Static Code Analysis

PV260 Software Quality

Stanislav Chren, Vaclav Hala

5. 5. 2020

Overview

> *Static program analysis is the analysis of computer software that is performed without actually executing programs. In most cases the analysis is performed on some version of the source code, and in the other cases, some form of the object code. The term is usually applied to the analysis performed by an automated tool.*

Things to keep in mind:

- The tools can usually not find all the problems in the system (False Negatives)
- Not all of the detections are real problems (False Positives)
- There is no one tool which could be used for everything, more tools are usually used in parallel (Overlaps will occur)
- We must know what we are looking for, using everything the tool offers usually results in too much useless data

Source Code Analysis - Checkstyle

http://checkstyle.sourceforge.io/

> *Checkstyle can check many aspects of your source code. It can find class design problems, method desig problems. It also have ability to check code layout and formatting issues.*

PROS:

- Can be used to check programming style
- Many premade checks available
- New checks are easy to add

CONS:

- Type information is lost making some checks impossible
- Files are checked sequentially, content of all files can't be seen at once

Available standard checks:

`http://checkstyle.sourceforge.net/checks.html`

Checkstyle - Running via Command Line

- Download checkstyle `https://github.com/checkstyle/checkstyle/releases/` , select the *checkstyle-VERSION-all.jar* variant
- Go to the folder where you have placed the checkstyle.jar
- Run the command `java -jar checkstyle-VERSION-all.jar` with the following options:
  - `-c` (configuration file) path to .xml configuration for the audit (there are few premade in the .jar file, for now let's use the sun_checks.xml, extract it)
  - `-f` (format) defaults to plain which we want, so no need to set. XML output is available using `'-f xml'`
  - `-o` (output file) if not provided output goes directly to console. This is impractical for longer audits, so use e.g. `'-o out.txt'`
  - lastly provide path to file you wish to analyze. If directory is provided, its contents are analyzed recursively

All paths are relative to working directory when running the command

Checkstyle - Configuring which checks to use

The whole configuration file knows two types of things

- Modules (backed by java classes) - each is either parent maintaining other modules or performs one specific check
- Property (backed by setter methods on parent modules) - any module can have user configurable properties which can be set through the nested Property element

There are two basic types of standard check modules:

- Children of the Checker module - these work either with whole files or with raw unparsed text inside these files
- Children of TreeWalker module - these work with parsed AST (Abstract Syntax Tree) of .java files

It is important to put each check under the parent which supports it. For every check its intended parent is at the bottom of its details in the checks documentation.

Task 1

Create a check configuration which will look for the following:

- Boolean expressions with 4 or more operators
- Methods with cyclomatic complexity over 10
- Classes which override equals but not hashcode
- Files longer than 1000 lines
- Files which contain the //TODO or //FIXME comments
- Violations of the naming convention for constants

Checkstyle - Inspecting the AST

When writing own checks it helps to see the parsed AST tree to which checkstyle converts the sources when running analysis. Graphical tool for inspecting the tree is provided in the checkstyle-VERSION-all.jar file. To use it:

- Go to the directory with the checkstyle-VERSION-all.jar
- From command line run `'java -cp checkstyle-VERSION-all.jar com.puppycrawl.tools.checkstyle.gui.Main'` (It is important to use the `-cp`, if `-jar` were used the main class from the .jar's manifest would be used, which is the analysis starting main, and the path to main class we wish to use would be interpreted as a file to analyze. Checkstyle would then exit with config file not specified error)
- In the window which pops up click 'Select Java File' in the lower left and select a file to parse

Checkstyle - AST Tree GUI Explained

Browse the tree by expanding (double click or click the latch left of names) nodes with folder icon. In the AST browser window you can find the following:

- Tree: parsed hierarchy of the whole file, naming format is TYPE[LINExCOLUMN]
- Type: identifier of the node, these names are also used when writing the checks
- Line, Column: Starting line and column of the node, ending of the node can be found by looking at the start of the sibling node(there are cases when this method would fail though)
- Text: If the node has some meaningful text associated with it from the source (e.g. IDENT contains the name) it is written here. If no such text exists type of the node is used
- Source code of the parsed file is in the bottom window

Checkstyle - How the TreeWalker Analysis Works

The basic idea of the analysis is following:

- First all the registered check are mapped from token Type to checks (that is, each check lets the walker know in which types of tokens it is interested in)
- For every file, the AST is obtained by parsing the file
- The walker goes through the tree top to bottom and in every node sends event to all checks which are interested in the particular node type
- It is responsibility of every check to keep any state/data it needs to perform its task, at any time during the analysis if the check decides it has a detection it reports it

Checkstyle - Writing a Check

- When writing checks, you need to have the checkstyle on your classpath as dependency
- Each check for the TreeWalker must extend the class AbstractCheck
- Every check must implement the getDefaultTokens(), this method tells the walker what token types the check is interested in
- To implement what happens on token visit, implement the visitToken(token) method. The argument you receive is the token just visited
- Action can also be taken when leaving the node (after all its children have been processed), implement the leaveToken(token) if you wish to receive this event
- To report your findings use one of the log() methods

Checkstyle - Workflow for Writing Checks

- Create some test code which contains the defects you wish to check for
- View this code in the AST viewer GUI to see how the parsed code will look like during the analysis
- Find patterns you can detect in the parsed AST
- Write some implementation of your check and run it against the test code
- Repeat until all valid cases pass and bad inputs fail

Unit testing of checks is possible but setting up the environment for these tests is rather time consuming so we will not do it here

Checkstyle - Using Your New Check

To be able to use your new check do the following:

- Create a .jar file containing the new check
- Add the jar to checkstyle classpath:
  - For command line usage: `java -cp yourjar.jar;checkstyle-VERSION-all.jar com.puppycrawl.tools.checkstyle.Main` rest of the command is same as normal, so `-c` config and so on
  - (in unix OS, replace ”;” with ”:” in the command above)
- Now you can use names (full names including the packages) of your checks in the xml config file

Task 2

2a Write a custom check which will be able to detect a for cycle with negative increment

2b Write a check which will detect a for cycle that never executes because its condition always starts as false
(e.g. `for(int i=0; i>1; i++)`)