Usage Guide

**Sonar**

Table of Contents

[1 Introduction 3](#_Toc428877287)

[2 Launching a Sonar Analysis from a Maven Build 3](#_Toc428877288)

[2.1 Configuring Connectivity Parameters 4](#_Toc428877289)

[2.2 Configuring Analysis Parameters 5](#_Toc428877290)

[2.3 Launching the Analysis 7](#_Toc428877291)

[2.4 Launching the Analysis from Jenkins Using the Jenkins Sonar Plugin 8](#_Toc428877292)

[3 Sonar’s Project Dashboard 9](#_Toc428877293)

[4 Component and Metric Drilldowns 12](#_Toc428877294)

[5 Time Machine: Reviewing the Project History 13](#_Toc428877295)

[6 DSM Analysis 15](#_Toc428877296)

[7 Taking Decisions: Quick Wins, Top Risks and Hotspots 17](#_Toc428877297)

[7.1 Quick Wins 17](#_Toc428877298)

[7.2 Top Risks 18](#_Toc428877299)

[7.3 Hotspots 18](#_Toc428877300)

[8 Customising the Analysis 19](#_Toc428877301)

[8.1 Project Analysis Configuration 19](#_Toc428877302)

[8.2 Customising the Quality Profile 19](#_Toc428877303)

[9 Sonar analysis with Eclipse plugin 20](#_Toc428877304)

[10 References 22](#_Toc428877305)

[11 Document Control 23](#_Toc428877306)

[11.1 Change History 23](#_Toc428877307)

# Introduction

Sonar is a Continuous Quality Assurance engine that consolidates static and dynamic quality metrics leveraging known tools as Checkstyle, PMD, FindBugs, JUnit, Cobertura, JDepend or Architecture Rules. Sonar integrates with Maven builds or with Jenkins jobs, automatically running all the tools, gathering all the relevant data and collating them all in a single dashboard.

Sonar is the recommended tool in Java Blueprint 3 for adopting Consolidated Quality Profiling practices in the server-side. This document contains detailed insights on how to launch a Sonar analysis from a Maven build and setup how to use the Sonar dashboard to identify, prioritise and track quality issues. The companion document, **Sonar Installation and Setup Guide**, includes information for administrators on how to install and configure Sonar for first use plus some typical administrative tasks.

# Launching a Sonar Analysis from a Maven Build

Sonar analyses Java artefacts leveraging multiple static and dynamic code profilers; most of them actually recommended by the Java Blueprint as they are best-of-breed, widely used quality tools.

The version recommended by the Java Blueprint 3.2, Sonar 3.4.1, uses the following tools to analyse and extract valuable insights from Java projects:

* Checkstyle, the popular static code profiler that checks for adherence to coding and naming standards, proper Javadoc usage and best practices.
* PMD, another popular static code profiler that focuses in code compliance to best practices.
* PMD-CPD, the copy-paste detector module in PMD that identifies sections of code duplicated along the analysed sources.
* FindBugs, static code profiler that analyses Java bytecode in search of code patterns that are known as a source of bugs: security, performance, multi-threading…
* Sonar Squid: a metric collector tool based on popular JavaNCSS, JDepend or Architecture Rules.
* JUnit: the widely used unit testing framework.
* Cobertura: the popular code coverage tool.

Sonar analysis are launched from a Maven build using the sonar:sonar goal. The Maven Sonar plugin needs to know where the Sonar dashboard and the Sonar database are. Maven needs, therefore, direct connectivity with both the dashboard and the database.

## Configuring Connectivity Parameters

The configuration of connectivity parameters from Maven Sonar plugin to Sonar dashboard and Sonar database can be done by modifying the Maven configuration file (**settings.xml**, either global or local) or alternatively using command-line parameters. First option is recommended, as it only needs to be done once per Maven instance.

Moreover, modifying the Maven configuration file will not require modifying the Maven project model (**pom.xml**) and therefore we will not tie the project model to a concrete Sonar instance. It will also enable to run the same project model in different scenarios:

* Using a local Sonar instance.
* Testing a new Sonar version.
* Using a project-centric Sonar instance, or another instance.
* The client owns a Sonar instance where they analyse code from different providers.

Configuration itself needs to create a new Maven profile. For doing that, let’s edit the Maven **settings.xml** file and find the **<profiles>** section. Add a child **<profile>** section with the following structure:

<profiles>

…

<profile>

<id>sonar</id>

<activation>

<activeByDefault>true</activeByDefault>

</activation>

<properties>

<sonar.jdbc.url>database-url</sonar.jdbc.url>

<sonar.jdbc.driver>database-driver</sonar.jdbc.driver>

<sonar.jdbc.username>database-user</sonar.jdbc.username>

<sonar.jdbc.password>database-password</sonar.jdbc.password>

<sonar.host.url>dashboard-url</sonar.host.url>

</properties>

</profile>

…

</profiles>

Assuming that we are using an instance installed from the Java Build Accelerator or using the companion document **Sonar Installation and Setup Guide**, then the parameters would take these values:

* **sonar.jdbc.url** = jdbc:derby://localhost:3306/sonar?useUnicode=true&characterEncoding=utf8
* **sonar.jdbc.driver** = <driver name>
* **sonar.jdbc.username** = sonar
* **sonar.jdbc.password** = sonarsonar
* **sonar.host.url** = http://localhost:8080/sonar

You should replace **localhost** by the IP or name of the box where Sonar is installed to launch the Sonar analysis from a different box. You should also update the HTTP ports if you have not used the default ones during setup.

When using a Sonar local instance, the connectivity parameters are, by default, these:

* **sonar.jdbc.url** = jdbc:derby://localhost:1527/sonar;create=true
* **sonar.jdbc.driver** = org.apache.derby.jdbc.ClientDriver
* **sonar.jdbc.username** = sonar
* **sonar.jdbc.password** = sonar
* **sonar.host.url** = http://localhost:9000

In those cases where you need to override the profile configuration, for example to run a test analysis or to use a different instance just once, you may use command-line parameters:

mvn –Dsonar.jdbc.url=*database-url* –Dsonar.jdbc.driver=*database-driver*

–Dsonar.jdbc.username=*database-user* –Dsonar.jdbc.password=*database-password*

-Dsonar.host.url=*dashboard-url rest-of-Maven-command-line*

## Configuring Analysis Parameters

The Maven Sonar plugin allows configuring some analysis parameters using properties in the Maven project model (**pom.xml**) or through command-line parameters.

As these analysis parameters are related to the project, it makes perfect sense to include them as properties in the project model. To do that, just edit the project **pom.xml** file and add child nodes to the **<properties>** section:

<properties>

…

<sonar.dynamicAnalysis>value</sonar.dynamicAnalysis>

<sonar.reuseExistingRulesConfiguration>

value

</sonar.reuseExistingRulesConfiguration>

<sonar.squid.analyse.property.accessors>

value

</sonar.squid.analyse.property.accessors>

…

</properties>

Using the command-line, the syntax would be analogous to the following:

mvn –Dsonar.profile=*quality-profile* –Dsonar.projectDate=*YYYY-MM-DD*

*rest-of-Maven-command-line*

The most relevant analysis parameters are these:

* **sonar.importSources**: It is used to tell Sonar to not import sources, so they are not shown in the dashboard. This is useful for those projects where security reasons or client agreements doesn’t allow sources to get out of their premises.
* **sonar.dynamicAnalysis**: It is used to tell Sonar to reuse (when value is **reuseReports**) existing JUnit test execution reports and Cobertura code coverage reports, available from a previous Maven execution. It is also used (when value is **false**) to skip the dynamic analysis, useful when a Project does not provide automated test cases or in some integration scenarios out of the scope of this guide (Ant, command-line).
* **sonar.surefire.reportsPath**: It is used to tell Sonar the path where it can find test reports. It is used along **sonar.dynamicAnalysis**.
* **sonar.cobertura.reportsPath**: It is used to tell Sonar the path where it can find coverage reports. It is used along **sonar.dynamicAnalysis**.
* **sonar.projectDate**: It is used to force the analysis date, for example to launch the analysis of a previous version. It is useful for projects that are adopting Sonar after its actual start and wants to incorporate historic metrics to analyse the project evolution with time.
* **sonar.projectVersion**: It is used to force the version identifier, overriding the one found in the project model. It is used along **sonar.projectData** to execute analysis of old project codebase versions.
* **sonar.profile**: It is used to force the quality profile used in the analysis. By default, Sonar uses the quality profile configured in the dashboard or the default one if no profile is configured for the Project. This parameter is also used the first time an analysis is executed when we want to use a profile other than the default one.
* **sonar.exclusions**: It is used to force the exclusion pattern (those sources that we want to remove from the analysis, e.g. automatically generated classes, sources contributed by other providers). This parameters overrides the exclusions configured in the dashboard. The value is a list of patterns comma-separated. Each pattern is a regular expression telling Sonar what source to exclude from analysis.
* **sonar.skippedModules**: It is used to exclude some modules or packages from the analysis.
* **sonar.phase**: It is used to force the execution of a Maven phase or goal before the actual analysis takes place. It is useful when we have generated or instrumented sources and we want them to be part of the analysis scope.
* **sonar.squid.analyse.property.accesors**: It is used to tell Sonar Squid (when value is **false**) to ignore getters and setters in complexity calculations or in the method count. From a point of view getters and setters are overhead needed to work with the properties idiom, so they should not be taken into account for complexity or method count.
* **sonar**.**login**: It is used to tell Sonar the user account to use to publish the results. This will be the sonar users who are setup during Sonar configuration and allowed to publish sonar analysis results.
* **sonar**.**password**: the password for the Sonar user connecting to the remote Sonar server.

## Launching the Analysis

Once we have configured connectivity parameters in Maven **settings.xml** file and analysis parameters in the project model **pom.xml** file, we are ready to launch the analysis.

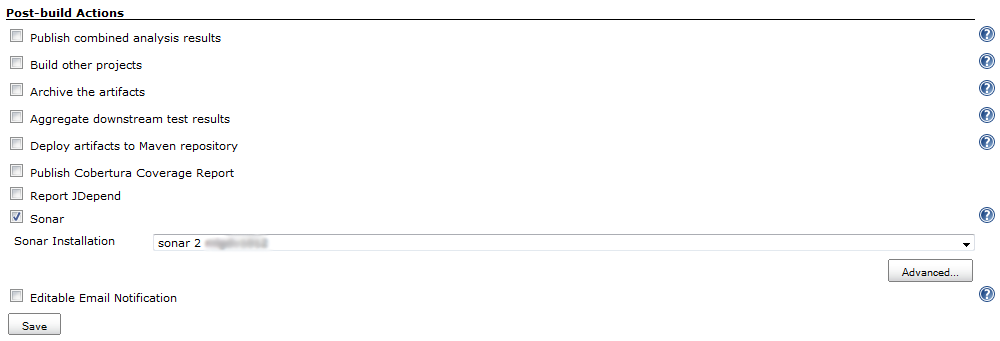
To do that, simply run Maven with sonar:sonar goal. Maven Sonar plugin will execute needed tasks (as compilation or unit tests) and then analyse the project, collect all metrics and send them to the Sonar database.

Once the process finishes you can go to the Sonar dashboard to review the analysis results.

## Launching the Analysis from Jenkins Using the Jenkins Sonar Plugin

Although we can launch a Sonar analysis any time, the most common and recommended to do that is from a Jenkins job, usually as part of a nightly job. A Sonar analysis may take around 5 minutes for a medium project or up to 20 for a large project (of course these estimations are highly dependent on the box where analysis takes place), so it is not recommended for continuous builds unless the project is critical and we need continuous insights on quality status.

To launch the Sonar analysis from Jenkins, the easiest way is to activate the Sonar post-build action to trigger the Sonar analysis at the end of the job:

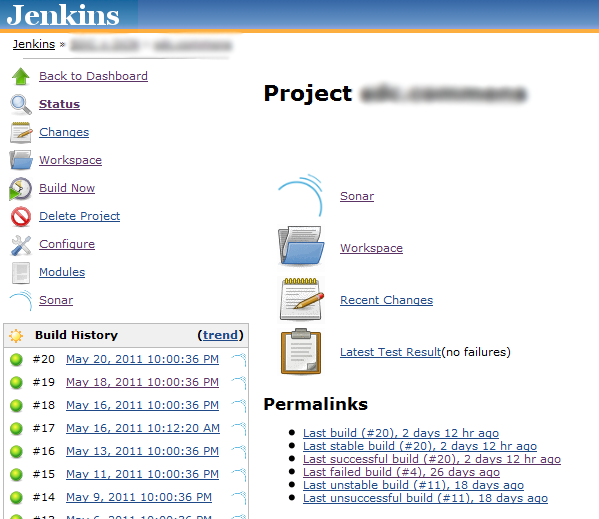
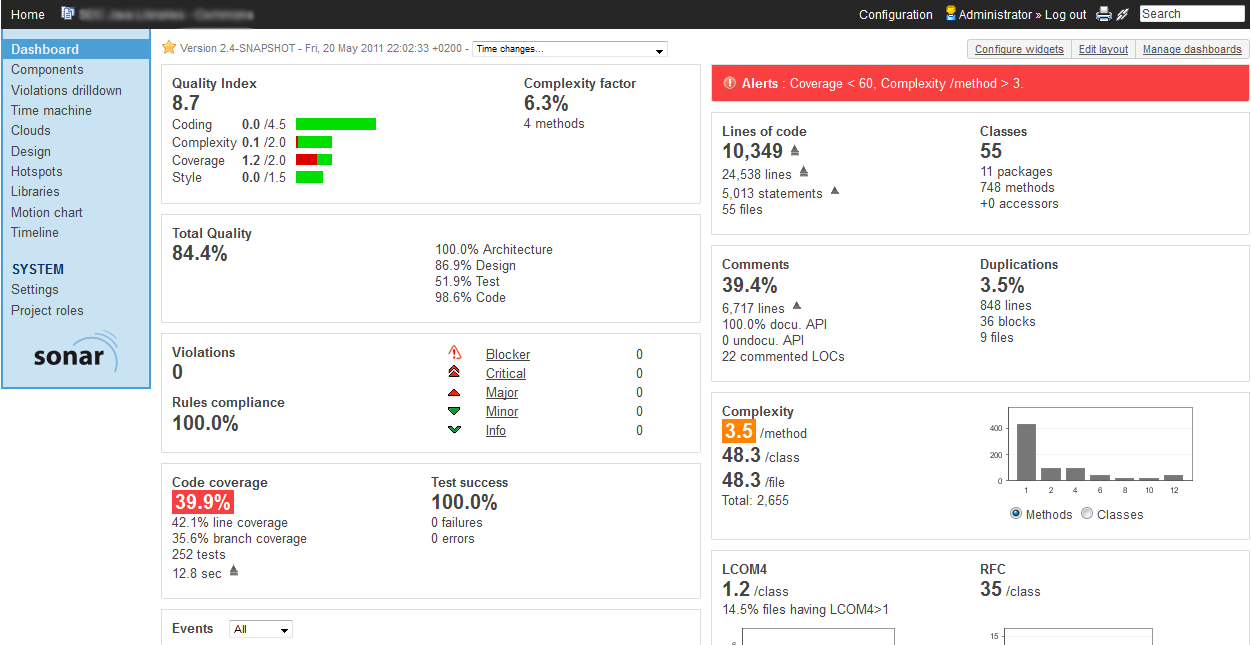


If more than one Sonar instances are configured in Jenkins, you may select the one that will be used for the analysis in a drop-down list below the Sonar check-box.

Please note that this post-build action only shows if the Jenkins Sonar plugin is installed and configured in your Jenkins instance. The companion documents Jenkins Installation and Setup Guide and Sonar Installation and Setup Guide contain information on how to install and configure the Jenkins Sonar plugin. Contact your Jenkins instance administrator to proceed with the installation. Note that the instance installed through the Java Build Accelerator already has the plugin installed and configured.

Once the post-build action is activated, the Sonar analysis will run after the build. Note that you do not need to select the sonar:sonar goal in the job configuration, as the post-build action takes care of that.

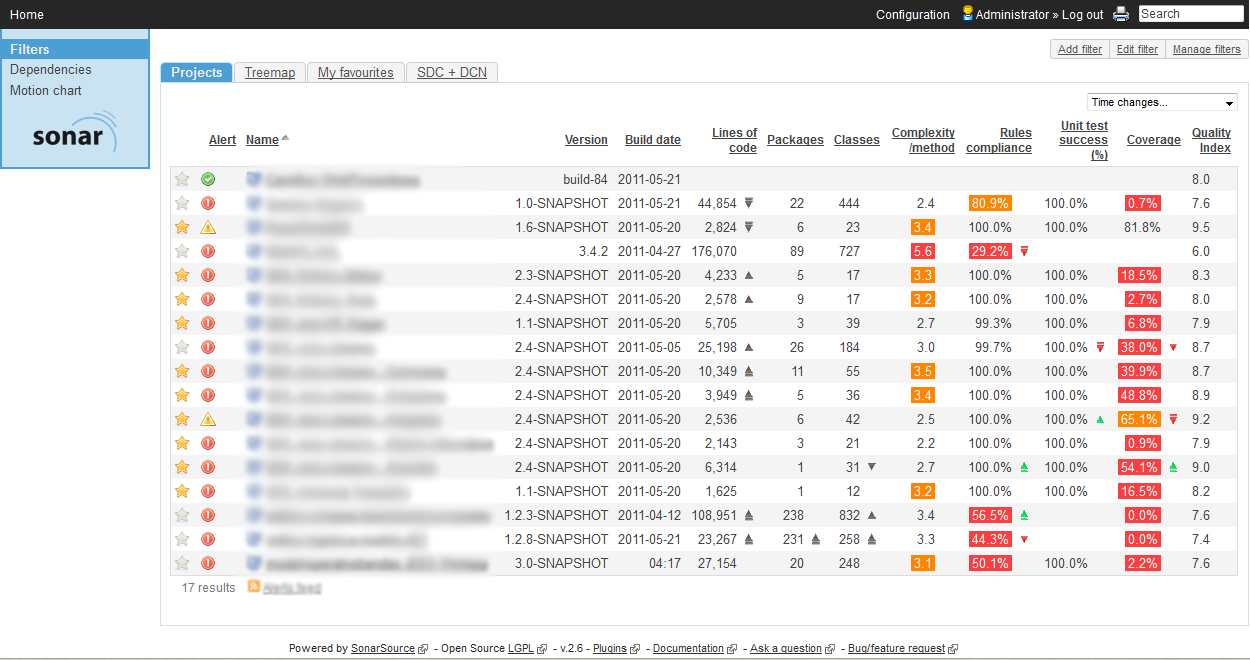
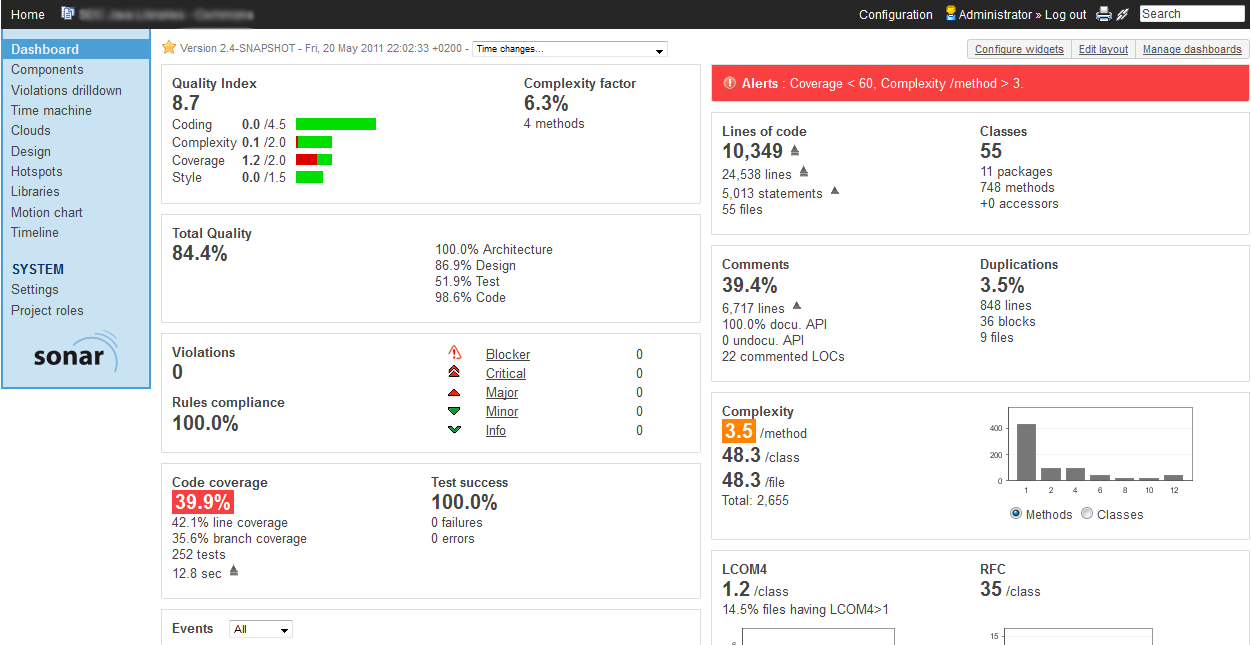
To easy access the analysis results from the Jenkins dashboard, simply click the Sonar icon that shows in the project page:

# Sonar’s Project Dashboard

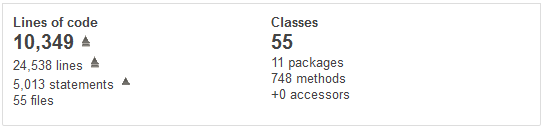
The project dashboard in Sonar is the hub where to find all quality-related information about your projects. The project dashboard is flexible, the layout can be arranged and extended with plugins to match your needs, and provides easy access to metrics, historical data, architecture and design analysis and many more.

To access a project dashboard, you may click on the project name from main Sonar dashboard, or just click on the Sonar logo in a job page in Jenkins.

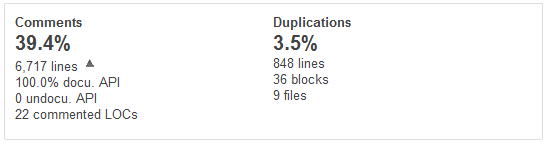
 

The most important panels to look at in the project dashboard are the following:

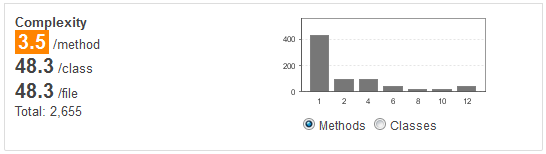
* **Project size**. Information on the project size: lines of code, statements, files, packages, classes and methods.



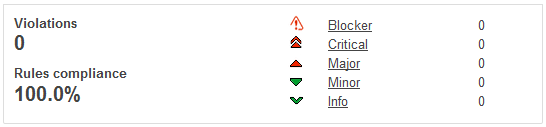
* **Documentation and duplication**. Information on the project documentation: density of comments, percentage of API documented (Javadoc), commented lines that may contain source code. Information on duplicated code including a percentage of duplicated code found.



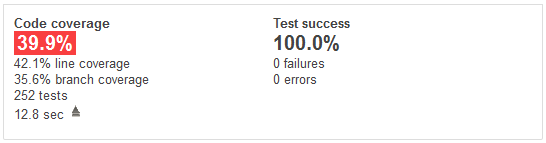
* **Complexity**. Information on source complexity: total complexity, average by file, by class and by method, complexity distribution histogram.



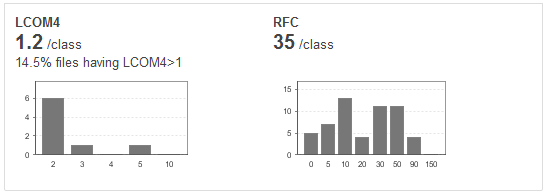
* **Quality**. Information on quality issues: number of violations, percentage of compliance (based on project size) and violations by severity (blocker, critical, major, minor, info).



* **Tests**. Information on test results: number of tests, test failures, percentage of test success, coverage of tests.



* **OOD Metrics**. Information on some object-oriented design metrics: lack of cohesion, response for class.

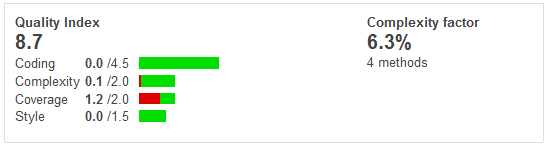
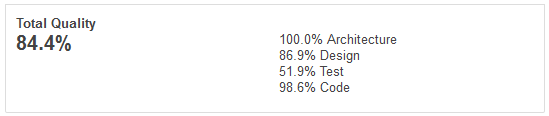


* **Alerts**. Summary of alerts, e.g. code coverage is under defined threshold.

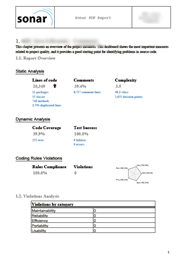


Other interesting panels contributed by plugins are the following:

* **Quality Index** or **Total Quality**. A blended index showing in just one value the overall status of the project, that may be helpful to quickly compare one projects with others:

* **PDF Report**. Creates and downloads a PDF report with the status of the project:

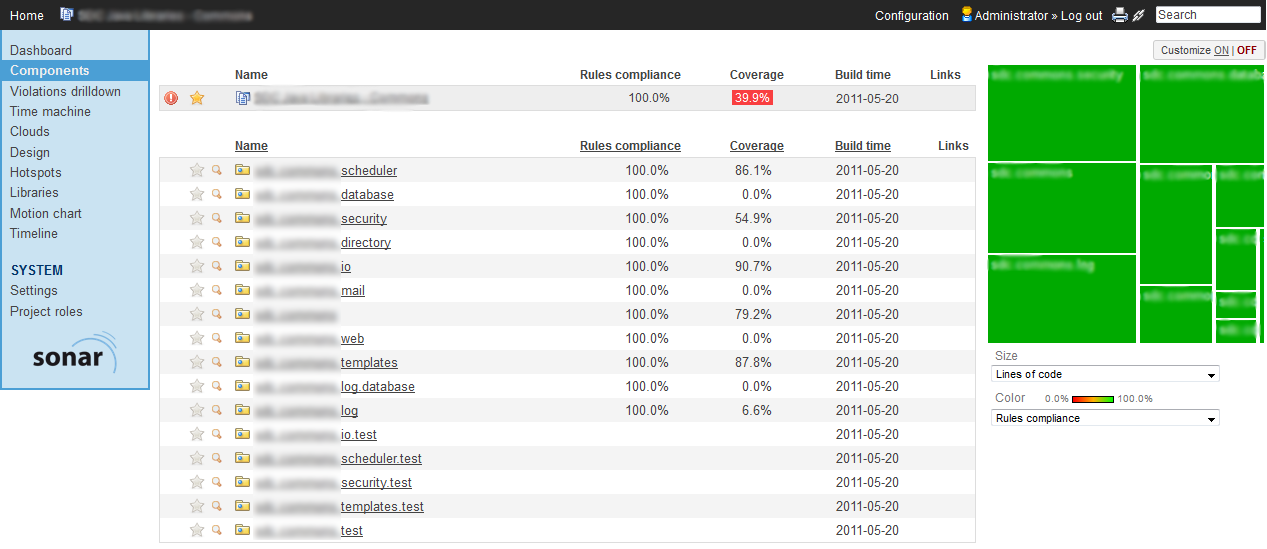
 

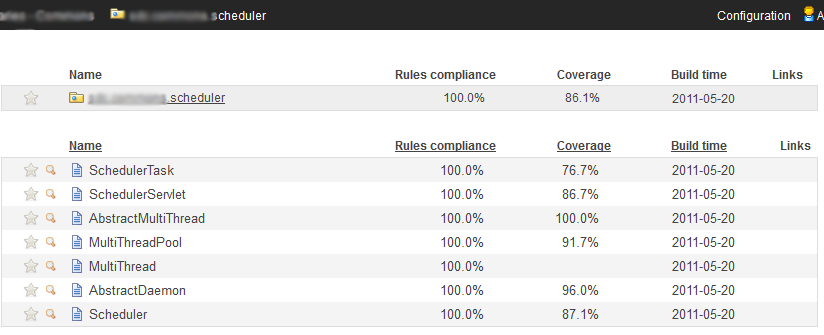
If you click in any metric showed in the project dashboard you will access the drilldown view, where you can explore the detailed values on the metric by package, class or at source level. We will describe drilldowns on next section.

# Component and Metric Drilldowns

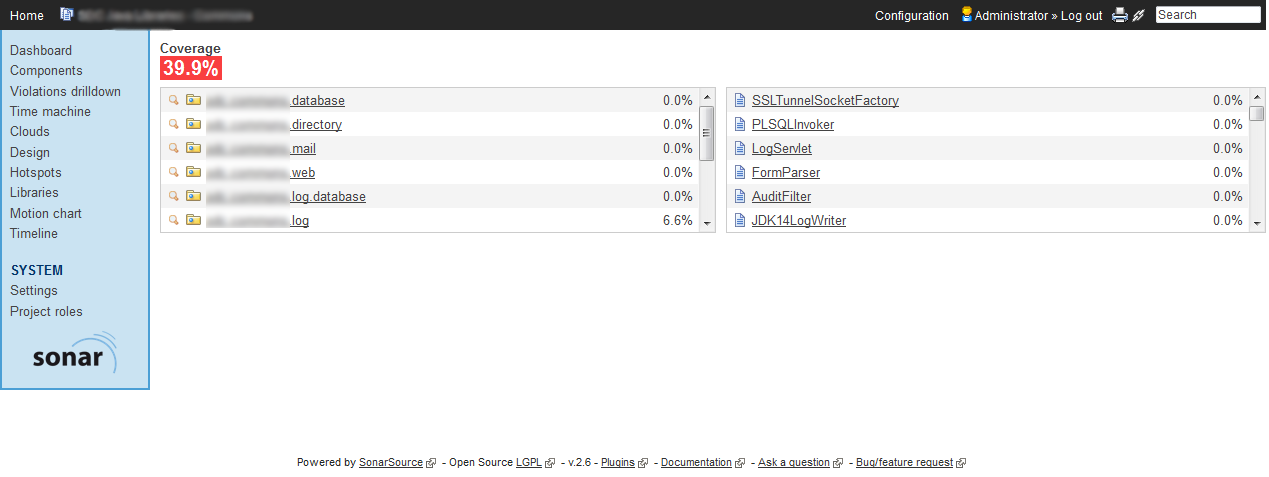
To access to details of the project status, Sonar dashboard allows exploration in two dimensions:

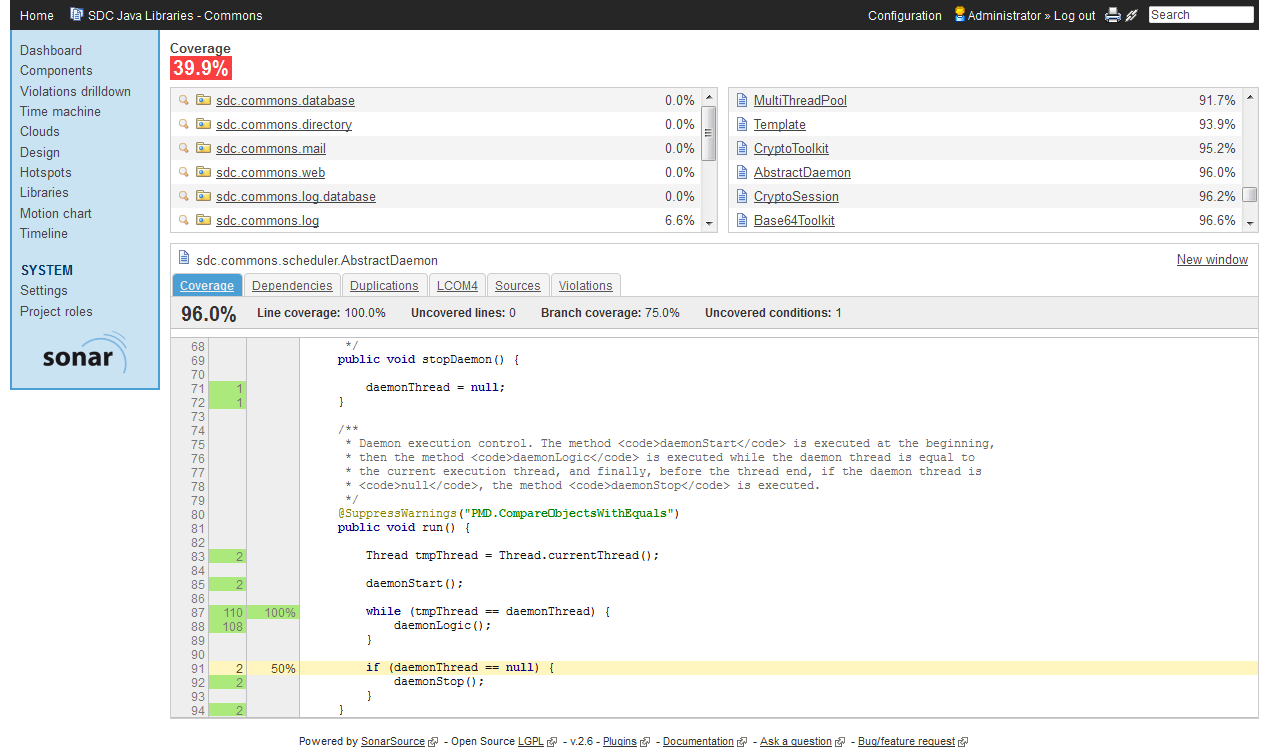
* **By component.** Selecting **Components** in the left menu will show you the list of ‘child’ components and you can select any of them to explore it down. Components may be, from top to bottom, Maven modules (when analysing multi-module projects), Java packages or Java classes.





* **By metric**. Clicking on any metric value in the project dashboard (either at project or component level) you will access the drilldown view when you can explore the metric values down from project level to source level.



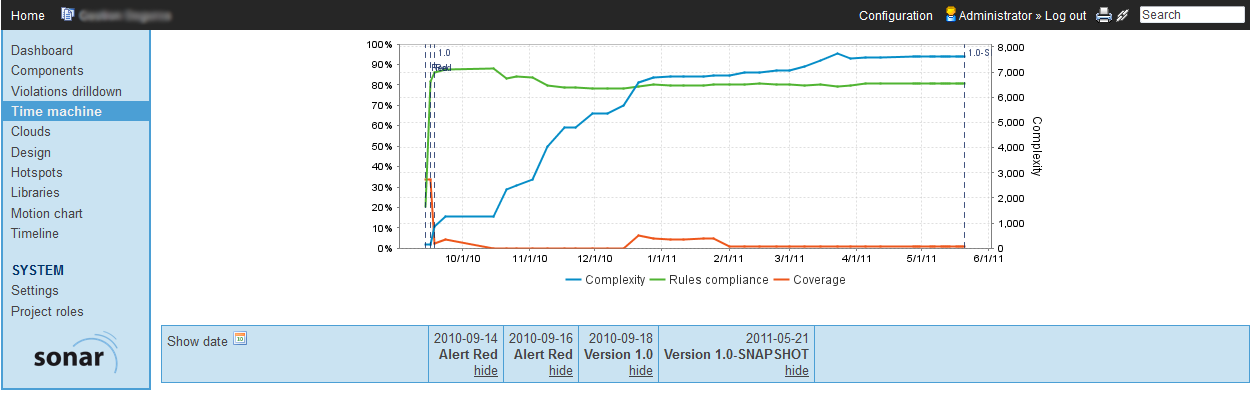


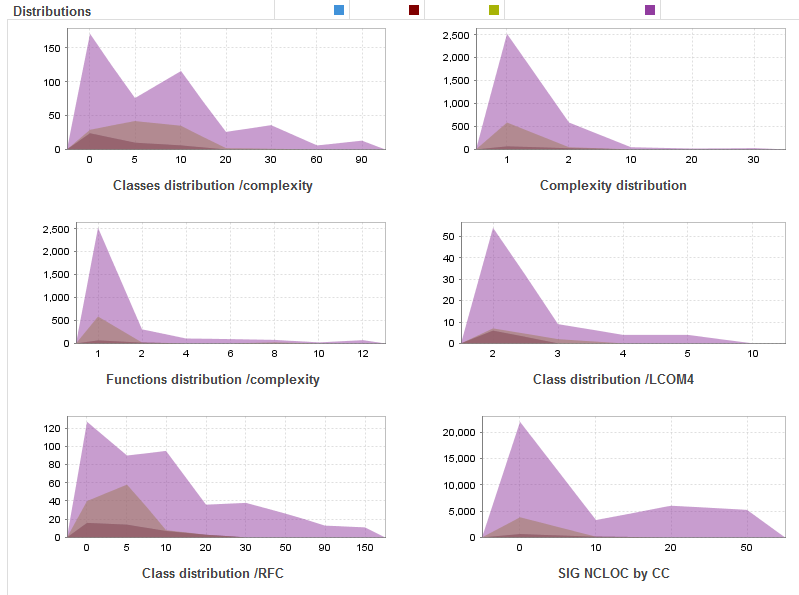
By combining component and metric levels of access project team members can access the relevant information for review in a few clicks.

# Time Machine: Reviewing the Project History

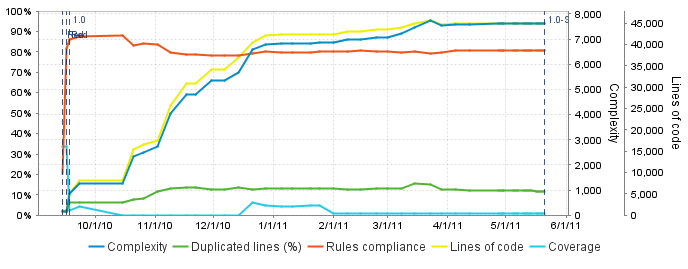
Time Machine is a powerful Sonar feature that allows easy access to historical data, show metric series in graphs or to compare two versions.

To access it, select the option **Time machine** in the left menu of the project dashboard:

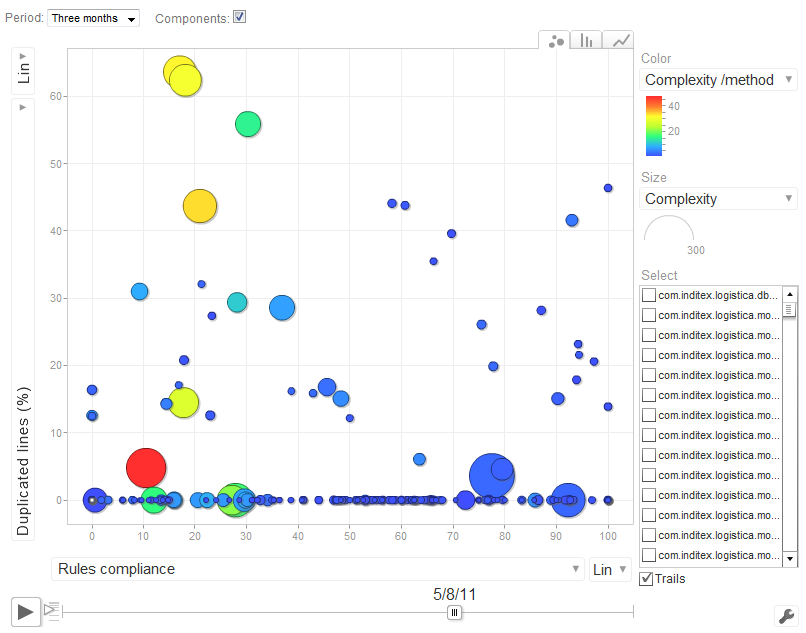


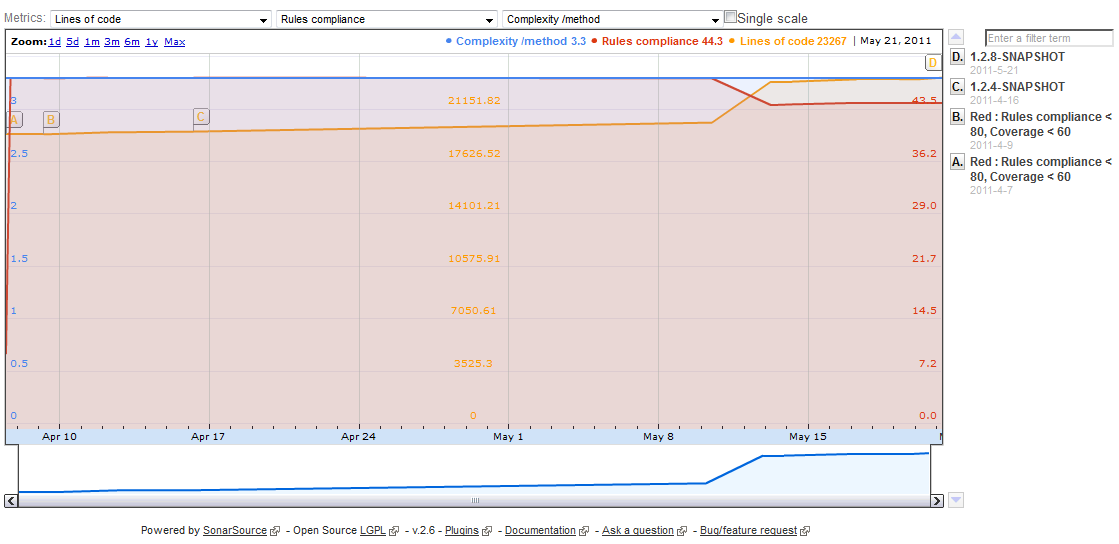


You can add new metrics to the graph by activating the check-boxes and clicking the button **Compare on chart** at the bottom of the screen:



Two alternatives to the Time Machine are provided by the Motion chart and the Timeline plugins. These plugins, once installed, creates nice views on historical data that may be of help to understand the how the project is evolving with time:



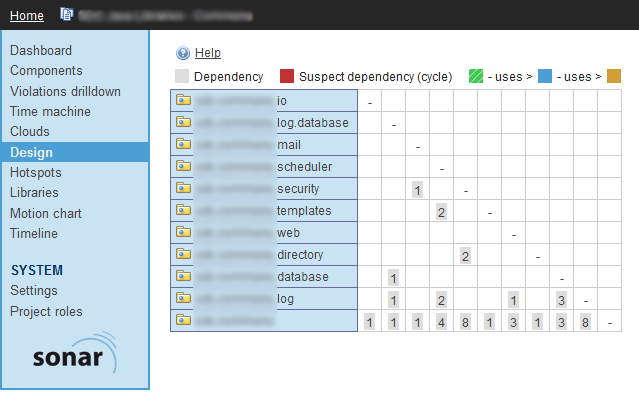


# DSM Analysis

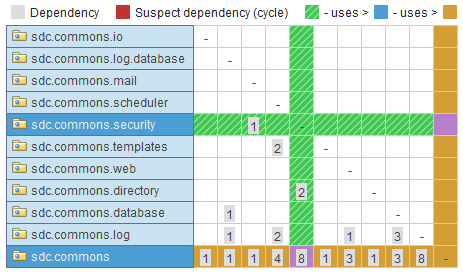
Another powerful feature of Sonar is the Design Structure Matrix (DSM) showing how components depend one of others, highlighting dependency cycles and showing details on dependencies.

The DSM view allows a quick way of examining the project compliance to architectural definition without needing to setup explicit rules to validate the architecture.

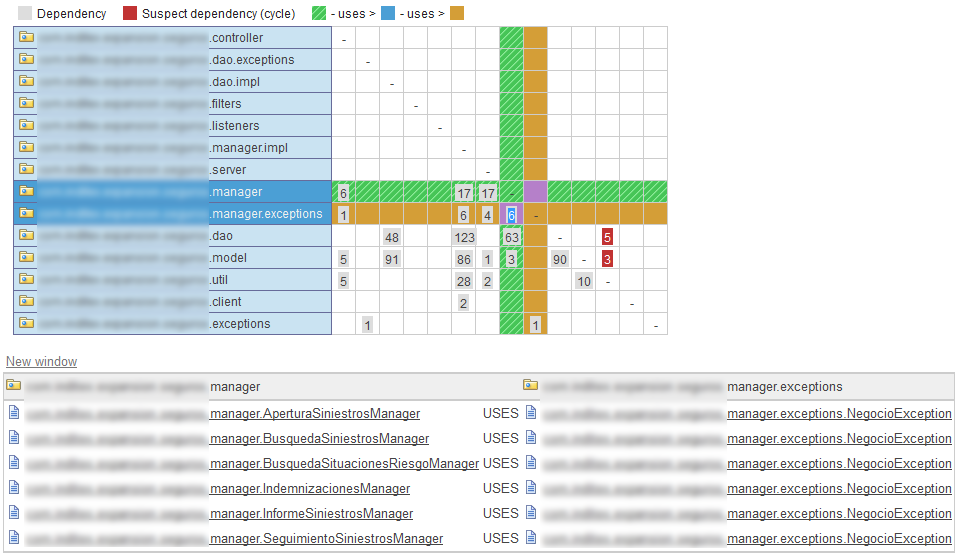
To access the DSM view, just select **Design** on the left menu of the project dashboard:



Clicking on a package or intersection will highlight related usage:



Double-clicking will load a detail panel with textual references to the dependencies:



You can drilldown double-clicking on package names. By combining these features you can discover architectural non compliances and dependency cycles, including the detailed information needed to fix the issues.

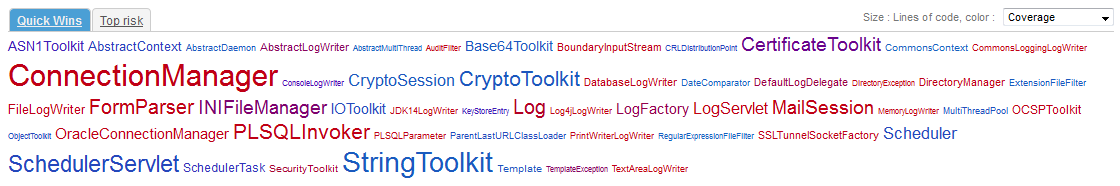
# Taking Decisions: Quick Wins, Top Risks and Hotspots

Besides showing raw metric data, Sonar has several views that combine multiple metrics to help taking decisions. If you have 5,000 violations, what should I focus on? If my project has low test coverage, what components should I improve first?

## Quick Wins

The Quick Wins view is a cloud diagram showing coverage or rules compliance versus size of components. The cloud shows at a glance what components I should focus if I want to improve code coverage metrics or rules compliance.

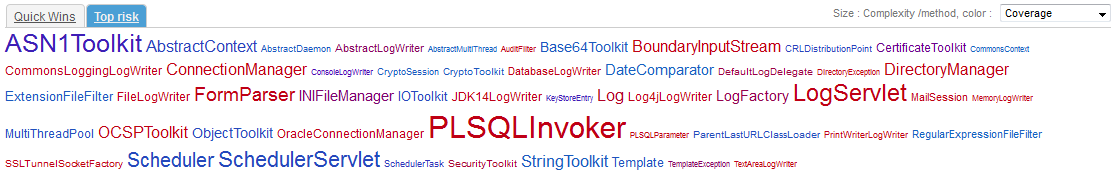
To access the Quick Wins view, select **Clouds** in the left menu of the project dashboard:



The greater the component (class) name, the greater it is. Colour goes from red – lower metric value – to blue – higher metric value. Therefore, focusing on big and red classes is a good move to improve these metrics.

## Top Risks

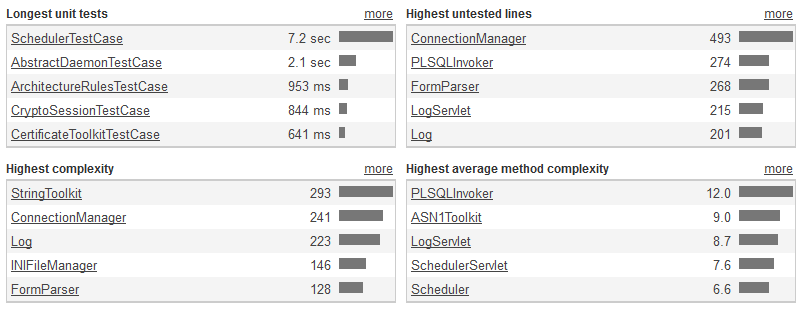
A second cloud is also available by clicking on **Top risk** tab. The Top Risks view focuses not on size but on complexity. Usually, complex components are most likely to cause errors and simple are straightforward, less prone to errors:



In this case, the name size is proportional to the complexity per method. Therefore, focusing on big and red classes is a good move to reduce risk due to complex components poorly tested.

## Hotspots

The Hotspots view can be accessed through the left menu in the project dashboard by selecting **Hotspots**. This view shows nine lists of components based on several criteria. These lists are an excellent way of finding out where to focus efforts:



* **Most violated rules**. The top five rules by number of violations.
* **Most violated**. The top five components by number of violations.
* **Longest unit tests**. The top five tests by running time.
* **Highest untested lines**. The top five components with most untested lines.
* **Highest complexity**. The top five components with the highest complexity.
* **Highest average method complexity**. The top five components with the highest average complexity.
* **Highest duplications**. The top five components with the highest amount of code duplicated.
* **Most undocumented APIs**. The five documents with less documentation.
* **Highest LCOM/RFC**. Top five by these OOD metrics.

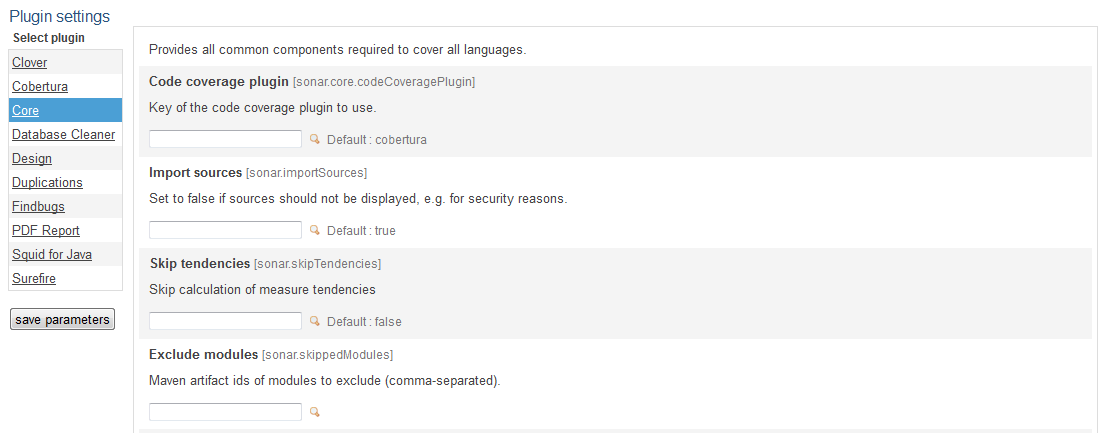
# Customising the Analysis

There are three ways of customising the Sonar analysis: through analysis properties (as explained in section 2 of this document), through the Sonar dashboard and through changes in the quality profile.

## Project Analysis Configuration

From the project dashboard view, select **Settings** in the left menu. This will show a screen from where we can customise the analysis. Most of these settings are overridable through analysis properties but it is recommended to add the configuration parameters here so they are always taken into account.

You may refer to section 2 for description of the main configuration settings or just follow UI explanations for all of them:



## Customising the Quality Profile

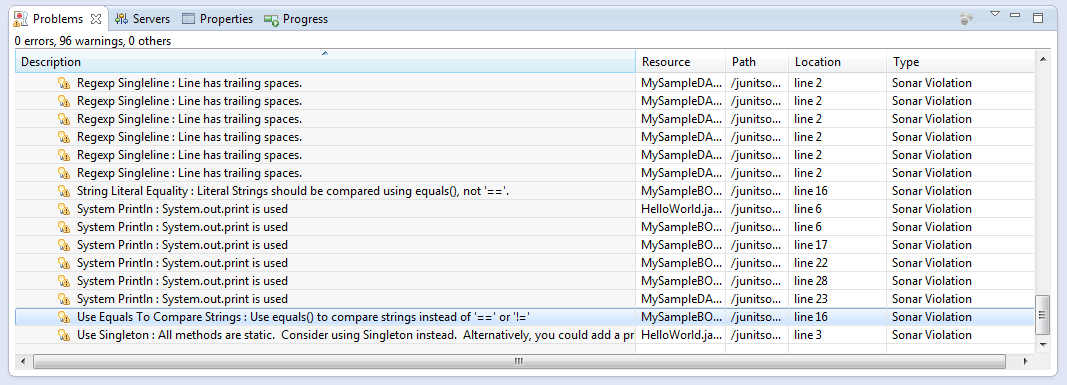
But this is a recommended profile, based on coding and naming standards plus industry best practices as those provided by Checkstyle, PMD or FindBugs. However, any project or client may have different needs, and therefore the customisation of the quality profile should be taken into consideration. This is remarkable for existing projects with no previous usage of static code profilers. Imagine the amount of violations that may be showed on Sonar when 600+ checkers are profiling years and years of code. It is best approach, after first analysis, to review what violations are most critical and ignore the other. Later, when the project evolves now under the umbrella of the profilers, we can raise the bar adding more rules to be checked.

For more information on how to customise the quality profile, including how to add architecture rules to it, refer to Sonar on-line documentation following the links included in the References section.

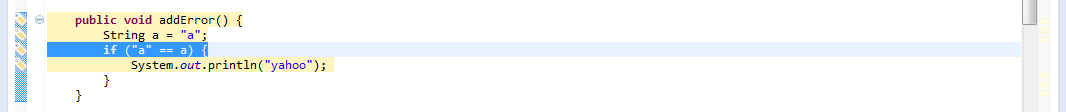
# Sonar analysis with Eclipse plugin

Eclipse can be configured to remotely connect to a Sonar instance using the Eclipse Sonar plugin. The installation and configuration of the plugin is described in the Sonar Installation and Setup guide.

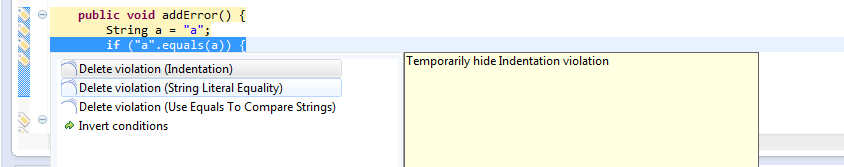
Once the Eclipse Sonar plugin is setup and the project is associated with the Sonar instance the Sonar violations will start showing up in the Problems tab under Type “Sonar Violation”.



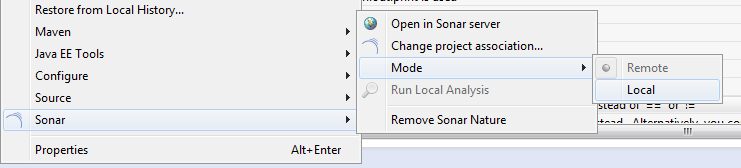
Selecting the violation and drilling down using double click of the mouse will open the line of code for fixing the violation.



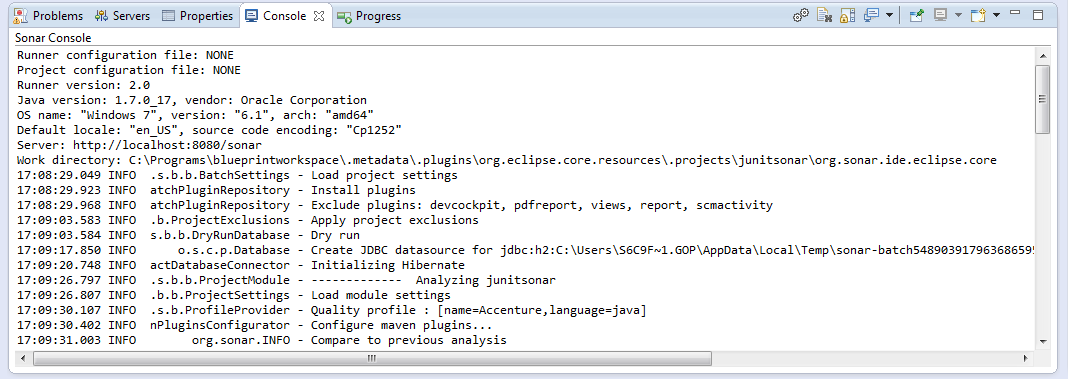
Once a violation has been fixed the violation can be removed from the list by clicking on the validation icon and Deleting the violation item.



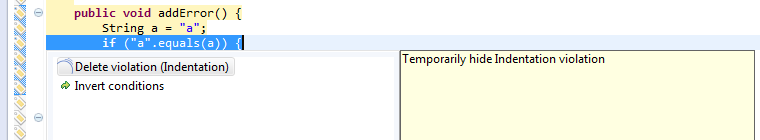
Once the code is fixed it is recommended that the developer run a local analysis using the Sonar local feature to check if the code change fixed the violation. This can be done by navigating to the Sonar section on the project level menu option and selecting local.



This will start a local Sonar analysis on the developers’ machine.



The developer can instantly view after the local analysis is complete if the code change resolves the issues.



It is recommended that developers as a practise use this local feature to review errors and fix them during the course of the workday. Once the violations are fixed the code will get checked in and the continuous integration setup will build the code on the remote server and profile it again with Sonar. The developer during the start of their day can connect to the remote instance after the daily build runs every day to refresh the list of violations.

# References

Sonar: <http://www.sonarsource.org/>

Maven: <http://maven.apache.org/>

Sonar on-line documentation: <http://docs.codehaus.org/display/SONAR/Documentation>

How to configure quality profiles: <http://docs.codehaus.org/display/SONAR/Quality+profiles>

How to configure architecture rules: <http://docs.codehaus.org/display/SONAR/Architecture+rule+engine>

# Document Control

## Change History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Version | Date | Author | Approver | Comment |
| 0.1 | 05/19/2011 | Jorge Hidalgo |  | First draft version |
|  |  |  |  |  |