**Chapter 9. Code Quality**

**Introduction**

Few would deny the importance of writing quality code. High quality code contains less bugs, and is easier to understand and easier to maintain. However, the precise definitions of code quality can be more subjective,varying between organizations, teams, and even individuals within a team.

This is where coding standards come into play. Coding standards are rules, sometimes relatively arbitrary, that define the coding styles and conventions that are considered acceptable within a team or organization. In many cases, agreeing on a set of standards, and applying them, is more important than the standards themselves. Indeed, one of the most important aspects of quality code is that it is easy to read and to understand. If developers within a team all apply the same coding standards and practices, the code will be more readable, at least for members of that team. And if the standards are commonly used within the industry, the code will also be more readable for new developers arriving on the team.

Coding standards include both aesthetic aspects such as code layout and formatting, naming conventions, and so forth, as well as potentially bad practices such as missing curly brackets after a condition in Java. A consistent coding style lowers maintenance costs, makes code clearer and more readable, and makes it easier to work on code written by other team members.

Only an experienced developer can really judge code quality in all its aspects. That is the role of code reviews and, among other things, practices like pair programming. In particular, only a human eye can decide if a piece of code is truly well written, and if it actually does what the requirements ask of it. However, code quality metrics tools can help a great deal. In fact it is unrealistic to try to enforce coding standards without the use of such tools.

These tools analyze your application source code or byte code, and check whether the code respects certain rules. Code quality metrics can encompass many aspects of code quality, from coding standards and best practices right through to code coverage, with everything from compiler warnings to TODO comments in between. Certain metrics concentrate on measurable characteristics of your code base, such as the number of lines of code (NLOC), average code complexity, or the number of lines per class. Others focus on more sophisticated static analysis, or on looking for potential bugs or poor practices in your code.

There are a wide range of code quality reporting plugins available for Jenkins. Many are for Java static analysis tools, such as Checkstyle, PMD, FindBugs, Cobertura, and JDepend. Others, such as fxcop and NCover, are focused on .NET applications.

With all of these tools, you need to configure your build job to generate the code quality metrics data before Jenkins can produce any reports.

The notable exception to this rule is Sonar. Sonar can extract code quality metrics from any Maven project, with no additional configuration required in your Maven project. This is great when you have large numbers of existing Maven projects that you need to integrate into Jenkins, and you want to configure consistent code quality reporting across all of your projects.

In the rest of this chapter, we will see how to set up code quality reporting in your Jenkins builds, and also how you can use it as an effective part of your build process

## Code Quality in Your Build Process

Before we look at how to report on code quality metrics in Jenkins, it can be useful to take a step back and look at the larger picture. Code Quality metrics are of limited value in isolation—they need to be part of a broader process improvement strategy.

The first level of code quality integration should be the IDE. Modern IDEs have great support for many code quality tools—Checkstyle, PMD, and FindBugs all have plugins for Eclipse, NetBeans, and IntelliJ, which provide rapid feedback for developers on code quality issues. This is a much faster and more efficient way to provide feedback for individual developers, and to teach developers about the organizational or project coding standards.

The second level is your build server. In addition to your normal unit and integration test build jobs, set up a dedicated code quality build, which runs after the normal build and test. The aim of this process is to produce project-wide code quality metrics, to keep tabs on how the project is doing as a whole, and to address any issues from a high level. The effectiveness of these reports can be increased by a weekly code quality review, in which code quality issues and trends are discussed within the team.

It is important to run this job separately, because code coverage analysis and many static analysis tools can be quite slow to run. It is also important to keep any code coverage tests well away from builds, as the code coverage process produces instrumented code which should never be deployed to a repository for production use.

Code quality reporting is, by default, a relatively passive process. No one will know the state of the project if they don’t seek out the information on the build server. While this is better than nothing, if you are serious about code quality, there is a better way. Rather than simply reporting on code quality, set up a dedicated code quality build, which runs after the normal build and test, and configure the build to fail if code quality metrics are not at an acceptable level. You can do this in Jenkins or in your build script, although one advantage of configuring this outside of your build script is that you can change code quality build failing criteria more easily without changing the project source code.

As a final word, remember that coding standards are guidelines and recommendations, not absolute rules. Use failing code quality builds and code quality reports as indicators of a possible area of improvement, not as measurements of absolute value.

### **CHECKSTYLE**

[Checkstyle](http://checkstyle.sourceforge.net/) is a static analysis tool for Java. Originally designed to enforce a set of highly-configurable coding standards, Checkstyle now also lets you check for poor coding practices, as well as overly complex and duplicated code. Checkstyle is a versatile and flexible tool that should have its place in any Java-based code quality analysis strategy.

Checkstyle supports a very large number of rules, including ones relating to naming conventions, annotations, javadoc comments, class and method size, code complexity metrics, poor coding practices, and many others.

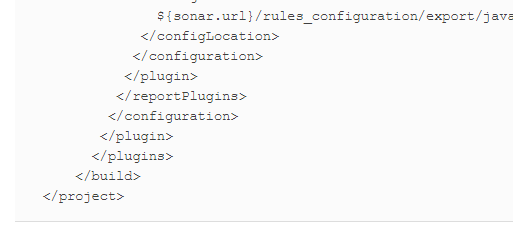
Duplicated code is another important code quality issue—duplicated or near-duplicated code is harder to maintain and to debug. Checkstyle provides some support for the detection of duplicated code, but more specialized tools such as CPD do a better job in this area.

In Maven 2, you could add something like the following to the <reporting> section:



For a Maven 3 project, you need to add the plugin to the <reportPlugins> element of the <configuration>section of the *maven-site-plugin*:





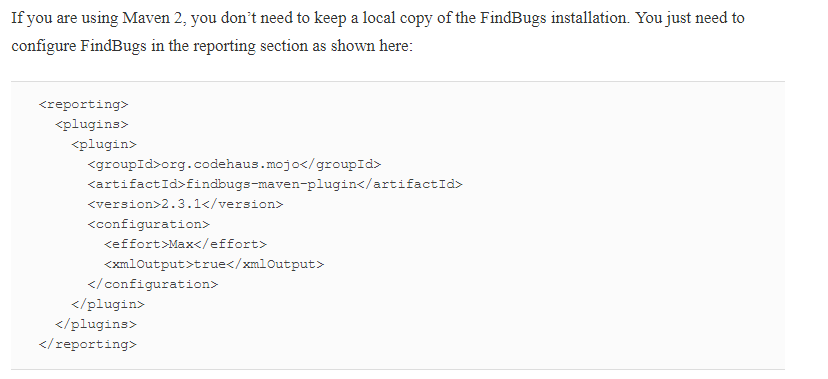
Now, running mvn checkstyle:checkstyle or mvn site will analyse your source code and generate XML reports that Jenkins can use.

Note that in the last example, we used a Checkstyle ruleset that we have uploaded to a Sonar server (defined by the ${sonar.url} property). This strategy makes it easy to use the same set of Checkstyle rules for Eclipse, Maven, Jenkins, and Sonar.

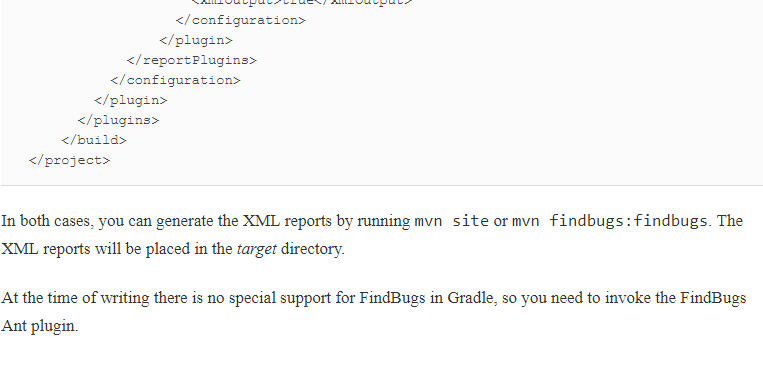
### FINDBUGS

FindBugs is a powerful code quality analysis tool that checks your application byte code for potential bugs, performance problems, or poor coding habits. FindBugs is the result of research carried out at the University of Maryland lead by Bill Pugh, that studies byte code patterns coming from bugs in large real-world projects, such as the JDKs, Eclipse, and source code from Google applications. FindBugs can detect some fairly significant issues such as null pointer exceptions, infinite loops, and unintentionally accessing the internal state of an object. Unlike many other static analysis tools, FindBugs tends to find a smaller number of issues, but of those issues, a larger proportion will be important.

FindBugs is less configurable than the other tools we have seen, though in practice you generally don’t need to fine-tune the rules as much as the other tools we’ve discussed. You can list the individual rules you want to apply, but you can’t configure a shared XML file between your Maven builds and your IDE, for example.



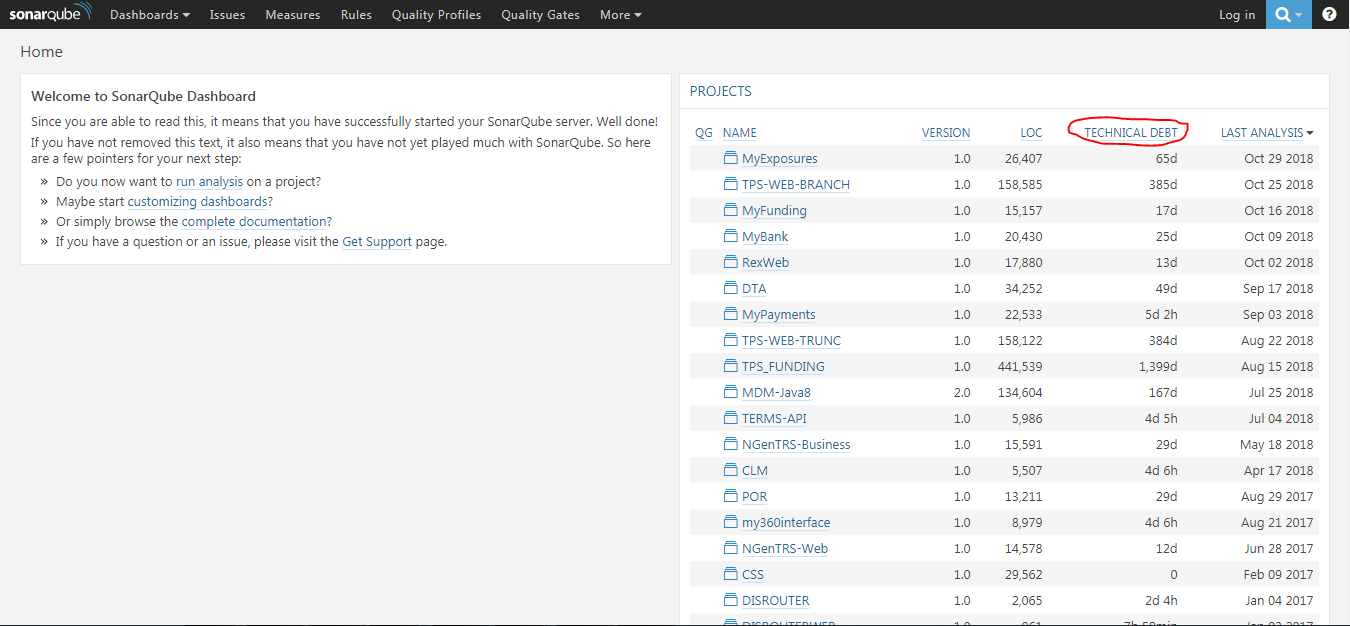


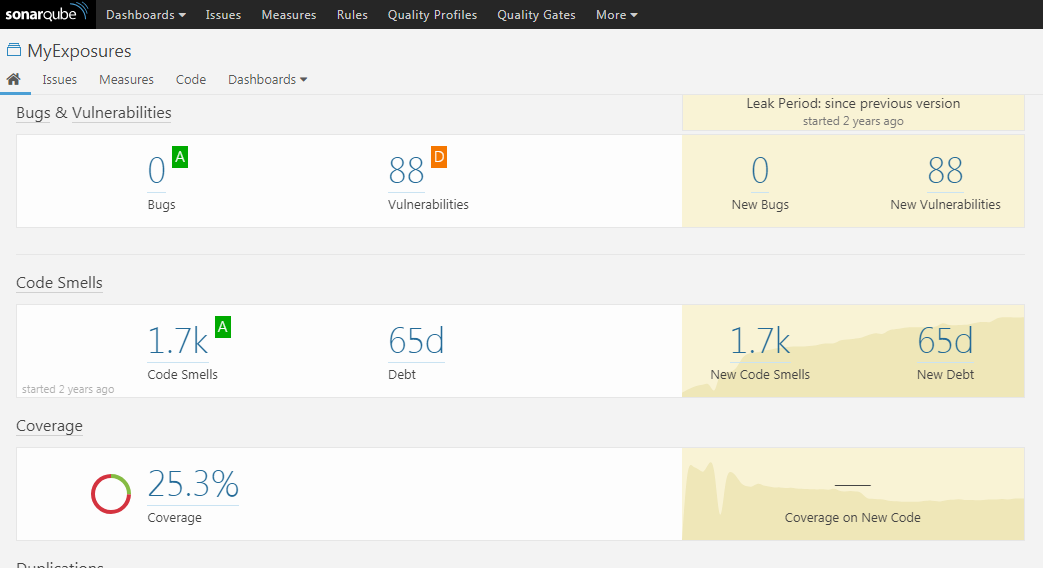


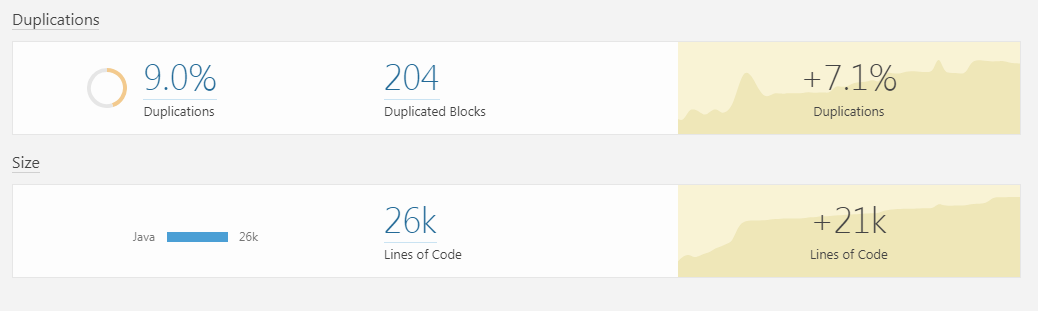
## Integrating with Sonar

[Sonar](http://www.sonarsource.org/) is a tool that centralizes a range of code quality metrics into a single website (see [Figure 9-18](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch09.html#fig-hudson-sonar)). It uses several Maven plugins (Checkstyle, PMD, FindBugs, Cobertura or Clover, and others) to analyse Maven projects and generate a comprehensive set of code quality metrics reports. Sonar reports on code coverage, rule compliance, and documentation, but also on more high-level metrics such as complexity, maintainability and even technical debt. You can use plugins to extend its features and add support for other languages (such as support for CodeNarc for Groovy source code). The rules used by the various tools are managed and configured centrally on the Sonar website, and the Maven projects being analyzed don’t require any particular configuration. This makes Sonar a great fit for working on Maven projects where you have limited control over the pom files.

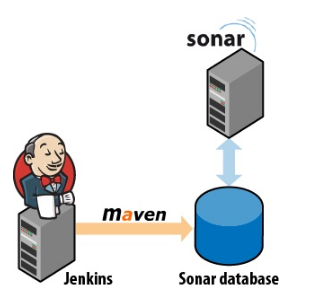
In one of the most common usages of Sonar, Sonar automatically runs a set of Maven code quality related plugins against your Maven project, and stores the results into a relational database. The Sonar server, which you run separately, then analyzes and displays the results as shown in



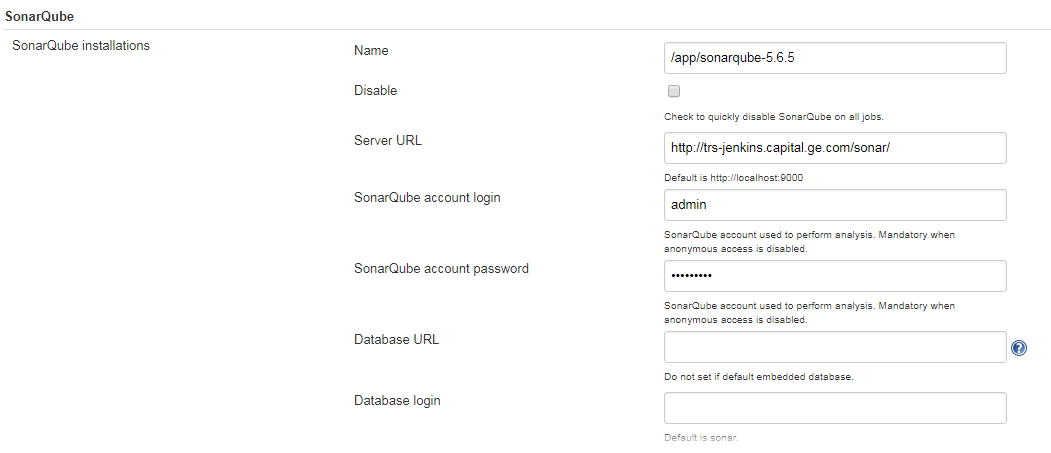


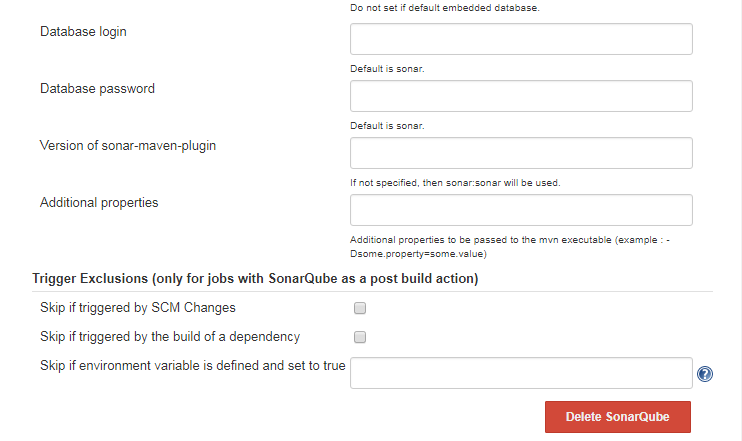


Jenkins integrates well with Sonar. The Jenkins Sonar Plugin lets you define Sonar instances for all of your projects, and then activate Sonar in particular builds. You can run your Sonar server on a different machine to your Jenkins instance, or on the same. The only constraint is that the Jenkins instance must have JDBC access to the Sonar database, as it injects code quality metrics directly into the database, without going through the Sonar website



You install the plugin in the usual way, via the Plugin Manager. Once installed, you configure the Jenkins Sonar plugin in the Configure System screen, in the Sonar section. This involves defining your Sonar instances—you can configure as many instances of Sonar as you need. The default configuration assumes that you are running a local instance of Sonar with the default embedded database. This is useful for testing purposes but not very scalable. For a production environment, you will typically run Sonar on a real database such as MySQL or Postgres, and you will need to configure the JDBC connection to the production Sonar database in Jenkins. You do this by clicking on the Advanced button and filling in the appropriate fields





The other thing you need to configure is when the Sonar build will kick off in a Sonar-enabled build job. You usually configure Sonar to run with one of the long-running Jenkins build jobs, such as the code quality metrics build. It is not very useful to run the Sonar build more than once a day, as Sonar stores metrics in 24-hour slices. The default configuration will kick off a Sonar build in a Sonar-enabled build job whenever the job is triggered by a periodically scheduled build or by a manual build.