

SER 216
Team 6
Maigan Davey
Seiichi Nagai
Jeremy Pasimio

Iteration 3 Testing tools

Sonar Platform

SonarSource provides a two part platform to aid in team management, code quality, and unit testing support.

- SonarQube is a service that provides code analysis and unit testing support for teams of developers. Code analysis includes features such as bug checking, code coverage analysis, and security vulnerability analysis. Code analysis through SonarQube is facilitated through separate code analysis utilities which are also available from their website.
- SonarLint is an IDE extension that provides real time alerts regarding current and potential errors, as well as alerts regarding quality errors such as following of conventions and reducing cognitive complexity.

Both products are free to use and are available from the company's website:

www.sonarsource.com. The free to use version is, obviously, not as robust as the paid versions of the product. There are three paid tiers available, each offering more features, support for more languages, and pricing based on lines of code being analyzed. Additionally, SonarSource offers an online version of their code analysis tools called SonarCloud. The service is freely available for open source projects and paid subscription is available for private projects. Pricing for SonarCloud is, again, based on number of LOC.

The sonar platform supports Unit Testing via SonarQube by providing at a glance statistics regarding code coverage, percentage of passing tests, and opportunities for refactoring to increase code coverage and overall code quality. It should be noted that SonarQube will not facilitate writing unit tests but will only provide statistical analysis of test status and results to help determine next steps in development. Unfortunately, many of these features are not available in the free version and so were not thoroughly analyzed. Information and screenshots of these features are available at:

<https://blog.sonarsource.com/sonar-to-manage-unit-tests-and-improve-code-coverage/>

SonarQube also provides an interesting team management feature called Quality Gates. These gates can be defined by the project lead and help ensure that all team members are developing to the same quality standards. At-a-glance reporting allows for an easy view of which modules

are passing the quality gates and which are not, as well as showing changes in a modules passing status (i.e. module passed quality gate until it was refactored and now does not pass).

Code Quality can be analyzed in real time with SonarLint. As an IDE extension, it is very simple to install and use. For Eclipse it is available directly from the Eclipse Marketplace and is set to be active by default with no additional configuration once installed. The benefit of this extension can be found in the ability to define custom “code rules.” There are many rules built into the extension based on the language being used and the team leader or project manager can define additional rules to make sure the same coding conventions are being followed by each team member. Some of the code checking seems redundant to what is already built into Eclipse, but SonarLint also highlights potential errors and code that could be hard to maintain while Eclipse really only catches syntax errors that will not allow for a successful compile.

The Sonar platform is an open source tool, written in Java to support analyzation of more than 20+ programming languages including Java, C/C++, Swift, Python, and many more. A full listing of supported languages is available on the website. Having Java installed on the development machine is a requirement for use. The number of languages supported is dependant on the tier of subscription being used.

Installation of SonarQube is fairly simple. Zip archives for both SonarQube and a code analyzer are simply downloaded and extracted to the directory of choice. After extraction, links to the appropriate directories are added to the Path variable to allow access to the dashboard and analyzation commands. Actual analyzation of the code is run from the command line after creating and defining a properties file in the base directory of the project. Once analyzation is complete all analyzed projects are available for viewing from the dashboard which is simply run in a browser window. As stated before, SonarLint is just an IDE extension which (In the case of Eclipse) can be installed directly from the IDE.

This tool's main strength is its team management aspect. By defining quality gates and code rules it allows for leaders on a project to set conventions to be followed by the whole team and allows visibility at-a-glance to where conventions are not being followed. It also serves as a great support to unit testing by showing data regarding passing vs failing tests, code coverage, and branch coverage.

Sources

<https://www.sonarqube.org/>

<https://docs.sonarqube.org/display/SONAR/Analyzing+Source+Code>

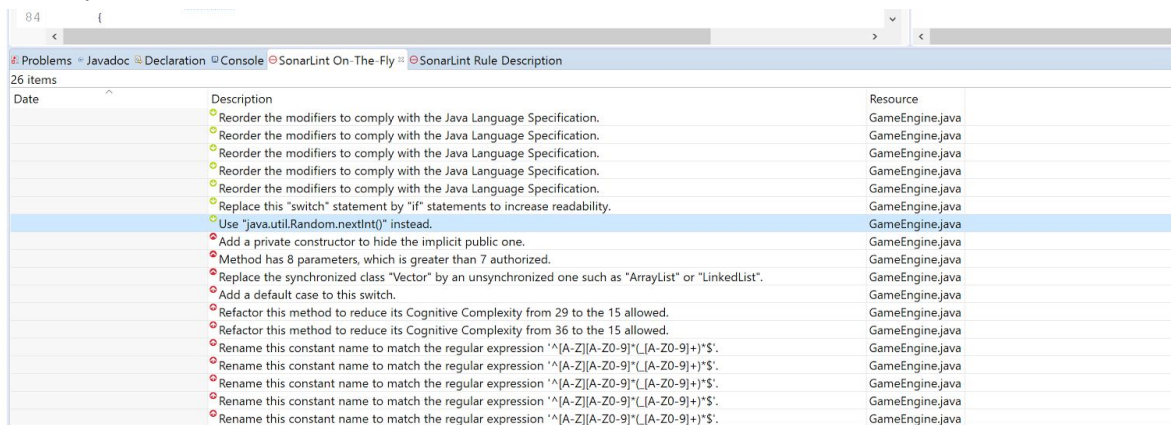
SonarQube Project Dashboard

The screenshot displays the SonarQube Project Dashboard. The left sidebar contains filters for Quality Gate (Passed: 3, Warning: 0, Failed: 0), Reliability (Bugs), Security (Vulnerabilities), Maintainability (Code Smells), and Coverage. The main area shows three projects: 'test-project', 'test:project', and 'test:Project', all with a 'Passed' status. Each project summary includes counts for Bugs, Vulnerabilities, Code Smells, Coverage, and Duplications. A warning message at the bottom states: 'Embedded database should be used for evaluation purpose only. The embedded database will not scale, it will not support upgrading to newer versions of SonarQube, and there is no support for migrating your data out of it into a different database engine.'

SonarQube Code Rule Example

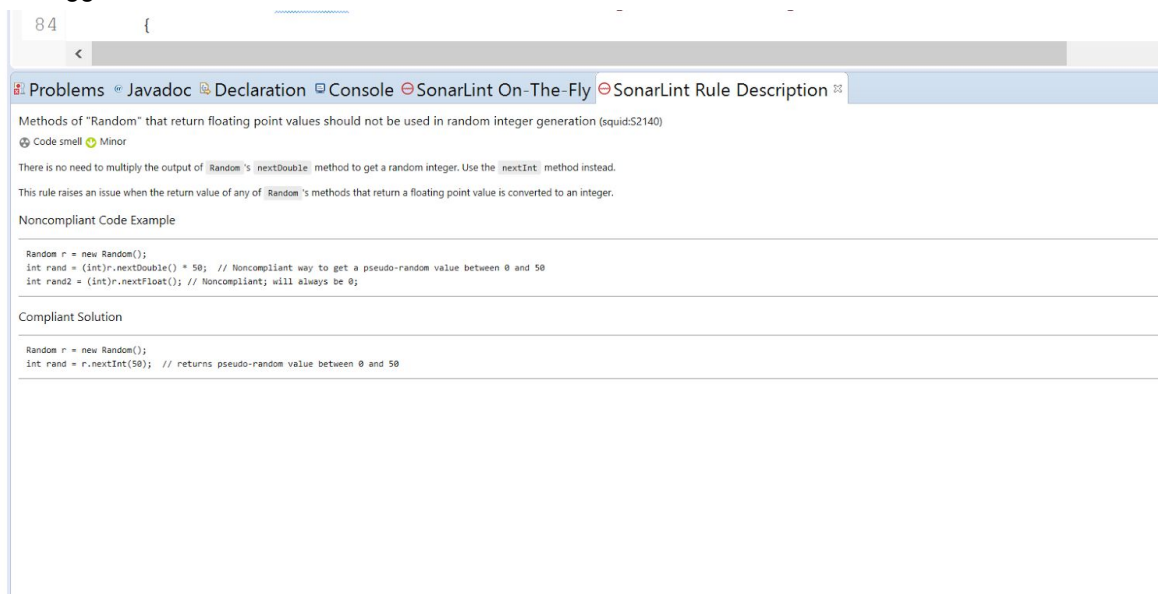
The screenshot shows the SonarQube Code Rule Example page. The left sidebar lists filters for Language (Java: 440, C#: 307, Python: 240, JavaScript: 189, PHP: 127, Flex: 79, TypeScript: 79, XML: 13) and Type (Bug: 104, Vulnerability: 33, Code Smell: 303). The main area displays the rule 'SQUID:52204' titled '"equals()" should not be used to test the values of "Atomic" classes'. It includes a description, a noncompliant code example, and a compliant solution. A warning message at the bottom states: 'Embedded database should be used for evaluation purpose only. The embedded database will not scale, it will not support upgrading to newer versions of SonarQube, and there is no support for migrating your data out of it into a different database engine.'

SonarLint Eclipse View



Date	Description	Resource
	Reorder the modifiers to comply with the Java Language Specification.	GameEngine.java
	Reorder the modifiers to comply with the Java Language Specification.	GameEngine.java
	Reorder the modifiers to comply with the Java Language Specification.	GameEngine.java
	Reorder the modifiers to comply with the Java Language Specification.	GameEngine.java
	Reorder the modifiers to comply with the Java Language Specification.	GameEngine.java
	Replace this "switch" statement by "if" statements to increase readability.	GameEngine.java
	Use "java.util.Random.nextInt()" instead.	GameEngine.java
	Add a private constructor to hide the implicit public one.	GameEngine.java
	Method has 8 parameters, which is greater than 7 authorized.	GameEngine.java
	Replace the synchronized class "Vector" by an unsynchronized one such as "ArrayList" or "LinkedList".	GameEngine.java
	Add a default case to this switch.	GameEngine.java
	Refactor this method to reduce its Cognitive Complexity from 29 to the 15 allowed.	GameEngine.java
	Refactor this method to reduce its Cognitive Complexity from 36 to the 15 allowed.	GameEngine.java
	Rename this constant name to match the regular expression '^A-Z[A-Z0-9]*([A-Z0-9]+)?\$'.	GameEngine.java
	Rename this constant name to match the regular expression '^A-Z[A-Z0-9]*([A-Z0-9]+)?\$'.	GameEngine.java
	Rename this constant name to match the regular expression '^A-Z[A-Z0-9]*([A-Z0-9]+)?\$'.	GameEngine.java
	Rename this constant name to match the regular expression '^A-Z[A-Z0-9]*([A-Z0-9]+)?\$'.	GameEngine.java
	Rename this constant name to match the regular expression '^A-Z[A-Z0-9]*([A-Z0-9]+)?\$'.	GameEngine.java

SonarLint Flagged Item Detail



84 {

Problems Javadoc Declaration Console SonarLint On-The-Fly SonarLint Rule Description

Methods of "Random" that return floating point values should not be used in random integer generation (squid:S2140)

Code smell Minor

There is no need to multiply the output of `Random`'s `nextDouble` method to get a random integer. Use the `nextInt` method instead.

This rule raises an issue when the return value of any of `Random`'s methods that return a floating point value is converted to an integer.

Noncompliant Code Example

```
Random r = new Random();
int rand = (int)r.nextDouble() * 50; // Noncompliant way to get a pseudo-random value between 0 and 50
int rand2 = (int)r.nextFloat(); // Noncompliant; will always be 0;
```

Compliant Solution

```
Random r = new Random();
int rand = r.nextInt(50); // returns pseudo-random value between 0 and 50
```

JHawk – Java Code Metrics

JHawk is a static code analysis tool developed by Visual Machinery. Commercial licenses start at \$30 for the Starter license and increase up to \$900 for the Corporate license. Other licenses available are: Personal (\$75), Professional (\$145) and Site (\$500). Depending on the license certain features such as Eclipse Plugin, Metric Creation, Command Line Jar, etc. become available. All licenses include the standalone application and import/export features. Academic licenses are available for \$99. The academic license is equivalent in features to the Professional license and may be used by all researchers within a single research department. All licenses and downloads are available at:

<http://www.virtualmachinery.com/jhawkprod.htm>

The JHawk Java Code Metrics tool provides automated, objective software measurements using static code analysis to assist in quality control. The tool reads source code from java files and determines numerous metrics to measure software quality. These metrics can be computed manually but can become increasingly complex and time-consuming depending on the size of the system. In addition, code reviews often involve the use of the most valued and expensive resources; the most experienced programmers. (1) Furthermore, the range of metrics allow users to pinpoint code that is more likely to have errors. This can help code reviews be directed toward problem areas to make sure that susceptible code is inspected thoroughly. (1)

Metrics are split into three levels: Method, Class and Package/System. At the Method Level, code metrics include:

- **Number of Java Statements** (instead of LOC). This allows for more freedom in syntax; added LOC for bracket, etc. (2)
- **McCabe's Cyclomatic Complexity** – “measurement of number of possible alternative paths through a piece of code”. A score less than 10 is ideal (2)
- Several **Halstead Metrics** – Length: the number of operators & operands. Vocabulary: complexity in statements. Volume: amount of code written. Difficulty: how difficult is code to write and maintain? Effort: amount of work to recode a method. Bus: the number of bugs liable to be in a particular piece of code. (2)
- **Maintainability Index**: high, medium or low degree of difficulty to maintain (2)
- Other Method Level metrics include: **Number of Arguments** in method headers, **Number of Comments**, **Variables Declared**, **Variables Referenced**, **Number of Expressions**, **Number of Loops**, **Max Nesting**, **Number of External Methods Called** and **Number of Classes Referenced**

At the Class Level, code metrics include:

- **Total**, **Average** and **Max Cyclomatic Complexity**, **Total Halstead Effort** and **Maintainability Index** (3)

- As well as **Lack of Cohesion of Methods** (LCOM) – measures correlation between methods and local instance variables of a class (3)
- **Unweighted Class Size** (UWCS) – number of methods plus number of attributes belonging a class. Less than 100 is ideal (3)
- **Response For Class** (RFC) – measures complexity of class in terms of method calls (3)
- **Message Passing Coupling** (MPC) - number of messages passed among objects of the class. This includes: **Coupling Between Objects, Fan Out, Fan In, Efferent Coupling, Afferent Coupling.** (3)

Finally, code metrics at the Package & System Level include:

- **Instability** – how susceptible package is to change (0-1) 0 is best
- **Fan In/ Afferent Coupling**
- **Fan Out/ Efferent Coupling**
- **Abstractness** – on a scale of 0-1. 0 is concrete, 1 is abstract.
- **Distance** – balance of particular package between abstraction and instability
- **Total/Average Cyclomatic Complexity, Halstead Effort, Maintainability Index**
- In terms of Performance, the user can rank methods in order of complexity to determine which methods will take the most time to execute.

JHawk improves productivity by showing where bugs may be likely to occur, preventing coders from writing complex methods (the numbers will eventually show that the methods are inefficient or bug prone), searching for the specified metrics automatically rather than by utilizing a programmer and by providing objective measurements.

This tool is useful in Unit Testing, Integration Testing, System/Functional Testing and Regression Testing. Quality Control can be measured at any level of testing, i.e. individual units (methods), overall classes, calls between classes (Integration), package and system (system testing). The results may also cause the programmer to want to refactor existing code, therefore the tool may be involved with regression testing. Static code analysis inherently involves looking inside the code, but the system metrics can be used in conjunction with Functional testing.

In order to use JHawk, you must have Java installed on your computer. In addition, you need to either purchase a license and download the tool, or you may download a free demo version of the product (all of which is available at the VirtualMachinery website). This tool is meant to analyze existing code, so you should have .java files written before using the program. The tool was created specifically for Java and therefore only supports the Java language. If you have the Personal License or greater, you may use the Eclipse plugin which has all of the same functionality but within the Eclipse IDE. The standalone comes ready to run. Simply run the jar file, select the .java files you wish to analyze and click 'Analyze'. When finished analyzing, the Results tab populates. Clicking on the Results tab shows further sub-tabs: Dashboard (Summary), System, Classes by package, Methods by class.

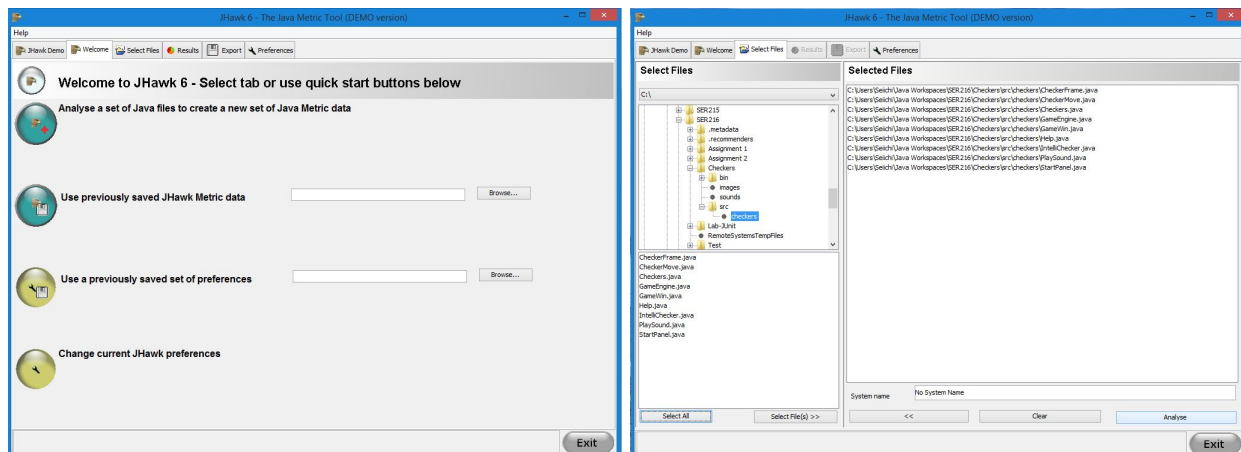
JHawk is a very user-friendly tool with easy set-up and a shallow learning curve. Full documentation and walkthroughs are also provided with download. Furthermore, the Eclipse integration allows Eclipse users to access metrics more quickly and smoothly. Another plus with this tool is that you can export results to HTML, CSV and XML formats. Additionally, the tool has a large variety of metrics to choose from and the user may even implement their own. The User can easily view results at different code levels: method, class, package and system. Finally, the JHawk tool itself has been thoroughly tested and all of the code modules in the JHawk distribution have values for Maintainability Index greater than 65. (2)

Despite all of the strengths the tool has a few weaknesses, the worst being that the Eclipse plugin is not supported for Eclipse versions 4.5 (Mars) and newer. The other more minor weakness is that the GUI has an outdated look and feel, as do the graphics.

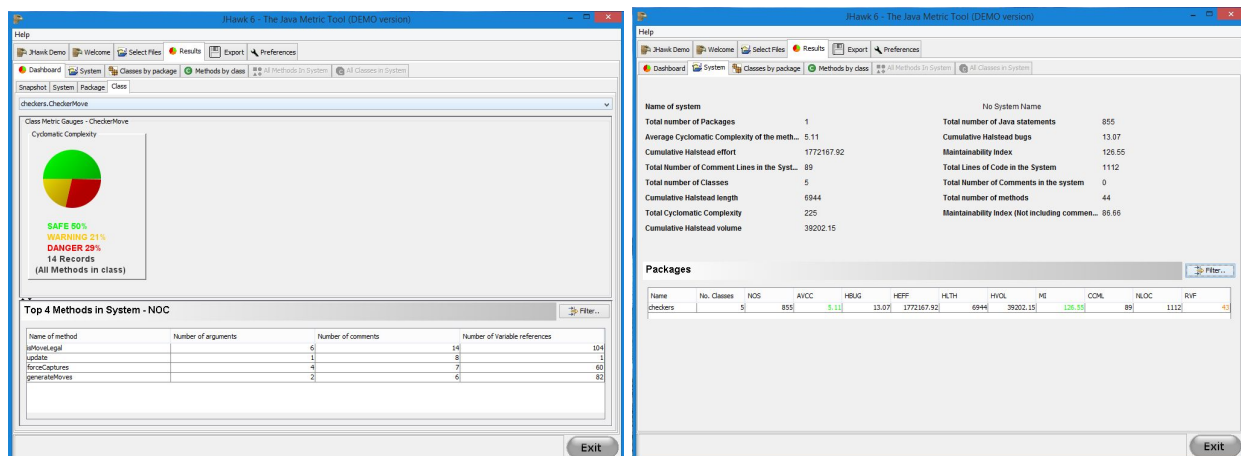
Overall, this is definitely a tool that would prove useful in quality assurance.

Below are some screenshots of the tool running (in the demo, only 5 classes can be analyzed at a time):

Welcome Screen (left), Select Files Screen (right):



Class View (left), System View (right):



Methods By Class View:

Methods

Name	COMP	NOCL	NOS	HLTH	HVOC	HEFF	HBUG	CREP	VMET	LMET	NLOC
ApplyMove(...)	7	8	16	194	43	32694.29	0.35	2	1	3	30
canCapture(...)	4	0	10	61	25	4448.78	0.59	0	0	2	10
canCapture(...)	30	4	27	767	36	762276.85	1.32	1	0	0	99
canWalk(int[] ...)	5	4	12	140	30	24387.25	0.23	1	0	1	20
colour(int[] ...)	2	1	9	46	18	1822.26	0.86	1	0	0	13
forceCapture(...)	7	3	28	252	51	67251.50	0.48	2	2	3	30
generateMove(...)	13	4	47	313	55	79103.87	0.60	3	4	6	48
getIndex(int....)	4	3	15	95	28	13396.50	0.15	0	0	0	14
isRange(int[] ...)	1	1	2	30	17	1226.24	0.34	0	0	0	4
isEmpty(int[] ...)	3	0	3	50	26	3153.21	0.08	1	0	0	7
isMoveLegal(...)	23	10	61	440	56	191642.71	0.85	2	1	2	82
isWalkLegal(...)	8	0	18	139	38	19955.47	0.24	2	1	1	26

checkers.CheckerMove

Exit

SOURCES

1. ROI -- <http://www.virtualmachinery.com/jhawkroi.htm>
2. Method Level Metrics-- <http://www.virtualmachinery.com/jhawkmetricsmethod.htm>
3. Class Level Metrics -- <http://www.virtualmachinery.com/jhawkmetricsclass.htm>
4. System Level Metrics -- <http://www.virtualmachinery.com/jhawkmetricssyspack.htm>