```java
package backend;

import java.util.HashMap;

import soot.Unit;
import soot.Value;
import soot.jimple.IntConstant;
import soot.jimple.Jimple;
import soot.jimple.NopStmt;
import soot.util.Chain;
import ast.Block;
import ast.BreakStmt;
import ast.ExprStmt;
import ast.IfStmt;
import ast.ReturnStmt;
import ast.Stmt;
import ast.Visitor;
import ast.WhileStmt;

/**
 * This class is in charge of creating Jimple code for a given statement (and its nested
 * statements, if applicable).
 */
public class StmtCodeGenerator extends Visitor<Void> {
    /** Cache Jimple singleton for convenience. */
    private final Jimple j = Jimple.v();

    /** The {@link FunctionCodeGenerator} that created this object. */
    private final FunctionCodeGenerator fcg;

    /** The statement list of the enclosing function body. */
    private final Chain<Unit> units;

    /** A map from while statements to their break target. */
    private final HashMap<WhileStmt, Unit> breakTargets = new HashMap<WhileStmt, Unit>();

    public StmtCodeGenerator(FunctionCodeGenerator fcg) {
        this.fcg = fcg;
        this.units = fcg.getBody().getUnits();
    }

    /** Generates code for an expression statement. */
    @Override
    public Void visitExprStmt(ExprStmt nd) {
        /* TODO: generate code for expression statement (hint: use ExprCodeGenerator.generate) */
        ExprCodeGenerator.generate(nd.getExpr(), fcg);
        return null;
    }

    /** Generates code for a break statement. */
    @Override
    public Void visitBreakStmt(BreakStmt nd) {
```

```
53          /* TODO: generate code for break statement (hint: use ASTNode.
    getEnclosingLoop and breakTargets;
54           *       use units.add() to insert the statement into the surrounding
    method) */
55          WhileStmt wstmt = nd.getEnclosingLoop();
56          Unit bt = this.breakTargets.get(wstmt);
57          units.add(j.newGotoStmt(bt));
58          return null;
59      }
60
61      /** Generates code for a block of statements. */
62      @Override
63      public Void visitBlock(Block nd) {
64          for(Stmt stmt : nd.getStmts())
65              stmt.accept(this);
66          return null;
67      }
68
69      /** Generates code for a return statement. */
70      @Override
71      public Void visitReturnStmt(ReturnStmt nd) {
72          Unit stmt;
73          if(nd.hasExpr())
74              stmt = j.newReturnStmt(ExprCodeGenerator.generate(nd.getExpr(), fcg))
    ;
75          else
76              stmt = j.newReturnVoidStmt();
77          units.add(stmt);
78          return null;
79      }
80
81      /** Generates code for an if statement. */
82      @Override
83      public Void visitIfStmt(IfStmt nd) {
84          Value cond = ExprCodeGenerator.generate(nd.getExpr(), fcg);
85          NopStmt join = j.newNopStmt();
86          units.add(j.newIfStmt(j.newEqExpr(cond, IntConstant.v(0)), join));
87          nd.getThen().accept(this);
88          if(nd.hasElse()) {
89              NopStmt els = join;
90              join = j.newNopStmt();
91              units.add(j.newGotoStmt(join));
92              units.add(els);
93              nd.getElse().accept(this);
94          }
95          units.add(join);
96          return null;
97      }
98
99      /** Generates code for a while statement. */
100     @Override
101     public Void visitWhileStmt(WhileStmt nd) {
102         /* TODO: generate code for while statement as discussed in lecture; add
    the NOP statement you
103          *       generate as the break target to the breakTargets map
```

```
104           */
105          NopStmt start = j.newNopStmt();
106          units.add(start);
107          Value cond = ExprCodeGenerator.generate(nd.getExpr(), fcg);
108          NopStmt end = j.newNopStmt();
109          breakTargets.put(nd, end);
110          units.add(j.newIfStmt(j.newEqExpr(cond, IntConstant.v(0)), end));
111          nd.getBody().accept(this);
112          units.add(j.newGotoStmt(start));
113          units.add(end);
114          return null;
115      }
116  }
117
```