Azzarita and Ty Fredrick
 *
 */
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;
    }

    /**
     * Reports whether no instruction name in {@code c} is the name of a
     * primitive instruction.
     *
```

```
67      * @param c
68      *            the context to check
69      * @return true if no instruction name is the name of a primitive
70      *         instruction; false otherwise
71      * @ensures <pre>
72      * noPrimitiveInstructions =
73      *   [no instruction name in c is the name of a primitive instruction]
74      * </pre>
75      */
76     private static boolean noPrimitiveInstructions(Map<String, Statement> c) {
77         return !c.hasKey("move") && !c.hasKey("turnleft")
78                 && !c.hasKey("turnright") && !c.hasKey("infect")
79                 && !c.hasKey("skip");
80     }
81
82     /**
83      * Reports whether all the bodies of instructions in {@code c} are BLOCK
84      * statements.
85      *
86      * @param c
87      *            the context to check
88      * @return true if all instruction bodies are BLOCK statements; false
89      *         otherwise
90      * @ensures <pre>
91      * allBlocks =
92      *   [all the bodies of instructions in c are BLOCK statements]
93      * </pre>
94      */
95     private static boolean allBlocks(Map<String, Statement> c) {
96         for (Map.Pair<String, Statement> pair : c) {
97             if (pair.value().kind() != Kind.BLOCK) {
98                 return false;
99             }
100        }
101        return true;
102    }
103
104    /**
105     * Creator of initial representation.
106     */
107    private void createNewRep() {
108
109        // Make sure to use Statement1 from the library
110        // Use Map1L for the context if you want the asserts below to match
111        this.name = "Unnamed";
112        this.context = new Map1L<>();
113        this.body = new Statement1();
114
115    }
116
117    /*
118     * Constructors ------------------------------------------------------------
119     */
120
121    /**
122     * No-argument constructor.
123     */
124    public Program2() {
125        this.createNewRep();
```

```
126     }
127
128     /*
129      * Standard methods ---------------------------------------------------
130      */
131
132     @Override
133     public final Program newInstance() {
134         try {
135             return this.getClass().getConstructor().newInstance();
136         } catch (ReflectiveOperationException e) {
137             throw new AssertionError(
138                     "Cannot construct object of type " + this.getClass());
139         }
140     }
141
142     @Override
143     public final void clear() {
144
145         this.createNewRep();
146     }
147
148     @Override
149     public final void transferFrom(Program source) {
150         assert source != null : "Violation of: source is not null";
151         assert source != this : "Violation of: source is not this";
152         assert source instanceof Program2 : ""
153                 + "Violation of: source is of dynamic type Program2";
154         /*
155          * This cast cannot fail since the assert above would have stopped
156          * execution in that case: source must be of dynamic type Program2.
157          */
158         Program2 localSource = (Program2) source;
159         this.name = localSource.name;
160         this.context = localSource.context;
161         this.body = localSource.body;
162         localSource.createNewRep();
163     }
164
165     /*
166      * Kernel methods -----------------------------------------------------
167      */
168
169     @Override
170     public final void setName(String n) {
171         assert n != null : "Violation of: n is not null";
172         assert Tokenizer.isIdentifier(n) : ""
173                 + "Violation of: n is a valid IDENTIFIER";
174
175         this.name = n;
176
177     }
178
179     @Override
180     public final String name() {
181
182         return this.name;
183     }
184
```

```
185     @Override
186     public final Map<String, Statement> newContext() {
187
188         return this.context.newInstance();
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         // make temp to hold old context, then switch
204         Map<String, Statement> temp = this.newContext();
205         temp.transferFrom(this.context);
206         this.context.transferFrom(c);
207         c.transferFrom(temp);
208
209     }
210
211     @Override
212     public final Statement newBody() {
213
214         return this.body.newInstance();
215     }
216
217     @Override
218     public final void swapBody(Statement b) {
219         assert b != null : "Violation of: b is not null";
220         assert b instanceof Statement1 : "Violation of: b is a Statement1";
221         assert b.kind() == Kind.BLOCK : "Violation of: b is a BLOCK statement";
222
223         // make temp body to hold old body, then switch
224         Statement temp = this.newBody();
225         temp.transferFrom(this.body);
226         this.body.transferFrom(b);
227         b.transferFrom(temp);
228
229     }
230
231 }
232
```