Source code analysis with SonarQube

Gilles QUERRET ● Riverside Software

What's source code analysis?

- Part of QA
- Extracts metrics from source code
  - LOC, NLOC, …
- Reports violations against various types of rules
  - Coding standards
  - Potential bugs
  - …

Why would you want to analyze source code?

- Refactor source code
- Performance problems
- Portability problems
- Coding standards

Existing tools for OpenEdge

- Limited number of tools :
  - Proparse analyses source code
  - Prolint detects pattern in Proparse output, and store the result
- Prolint output is quite minimalist

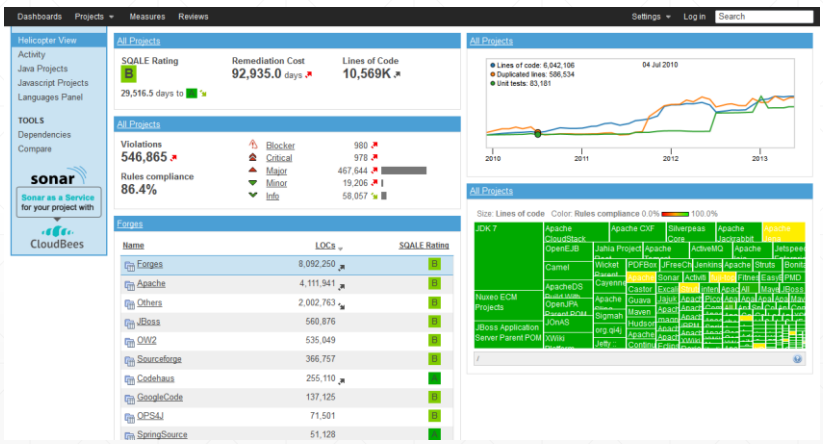
Compared to what's available in Java

- Bytecode analyzer : ObjectWeb ASM, …
- Static code analysis : FindBugs, Checkstyle, PMD, …
- Code coverage : JaCoCo, …

- And this is only for the opensource part.
- Many commercial products

And SonarQube to bind them all !

- SonarSource gather the results in a single database, and display them in a very nice web UI.



How does it work ?

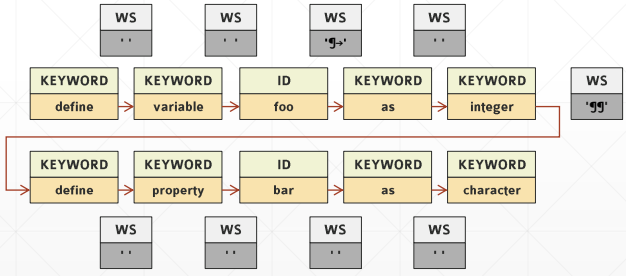
- Static code analysis usually rely on Lexer and Parsers
- Lexer converts a sequence of characters into a sequence of tokens
- Parser converts a sequence of tokens into a syntax tree

- Let's see a simple example :

Lexer

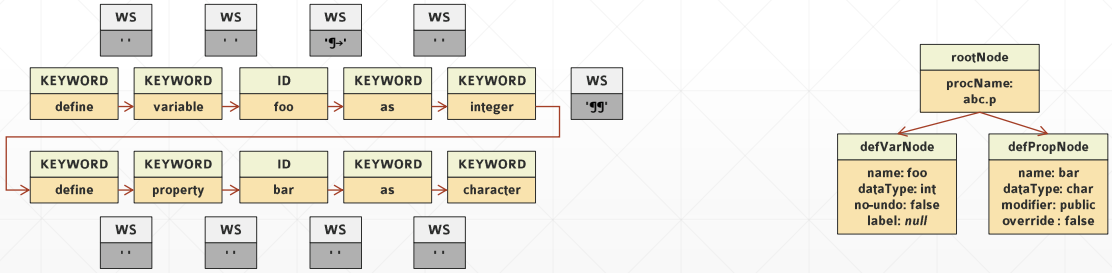
- `KEYWORD : 'define' | 'variable' | 'property' | 'as' | 'character' | 'integer'`
- `WS : (' ' | '\t' | '\n')* -> HIDDEN`
- `ID : [a-zA-z]*`
- `EOS : '.'`

```
define variable foo¶
 → as integer.¶
¶
define property bar as character.
```

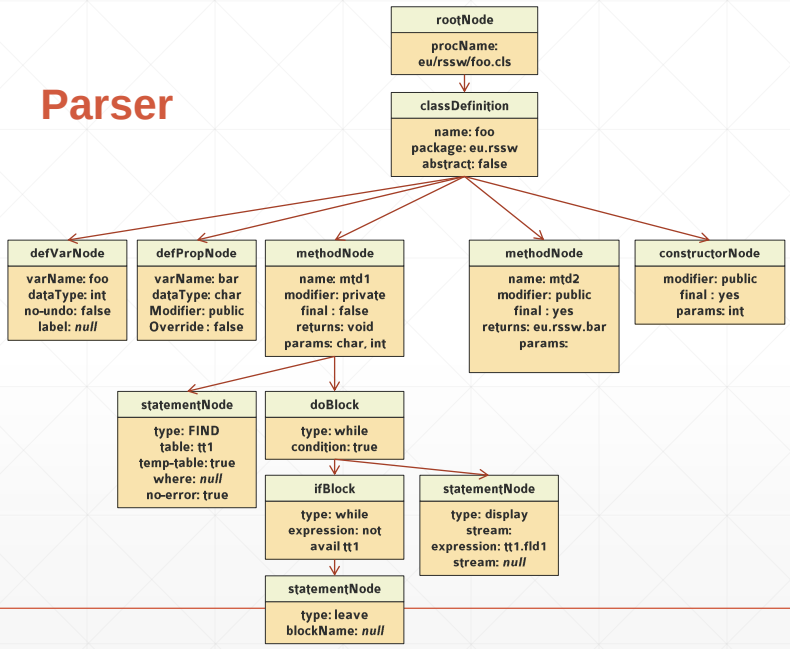


Parser

- defVar: KW_DEFINE KW_VARIABLE ID KW_AS DATATYPE KW_NO_UNDO? (KW_LABEL QUOTED_STRING)?
- defProp: KW_DEFINE (KW_PUBLIC | KW_PROTECTED | KW_PRIVATE)?
  (KW_STATIC | KW_ABSTRACT)? KW_OVERRIDE? KW_PROPERTY ID
  KW_AS (DATATYPE | CLASSNAME)



Parser



Only parsing source code ?

- Parsing can be done :
  - On profiler output (data structures for execution times, code coverage, …)
  - On DF files to know the database structure
  - On XREF files

OpenEdge extension for SonarSource

- SonarRunner is the batch process to analyse local source code, and push the results to the remote SonarSource instance
- An extension is made of several « sensors », each sensor analyzing a specific set of data
- Current sensors :
  - Rcode sensor
  - Dump files sensor
  - Code duplication sensor
  - Code coverage sensor
  - Source code sensor

Rcode sensor

- Extract information from rcode, and generate metrics :
  - Coupling between classes
  - Circular references between packages
  - Number of public / private methods
  - Common OO violations (using variables in classes, constructor for utility classes, …)

Dump file sensor

- Extract information from DF files:
  - Allow working with a known database structure when analyzing source code
  - Violations on table/field/index names, useless indexes, common SQL problems

Code duplication sensor

- Copy / paste is widely used during software development
- High code duplication means you should refactor your code
- Duplicated code usually mean that a fix is not applied everywhere

- Doesn’t do the refactor for you !

Code coverage sensor

- Use profiler output to exactly know which lines of code have been tested
- Multiple profiler files can be parsed and aggregated
- Let you know exactly which part of your code is being tested (or not tested)

Source code sensor

- Generate syntax tree from every class / procedure
- Then execute a set of lint rules, and report violations
- Source code uploaded to SonarSource server in order to display it with annotations
- Code is displayed with syntax highlighting

Writing your own rules

- Lint rules will have access either to the source tree, or to a fully built object
- The source tree know the exact position of your tokens
- Sonar entry point to report violations

- Rules have to be written in Java
- Implementation time is nothing compared to design time !

Lint rule example

Verification of variable name length :

```
@Rule(priority = Priority.INFO, name = "Short variable names", description = “Verifies that
variable names are at least X characters.")
@BelongsToProfile(title = LintList.SONAR_WAY_PROFILE, priority = Priority.MINOR)
public class ShortVarNamesRule extends AbstractLintRule {
  private static final int DEFAULT_MINIMUM_LENGTH = 2;

  @RuleProperty(key = "minimumLength", defaultValue = "" + DEFAULT_MINIMUM_LENGTH)
  public int minimumLength = DEFAULT_MINIMUM_LENGTH;

  public boolean visit(VariableDeclarationStatement decl) {
    if (decl.getName().length() <= minimumLength)
      reportViolation(getFirstLine(decl), decl.getName());

    return true;
  }
}
```

Developer Studio integration

- The Sonar database can be connected from Developer Studio
- Violations are displayed in the standard editor
- Lint rules can be executed locally

Demo

Availability & licensing

- The OpenEdge plugin for Sonar will be available at the beginning of 2014
- Core functionalities will be available as open-source software (i.e. lexer/parser, connectivity with SonarQube, code duplication, transaction scopes)
- Rules will be available as commercial plugins
  - Different packages will be available (legacy code, OO code, …)

Questions ?

Reference ?

- Sonar Source : http://www.sonarsource.com
- Sonar OE plugin demo site : http://sonar.riverside-software.fr

- Riverside Software : http://riverside-software.fr
- Contact : contact@riverside-software.fr