```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 Krish Patel and Chloe Feller
 *
 */
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
```

```java
     * primitive instruction.
     *
     * @param c
     *            the context to check
     * @return true if no instruction name is the name of a primitive
     *         instruction; false otherwise
     * @ensures <pre>
     * noPrimitiveInstructions =
     *   [no instruction name in c is the name of a primitive instruction]
     * </pre>
     */
    private static boolean noPrimitiveInstructions(Map<String, Statement> c) {
        return !c.hasKey("move") && !c.hasKey("turnleft")
                && !c.hasKey("turnright") && !c.hasKey("infect")
                && !c.hasKey("skip");
    }

    /**
     * Reports whether all the bodies of instructions in {@code c} are BLOCK
     * statements.
     *
     * @param c
     *            the context to check
     * @return true if all instruction bodies are BLOCK statements; false
     *         otherwise
     * @ensures <pre>
     * allBlocks =
     *   [all the bodies of instructions in c are BLOCK statements]
     * </pre>
     */
    private static boolean allBlocks(Map<String, Statement> c) {
        for (Map.Pair<String, Statement> pair : c) {
            if (pair.value().kind() != Kind.BLOCK) {
                return false;
            }
        }
        return true;
    }

    /**
     * Creator of initial representation.
     */
    private void createNewRep() {

        this.name = "Unnamed";
        this.context = new Map1L<String, Statement>();
        this.body = new Statement1();

        // Make sure to use Statement1 from the library
        // Use Map1L for the context if you want the asserts below to match

    }

    /*
     * Constructors ------------------------------------------------------------
     */

```

```java
    /**
     * No-argument constructor.
     */
    public Program2() {
        this.createNewRep();
    }

    /*
     * Standard methods ---------------------------------------------------------
     */

    @Override
    public final Program newInstance() {
        try {
            return this.getClass().getConstructor().newInstance();
        } catch (ReflectiveOperationException e) {
            throw new AssertionError(
                    "Cannot construct object of type " + this.getClass());
        }
    }

    @Override
    public final void clear() {
        this.createNewRep();
    }

    @Override
    public final void transferFrom(Program source) {
        assert source != null : "Violation of: source is not null";
        assert source != this : "Violation of: source is not this";
        assert source instanceof Program2 : ""
                + "Violation of: source is of dynamic type Program2";
        /*
         * This cast cannot fail since the assert above would have stopped
         * execution in that case: source must be of dynamic type Program2.
         */
        Program2 localSource = (Program2) source;
        this.name = localSource.name;
        this.context = localSource.context;
        this.body = localSource.body;
        localSource.createNewRep();
    }

    /*
     * Kernel methods -----------------------------------------------------------
     */

    @Override
    public final void setName(String n) {
        assert n != null : "Violation of: n is not null";
        assert Tokenizer.isIdentifier(n) : ""
                + "Violation of: n is a valid IDENTIFIER";

        this.name = n;

    }

```

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

236