Tips for programming static analysis Rules

Tips for programming static analysis Rules

Rule programming best practices

To simplify rule programming and produce more robust and maintainable programming rules, see the following best practice tips.

Use `BaseNode` interface whenever possible

Never use implementation node classes, like the generic `Node` interface or `SimpleNode` class generated by the parser generator (e.g. JavaCC). The reason is obvious: An AST tree may have nodes generated by different parsers (heterogeneous tree). `BaseNode` is the common denominator, all AST nodes implement this interface. If not, your rule may produce a `ClassCastException` when traversing AST nodes generated by the parsers for embedded languages (like SQL nodes in the COBOL AST tree, when EXEC SQL sentences are parsed). To avoid problems, and implement rules less dependent on potential grammar changes, it is advisable to use, when possible, the `BaseNode` interface.

If you need to check for the language the node belongs to, use `BaseNode.getLanguage()`. This is a simple way of determining which language the node belongs to.

Use common tree navigation facilities, instead of implementing your own "utility methods" for traversing AST trees.

The `TreeNode` decorator is a powerful AST tree navigation helper. You can traverse trees rooted at a certain `BaseNode` using a `NodeVisitor` (recursively or traversing the node and its immediate children), fetch for nodes that match a certain condition (using `NodePredicate`), or traverse the siblings of a certain node. See the javadoc for `BaseTree` for further information.

You may add utility methods if you need certain reusable extraction operations, but in this case, use the `TreeNode` facilities to implement such methods, avoiding usage of the specific AST implementation classes, as mentioned in the previous tip.

Never add "artificial state" to your rule classes.

Rules should be, if possible, "stateless", with no side-effects. Never add instance variables (like a node list) that depend on the nodes of the currently processed file. If you do not obey this simple rule, your rule must clean-up such artificial state before processing the next source input, increasing memory footprint and opening the opportunity for all kinds of errors (when a list of nodes is not cleaned-up, the rule will see in the list nodes belonging to previous input sources, producing bugs difficult to resolve).

Exceptions to this rule are:

- **Configuration state** (for rules that use complex expressions, lists of accepted/rejected tokens, and so on). Rules may override the `initialize()` method, that will be invoked by the framework at analysis startup, when the rule should prepare complex configuration items (like pre-compiling regular expressions to be used in the `visit()` method). Remember that, for simple properties, this is not necessary (`getProperty()` methods provide access to rule parameters). For example, a numeric threshold does not need to be an instance variable.
- **Global state** (for global rules). Global rules are rules that will first visit all input sources, compiling a global model that will be checked for when all input sources are processed. The rule here needs to remember a model of all input sources, before looking in that model for certain conditions. A dependencies rule that checks for dependencies between input sources is a typical rule of this kind: The `visit()` method implement the model building phase, and the `postProcess()` rule method will be used to check for the intended conditions in the model. But remember that local rules (rules that inspect a single input source at a time) are the most usual pattern for static analysis.

Emit violations judiciously

For naming/format convention rules, when a line does not match the convention, a `RuleViolation` could be emitted. The problem with this approach is that it could produce a rule violations storm, when hundreds or thousands of violations are registered in the violations report, making the vision on the most important issues to fix first more difficult. So it is better to decide before if the more conservative approach of emitting a single violation per file, to tell programmers which files do not follow the convention, could fit the bill avoiding the violations storm.

One violation per rule or single violation per file?

It depends on the importance of the convention. Obviously emitting a violation for all lines tells the programmer where each violation can be found, but at the cost of contaminating both the violations report and the confidence factors, hiding other rules (like potential bugs) that should be visible. So as a trade-off, for such convention-checking rules, the single violation per file approach is better.

Avoid plain-text checks

The purpose of the AST is to present a source code structure, in a way that simplifies structural checks (as the AST eliminates issues like the exact source code format). So if the AST contains the needed information for the rule revision, use it.

Obviously, there are certain kind of rules that do need access to the source code: most parsers discard comments (comments are not seen in the AST) and, due to grammar limitations, even essential source code is not available in the image field of the AST nodes. When this happens, use the `FileContents` int erface as follows:

```
FileContents contents = ctx.getOriginalFileContents(); // Or ctx.
getTransformedFileContents()
    String code = contents.getContent(); // To process all code content
    String[] codeLines = contents.getLines(); // To process all lines
    String line = contents.getLine( node.getBeginLine()-1 ); // Direct
access to the code line for node
    ... put your checks on code, codeLines or line here ...
```

When you find yourself creating complicated regular expressions to fetch code elements. Think twice if the AST tree provides the needed information.

The static analysis framework provides a `FileContents` bean that gives direct access to source code contents/lines, without the need to re-read the input file (and leave open file descriptors around).

Using `FileContents` for processing source code text is safer (no `IOException` possible), more efficient (as the input files are read only once), and more maintainable (as your rule "business" code is not polluted with file I/O).

Try to limit dependencies with the underlying grammar

If your rule code is coupled too much with the grammar artifacts (node classes, node and token images, node hierarchies for language constructs, etc.), then changes in the grammar and generated parser will produce AST trees that will break your rules. Of course, this is easy to say, but very difficult to follow. Rules usually need to locate nodes that match grammar rules. But try to limit such dependencies on underlying grammar as much as possible.

Rule programming anti-patterns

The next section explains some anti-patterns, that you could follow if you want your rules to be unmaintainable, break for difficult-to-track causes, and consume lots of memory and lead to an `OutOfMe mory` error.

Add artificial state to your rules (in the form of instance variables)

As said before, local rules should not have side-effects between different source code inputs. Never use `R uleContext` as an instance parameter, pass it as method parameter from the `visit()` method to the helper methods that need it. And never collect (in local rules) lists of AST nodes: use local variables instead (that will be garbage collected quickly when the variable is out of scope, without the need to clear collections at the end of the visit method), and pass them to the utility method. Or better, use AST navigation facilities provided by `TreeNode`, to process nodes of interest locally, and avoid creating node collections and iterating over such collections later. Your code will be crystal-clear.

In the past, we have seen many rule programmers following this anti-pattern, copying&pasting many code blocks that use unnecessary instance variables. This anti-pattern should be never followed (but remember that rule configuration and global model data are exceptions to the avoid artificial state rule).

Read source code files liberally

Once again, never read source files. Use `FileContents` to access source code contents in the easiest way.

If every rule of N reads the file, and you have M files, your I/O rate will be N*M (N times more than the M rate that the static analyzer itself needs).

If you have 1,000 rules and 1,000 input sources, you are reading files 1,000,000 times.

Your static analysis performance will crawl and take hours to complete. Your rules are intelligent, so to demonstrate how important they are, read files liberally, to demonstrate how much complexity your brain can manage.

Put nonsense comments in your code, but refuse to document the rule purpose and how the condition is checked.

Some things that do not add value to your code are:

- Useless code comments, like single-line comments at the end of a method or class,
- @param javadoc entries for obvious parameters,
- Or trivial documents to a trivial line of rule source code.

If your rule purpose is not clearly documented in JavaDoc comments, and complex checks are not documented appropriately, your rule will not be maintainable. Make sure to document exceptions, put sample code with the kind of conditions detected by the rule, and so on.