Rule Development Manual

Contents

- Introduction
- API alternatives
    - Abstract Syntax Tree (low-level and high-level) API
    - NavigableNode / TreeNode API
    - XPath rules
    - Query API
    - Additional static analysis facilities
        - Control-flow graph (CFG) and data-flow analysis
        - Library metadata
        - Tainting propagation
        - Local / Global Symbol Table
            - Global Symbol Table (GST)
            - Local Symbol Table (LST)

Introduction

This guide explains the rule development facilities in Kiuwan.

Kiuwan engine is the static analysis platform embedded in Kiuwan products. With it, it is possible to perform many different types of static analysis on source code:

- Verification of compliance with coding standards: are standards been followed in the software implementation?
- Inefficiencies in code (examples: string concatenation, improper use of synchronization, unnecessary object construction...).
- Security checks (like tainting propagation, to see if a user-controller input can reach a "sink", a resource like a database, without proper validation).
- Dependency analysis, to detect design smells (god class, implementation dependency, low cohesion and high coupling), or architecture violations (a layer is using another layer, and should not).
- Code metrics evaluation. A code metric on a software artifact represents a certain property (like size or coupling) of the analyzed software.

Each check is coded in a unit, called Kiuwan RULE, comprising of:

- Check code (a Java class),
- A rule definition (XML file) including configuration and documentation.

The **rule definition XML descriptor** contains:

- An identifier,
- An implementation classname,
- Technologies that analyze,
- Message/description,
- Priority (1=critical ... 5=informative),
- Configuration properties,
- Examples for a bad / repair code,
- Benefits and incovenients for adopting the rule, etc.

Rules are aggregated in rulesets (parts of the quality model), used during static analysis.

When a flaw (non-compliant code) is detected in a certain source file, a violation is emitted and added to the analysis report. Metric rules create a metric value for a certain software artifact instead.

The guide exposes the different alternatives the rule programmer has when developing rules for automated static analysis verifications, and how to extend the Kiuwan engine platform for more advanced usages.

Prerequisites: Kiuwan is developed in Java, so good skills on Java programming (JDK only, not J2EE) and design principles are needed for rule developers.

Note: For detailed documentation of the API classes, consult the JavaDocs provided with the API deliverable.

API alternatives

There are different alternatives for developing rules in Kiuwan, according to the level of abstraction: Abstract Syntaxt Tree, NavigableNode, XPath or Query API.

Certain primitive objects could be passed in many parts of the rules API and are used in many places in the Kiuwan API:

Primitive	Effect	Method to implement
NodeVisitor	Apply logic to node	void visit(BaseNode)
NodePredicate	Does this node satisfy X?	boolean is(BaseNode)
NodeSearch	"Find a related node from a given node"	BaseNode apply(BaseNode)
Navigation	Perform certain navigation on AST starting at given node, possibly applying another primitive	NodeSet navigate(BaseNode) NodeSet navigate(BaseNode, NodePredicate) void visit(BaseNode, NodeVisitor)

As examples for typical `NodeSearch`, think of finding the variable declaration for a certain variable usage, function definition for a call, container node for any other node, etc.

`Navigation` combines traversal + primitive, used in Query API (discussed later).

Abstract Syntax Tree (low-level and high-level) API

Kiuwan engine analyzers parse source code using a programming language grammar parser. The parser translates plain text to a tree-based representation of the source code (named *Abstract Syntax Tree*). This tree is a low-level representation of the source code file, and represents in a very detailed way all the elements, from top-level language elements (like class or function definitions) to the literals and expressions contained in the source file. Low-level AST is named **LLA** in what follows.

For certain languages, an alternate high-level AST (**HLA**) is produced from the low-level AST. The high-level AST represents the main structures in source code, frequently expressed as nodes in a language-independent way.

The low-level AST (LLA) may have too much detail for what is necessary to implement many rules. The high-level AST is a condensed view of the source syntax, showing relevant items for the target programming language (programs, types, functions, statements), but remove punctuation nodes, or detail nodes (e.g. expressions are leaf nodes in the HLA).

For processing AST (both LLA or HLA), rules could extend `com.als.core.AbstractRule`, and implement logic in method `void visit(BaseNode ast, RuleContext ruleContext)`. The first argument is the AST root node (e.g. a `CompilationUnit` in Java grammar), and the rule logic must navigate the AST to look for the condition that the rule detects. Navigating the AST is rather difficult, but this level of abstraction is adequate for rules that need access to the finest details of the source code, that may not be afforded with the rest of the APIs.

Technically, AST is provided as a tree where each node implements the interface `com.als.core.ast.BaseNode`. For some technologies, nodes extends this interface (like `com.als.core.ast.NavigableNode` or `com.als.core.ast.TreeNode`, see below) that provide extra facilities for searching nodes in the AST.

For most languages, a single AST node type for each language construct is provided (named CobolNode, PhpNode, CppNode, ...) implementing the common interface BaseNode.
For a few languages (Java, JSP, PowerScript, VB6), a different Java type is provided for each different grammar construct. The AST nodes provide accessors (*getters*) to fetch properties.

To help processing AST for common needs, a set of utility classes are provided for each technology. See JavaDoc for full details.

NavigableNode / TreeNode API

This API (`com.als.core.ast.TreeNode` class) decorates a BaseNode for any language, exposing `NavigableNode` interface that simplifies node searches and navigations. Could be used for many simple rules that do not need advanced static analysis facilities.

Note: For certain technologies (like Cobol, JavaScript or PHP), LLA already implements NavigableNode, which does not need to be wrapped in TreeNode to access extra search methods exposed.

TreeNode is a decorated BaseNode that support many navigation operations. Provide methods for doing something (finding first, finding all, counting, checking, navigating...) on a certain navigation "axis" in the tree with respect to wrapped node (children, successors, brothers at left/right, parent, ancestors), using certain constraints (a predicate, node types), and using a processing paradigm (a NodeVisitor, code in a for-each loop...).

Decorating a BaseNode is simple: `TreeNode node = TreeNode.on(baseNode)`.

Child (direct or immediate children) axis:

- acceptChildren(NodeVisitor), apply a certain operation (encoded in visitor) to this node and its immediate children.
- child(NodePredicate), to find first direct child matching predicate.
- findAllChildren(NodePredicate), returns list of child nodes matching predicate.

- countChildren(NodePredicate) and similar methods, to count children matching predicate.
- hasChildren(NodePredicate), return true if at least one child matches predicate.

TreeNode itself is an iterable that iterates on direct children:

```
for(TreeNode child : treenode) {...}
```

will iterate on treenode children, left to right.

Successor (subtree) axis:

- find(NodePredicate, boolean), to look for first node matching certain condition
- findAll(NodePredicate, boolean), to look for all occurences of nodes matching a certain condition
- accept(NodeVisitor, boolean), to apply a certain operation to this node and all of its descendants (parent-first)
- has(NodePredicate) and similar methods, for checking if at least one of the successors matches predicate
- acceptChildrenFirst(NodeVisitor, boolean), to apply a certain operation to all of this node descendants, then to this node (children-first)
- subtreeBreadthFirst(NodePredicate), to use in a for-each loop to process successor nodes matching predicate (traversed breadth-first)
- subtreeDepthFirst(NodePredicate), also with for-each loop as above (but traversed depth-first)
- count(NodePredicate), to count nodes matching predicate

Parent (antecessor) axis:

- findFirstAncestor(NodePredicate) or ancestor(NodePredicate), to look for first node matching certain condition
- findAllAncestors(NodePredicate), to look for all occurences of nodes matching a certain condition
- acceptAncestors(NodeVisitor), to apply a certain operation to this node and all of its descendants
- countAncestors(NodePredicate), count of ancestors matching predicate
- hasAncestor(NodePredicate), returns true if at least one of the ancestors in path from this to root matched predicate
- onAncestor(NodePredicate), Iterable that permits to process in a for-loop each node in path from this to root matching predicate

Sibling axis:

- getLeftSibling() The immediate brother to the left
- getLeftSiblings() Iterator to brothers to the left, traversed from right to left.
- onLeftSiblings(NodePredicate) Iterable for processing left brothers that match predicate in a for-each loop.
- getRightSibling() The immediate brother to the right
- getRightSiblings() Iterator to brothers to the right, traversed from left to right.
- onRightSiblings(NodePredicate) Iterable for processing right brothers that match predicate in a for-each loop.

Additionally there are safe simple navigation methods that check if the requested parent or child exists, returning NULLTREE so methods could be chained without fear of runtime exceptions.

Example:

```
TreeNode n = TreeNode.on(node);
// from n, get first child, then first child of type {{Expression}},
// then go up to parent, then go to second node, then first, then first
again
n = n.child(0).child("Expression").parent().path(1, 0, 0);
if(n.isNotNull()) ....; // Voila! such complex navigation reached node of
interest...
```

In rules this facility can be used to focus on "node detection" logic, with navigation logic hidden in the navigation methods of this class.
Sample usage:

```
new TreeNode(root).find("ProcedureDivision").accept(new NodeVisitor() {
  public void visit(BaseNode node) {
    // Logic for any node under Cobol PROCEDURE DIVISION
  }
}
```

XPath rules

XPath is a language for searching in trees (typically XML trees, but it could be applied to AST as well).

The simplest way to code a Kiuwan rule is an XPath rule, that express using XPath notation the condition that AST nodes must match to be considered violations of the rule. An XPath rule is **declarative**, and does not need to be programmed, but require a certain knowledge of the AST.

For example, the following XPath expression detects loops without initialization nor update, that may be replaced by easier-to-understand `while` loops (last predicate exludes "for each" loops):

```
//ForStatement
  [count(*)>1]
  [not(ForInit)]
  [not(ForUpdate)]
  [not(Type and Expression and Statement)]
```

Adequate for simple rules. Typically, naming conventions could be easily verified using XPath expressions.

For example, if we are looking for the usages of `System.gc()` in Java code, the rule may use the XPath expression:

```
//Name[@Image='System.gc' or @Image='java.lang.System.gc']
```

NOTE: XPath attribute axis (`@property`) fetches a AST node property (getter returning a primitive type). For example, `@Image` invokes getImage() method on context AST nodes.

XPath-based rules could be implemented configuring `com.als.core.rule.XPath`, passing proper XPath expression in *xpath* rule property.

Many convenient XPath functions are provided, for full details on XPath, see the XPath section.

Query API

The class `com.optimyth.qaking.highlevelapi.dsl.Query` represents a query in an abstract syntax tree (or high-level tree). Query provides a fluent interface for expressing a search on AST, specifying the sequence of operations (find, filter, navigate, visit) to perform, starting at a given set of nodes. Each operation is configured by passing primitive objects (NodePredicate, NodeVisitor, NodeSearch or Navigation). Then one of the `run()` methods should be called to execute the operations registered.

As a simple example, imagine that you need to report getter methods that return null. Such rule could be implemented in a few code lines:

```
Predicate isGetter = ...;
NodePredicate returnsNull = ...;
Query q = Query.query()
  .find(methods(isGetter))
  .filter(returnsNull)
  .report();
...
// execute query from high-level root node
q.run(rule, ctx, ctx.getHighLevelTree());
```

Appropiate primitives could be found for each supported language; for example, for Java, `com.optimyth.qaking.java.hla.JavaPredicates` provides instances for `returnsNull` and `isGetter`.

When a query is executed using `run()` over a certain set of initial nodes (typically the root HLA or LLA node), nodes reached by each operation are remembered and act as context nodes for the next operation in sequence.

Operations that could be registered in a Query are:

Operation	Effect	Signature
find sucessors	Find all successors matching predicate	find(NodePredicate match)

find following a navigation	Find nodes traversed by navigation, matching predicate	find(NodePredicate match, Navigation navigation)
navigate	Traverses the given navigation from current (context) nodes	navigate(Navigation) navigate(NodeSearch) navigate(String xpath)
filter	Filter current context nodes	filter(NodePredicate) filter(Query)
visit	Visits each node in current context	visit(NodeVisitor)
navigate & visit	Visit each node reachable via given navigation with visitor	visit(NodeVisitor, Navigation)
custom operation	Registers a custom operation	operation(QueryOperation)
snapshot	Create a "snapshot" of current context nodeset, giving it a name	snapshot(String)
report	Emit a rule violation for each current context node or snapshot, using the ToViolation object to custimize violation to emit	report() report(String snapshot) report(ToViolation) report(String, ToViolation)
execute query	Executes the query, specifying initial node(s)	run(Rule, RuleContext) run(Rule, RuleContext, BaseNode...) run(Rule, RuleContext, NodeSet)

If you know about XPath, the `com.optimyth.qaking.highlevelapi.navigation.Region` provides Navigation instances for each XPath axis:

Region name	XPath axis	Nodes traversed
SELF	self::	context node itself
ROOT	/	go to root node
CHILDREN	child::	immediate children
PARENT	parent::	parent node
ANCESTORS	ancestor-or-self::	ancestors, including self
SUCCESSORS	descendant::	subtree nodes from node, not including itself
LEFTSIBLINGS	preceding-sibling::	Siblings of node at left, not including itself
RIGHTSIBLINGS	following-sibling::	Siblings of node at right, not including itself
PRECEDING	preceding::	Nodes appearing before node (before in code text)
FOLLOWING	following::	Nodes appearing after node (before in code text)

NOTE - Thread-safety: Query is thread-safe if primitives passed are thread-safe. This means that you may use a Query object as instance field of a rule, and call query.run() on the visit() method of the rule to process each input source, in multi-thread analysis this do not produce race conditions.

Examples:

Find unused vars (Java)

```
private static final Query unusedVars = Query.query()
  .find( varsPredicate )
  // An unused var should not have initialization with side-effects
  // (because then, declaration cannot be removed)
  .filter( not(hasSideEffectInInitPred) )
  .filter( not(hasUsages) )
  .report( reportVarName ); // special reporting
```

See `com.optimyth.qaking.rules.samples.java.UnusedVars` sample rule for the full implementation.

Avoid data references in arithmetic expressions where data item is DISPLAY (Cobol)

```
private final Query query = Query.query()
  .find(dataRefInArithStmt) // get data references in arithmetic statement
  .filter(isDisplayType) // ... but only DISPLAY / DISPLAY-1 types
  .report();
```

See `com.optimyth.qaking.rules.samples.cobol.NoDisplayDataInArithmeticOp` sample rule for full implementation.

Additional static analysis facilities

Static analysis is more than a syntax issue. Some well-known static analysis elements are: type resolution, symbol table, data-flow analysis, tainting propagation, constant or expression propagation, semantic metadata for technology APIs (and many more).

For certain technologies, extended static analysis facilities are provided. Most of them are located in the h`ighLevelAPI.jar` and `codeAnalysis.jar` JAR files.

Control-flow graph (CFG) and data-flow analysis

The **control-flow graph** is a graph where nodes represent statements and other code items (like variable declarations or function declarations). Nodes are connected according to the control flow (a statement s1 is connected to statement s2 if s2 may follow execution of s1 under certain conditions). Traversal of the CFG is useful to follow control logic, and certain properties of data flowing through the statements in the CFG could be derived statically (this is called "data-flow analysis").

An instance of the CFG is compiled for each behavioural unit (like a function or method). For some languages (e.g. in Cobol) the behavioural unit considered could be more global (e.g. the whole program in Cobol).

Dummy start and end nodes are added to the CFG (typically all exit points terminate in the end node, while the start node represents the entry point to the behavioural unit).

Two objects are used: `DataFlowNode` (represents a node in the CFG) and `DataFlowGraph` (represents a CFG for a certain behavioural unit). These elements are under `com.optimyth.qaking.codeanalysis.controlflow` package.

AST nodes for languages with CFG support implement the HasCFG interface:

```
DataFlowGraph getDataFlowGraph()
boolean hasDataFlowGraph()
DataFlowNode getDataFlowNode()
boolean hasDataFlowNode()
```

To build the CFG containing a certain AST node, the API provides instances of `ControlFlowSupport`, builder of the CFG from the AST for a particular behavioural unit. For example, for JavaScript and Cobol:

```
DataFlowGraph cfg = new JavascriptControlFlowSupport(ctx).getFlowGraph
(function);
DataFlowGraph cfg =  new CobolControlFlowSupport(ctx).getFlowGraph
(procedureDivision);
```

Two operations are typical with the CFG:

- Traversal (forward- or backward-, depth- or breadth-first) from a starting point, that could traverse the CFG (without infinite loops, as there may be cycles in the CFG) following the desired navigation. The `com.optimyth.qaking.codeanalysis.controlflow.ControlFlowNavigator` provides the navigation methods, typically calling a user-provided `ControlFlowVisitor` with each CFG node traversed in the navigation.
- Process all potential paths between two CFG nodes: `com.optimyth.qaking.codeanalysis.controlflow.paths.PathFinder`, that discover potential distinct paths during a certain CFG traversal and calls an user-provided `PathVisitor` with each matching path found.

The following image summarizes the API for operating with control-flow graph:

blocked URL

Library metadata

Document the behaviour for items (global variables, macros, functions / methods, types / classes...) in external APIs (e.g. class or function libraries common for target technology).

Library metadata facility permits to declare the metadata information (in an XML file descriptor) for each 'library', and to find metadata for a particular entity.

A `Libraries` object provides a façade for queries on library metadata. The relevant classes are located under package `com.optimyth.qaking.codeanalysis.metadata.model` and subpackages.

Typically, the base rule (or utility) for technologies supporting Library Metadata provides a `Libraries loadLibraries(RuleContext)` method to get the `Libraries` instance for a particular programming language.

Entity items have different incarnations: `LibraryDescriptor, FieldDescriptor, FunctionDescriptor, GlobalObject, HeaderDescriptor, MacroDef, Type, ClassDescriptor, MethodDescriptor`.

Library Metadata is used by *tainting propagation* facility in order to document the behaviour of API items. For tainting propagation, items may declare a Neutralization (declaring a point where inputs are "neutralized", *Source* (declaring an external input point) or *Sink* (declaring a point where the API gives access to a resource).

See javadoc for full details. Library Metadata XML descriptors could be found under the `JKQA_HOME /libraries` directory.

Tainting propagation

Detect a dataflow path connecting a 'source' (info input) and 'sink' (a point where API provides access to a sensitive resource). Relevant for security flaws (like 'injection vulnerabilities': SQL injection, cross-site scripting, etc.) Leverages control-flow graph and library metadata to perform.

Variables affected by sources (like external inputs to program) are 'tainted', and this condition propagates according to the semantics of statements and API items between sources and sinks. Taintedness status may be intercepted possibly by a 'neutralization' (a point where data is checked or transformed), that 'untaints' the data.

For technologies with tainting propagation support, a tainting rule simply declares what sources, sinks and neutralizations should be considered. For example, a "Path Traversal" security rule in PHP is simply:

Path traversal rule for PHP

```
public class PathTraversalRule extends AbstractPhpTaintingRule {
  private static final String SINK_KIND = "path_traversal";

  private List<SinkChecker> checkers;

  @Override public void initialize(RuleContext ctx) {
    super.initialize(ctx);
    Predicate<SinkDef> sinksPred = getPredicate(SINK_KIND);
    checkers = getSinkCheckers(sinksPred, ctx);
  }

  @Override protected void visit(BaseNode root, RuleContext ctx) {
    propagateTainting(root, getSourcesPredicate(), checkers, ctx);
  }
}
```

Local / Global Symbol Table

Model code items with a given name (a "symbol": functions, types, global variables...) that could be later searched for.

Symbol Table could be *Global* (**GST**: all source code inputs are parsed and processed first to compile global symbols), or *Local* (**LST**: only the symbols declared in current source unit are compiled).

Global Symbol Table (GST)

Entities modelled in the GST are: `SourceFile, Function, Program, Relation, Type, TypeMethod, Variable` (under package {{com.optimyth.qaking.globalmodel.model }}, see JavaDoc for full details).

Certain dependencies (like inheritance) are modelled in the GST.

The GST facilitates global analysis, but should be used with caution, as it is not as efficient in execution times as pure local analysis.

Note: As building the GST is expensive, it is built only if at least one of the rules in the analysis declares its intention to use GST (rule decorated with the `@UseGlobalSymbolTable` annotation).

The façade `SymbolTable` (in `com.optimyth.qaking.globalmodel.query` package) provides access to the GST:

```
SymbolTable SymbolTable.get(RuleContext); // Get instance representing the
GST
```

SymbolTable provides many query methods for finding item(s) in the GST, but additional query utilities are provided for extensive searches on certain item types:

- `FunctionQuery`: queries for function and method entities.
- `InheritanceQuery`: queries on type inheritance information.
- `TypeQuery`: queries over types (classes).
- `VariableQuery`: queries over instance variables (fields) and global variables.

See javadoc for `com.optimyth.qaking.globalmodel.query` package for full details.

For example, to check for indirect inheritance on java.io.Serializable interface:

```
SymbolTable table = SymbolTable.get(ctx);
if(table != null) {
  // Check only indirect inheritance (direct inheritance is processed in
visit)
  InheritanceQuery inh = new InheritanceQuery(table);
  for(InheritanceRow rel : inh.findInheritanceRows("supername='java.io.
Serializable' AND level>1")) {
    String subtype = rel.getSubtype();
    Variable field = table.findVariable(FIELD_NAME, subtype, "java");
    ...
  }
}
```

See sample rule `com.optimyth.qaking.rules.samples.java.SerializableWithVersionUid` under `JKQA_HOME/doc/samples` for full details.

`SymbolTable` provide a mechanism for getting AST of code files containing declarations referenced in current AST ("inter-AST searches"). As an example, imagine that the definition of functions called at certain points need to be processed, and that definition is not in the code file with the call:

```
SymbolTable table = SymbolTable.get(ctx);
...
BaseNode callNode = ...;
String calledFunction = FunctionUtil.getShortFunctionName(callNode);
// Find definition for the called function, by name
Function def = table.findFunction(calledFunction, null, null);
if(def != null) {
  // Loads the AST with the function declaration
  BaseNode defNode = table.loadNode(def);
  ...
}
```

Local Symbol Table (LST)

For certain technologies (see table below), a **Local Symbol Table** (LST) could be compiled efficiently to model specific symbols defined in current code unit. LSTs are used, for example, for locating usages for a variable declaration, or declaration for a variable usage.

A few examples may help to understand how LSTs are compiled and used:

LST for PHP

```
 LocalSymbolTable symtab = LocalSymbolTableBuilder.getSymbolTable((PhpNode)
root);
 symtab.visitForward(new Visitor() {
   public boolean onSymbol(Symbol symbol) {
     // ignore symbols with usages or global
     if(symbol.hasUsages() || symbol.isMagicConstant()) return true;

     if(symbol.getKind()== SymbolKind.PARAMETER) {
       // Check that the parameter symbol is in a function with body.
       // Interface methods and abstract methods do not use their
parameters
       if(hasBody(symbol.getNode())) {
         report(symbol, "{0}: unused parameter {1}", ctx);
       }

     } else if(isPrivateField(symbol)) {
       report(symbol, "{0}: unused private field {1}", ctx);

     } else if(isPrivateMethod(symbol)) {
       report(symbol, "{0}: unused private method {1}()", ctx);
     }
     return true;
   }
 });
```

See JavaDoc for `com.optimyth.qaking.php.symboltable.LocalSymbolTable` class.

LST for JavaScript

```
// Build symbol table for this source unit
LocalSymbolTable symTable = LocalSymbolTable.build((JSNode)root);

// Fetch configured globals in source code comment
Set<String> configuredGlobals = getGlobals((JSNode) root);

// Report unused vars
List<SymbolEntry> unusedList = symTable.getUnusedSymbols(UNUSED,
NodePredicate.TRUE);
for(SymbolEntry unused : unusedList) {
  String msg = getMessage() + ": unused symbol " + unused.getSymbol().
getName();
  reportViolation(ctx, unused.getDefinition(), msg);
}

// Report undefined vars used
List<SymbolEntry> undefList = symTable.getGlobalSymbols(UNDEF);
for(SymbolEntry undef : undefList) {
  if(configuredGlobals.contains( undef.getName() )) continue;
  String msg = getMessage() + ": undefined symbol " + undef.getSymbol().
getName();

  for(JSNode undefUsage : undef.getUsages()) {
    reportViolation(ctx, undefUsage, msg);
  }
}
```

See sample rule `com.optimyth.qaking.rules.samples.javascript.AvoidUndefUnusedVars`.

LST for Java

```
public class AvoidLocalVariablesDifferUpperLowerCase extends AbstractRule {

        protected void visit(BaseNode root, final RuleContext ctx) {
    if (!(root instanceof ASTCompilationUnit)) return;
    ASTCompilationUnit cu = (ASTCompilationUnit) root;
    initLocalSymbolTable(cu);

    TreeNode.on(root).accept(new NodeVisitor() {
      public void visit(BaseNode node) {
        if(node instanceof ASTLocalVariableDeclaration) {
          ASTLocalVariableDeclaration varDecl =
(ASTLocalVariableDeclaration)node;
          Set<String> lowerVars = Sets.newHashSet();
          for(ASTVariableDeclaratorId varName : varDecl.getVariableIds()) {
            lowerVars.add(varName.getImage().toLowerCase());
          }
          Scope scope = varDecl.getScope();
          while(scope instanceof LocalScope) {
            findCollidingVar((LocalScope)scope, varDecl, lowerVars, ctx);
            scope = scope.getParent();
          }
        }
      }
    });
        }

  /**
   * Check that local variables in scope collide with the variable names
declared in varDecl
   * and stored in lowerVars set. If collision, emit a violation
   */
  protected void findCollidingVar(LocalScope scope,
ASTLocalVariableDeclaration varDecl,
                                  Set<String> lowerVars, RuleContext ctx) {
    for(VariableNameDeclaration vn : scope.getVariableDeclarations().
keySet()) {
      AccessNode influencingDecl = vn.getAccessNodeParent();
      if(varDecl==influencingDecl || !(influencingDecl instanceof
ASTLocalVariableDeclaration)) continue;
      if(influencingDecl.getEndLine() > varDecl.getBeginLine()) continue;
      ASTVariableDeclaratorId id = vn.getDeclaratorId();
      String varname = id.getImage().toLowerCase();
      if(lowerVars.contains(varname)) {
        report(this, varDecl, ctx);
      }
    }
  }
}
```