**Chapter 6. Automated Testing**

**Introduction**

If you aren’t using automated tests with your Continuous Integration setup, you’re really missing out on something big. Believe me—CI without automated tests is really just a small improvement on automatically scheduled builds. Now don’t get me wrong, if you’re coming from nothing, that’s already a great step forward—but you can do much better. In short, if you are using Jenkins without any automated tests, you are not getting anywhere near as much value out of your Continuous Integration infrastructure as you should.

One of the basic principles of Continuous Integration is that a build should be verifiable. You have to be able to objectively determine whether a particular build is ready to proceed to the next stage of the build process, and the most convenient way to do this is to use automated tests. Without proper automated testing, you find yourself having to retain many build artifacts and test them by hand, which is hardly in the spirit of Continuous Integration.

There are many ways you can integrate automated tests into your application. One of the most efficient ways to write high quality tests is to write them first, using techniques such as Test-Driven Development (TDD) or Behavior-Driven Development (BDD). In this approach, commonly used in many Agile projects, the aim of your unit tests is to both clarify your understanding of the code’s behavior and to write an automated test that the code does indeed implement this behavior. Focusing on testing the expected behavior, rather than the implementation, of your code also makes for more comprehensive and more accurate tests, and thus helps Jenkins to provide more relevant feedback.

Of course, more classical unit testing, done once the code has been implemented, is also another commonly-used approach, and is certainly better than no tests at all.

Jenkins is not limited to unit testing, though. There are many other types of automated testing that you should consider, depending on the nature of your application, including integration testing, web testing, functional testing, performance testing, load testing and so on. All of these have their place in an automated build setup.

Jenkins can also be used, in conjunction with techniques like Behavior-Driven Development and Acceptance Test Driven Development, as a communications tool aimed at both developers and other project stakeholders. BDD frameworks such as easyb, fitnesse, jbehave, rspec, Cucumber, and many others, try to present acceptance tests in terms that testers, product owners, and end users can understand. With the use of such tools, Jenkins can report on project progress in business terms, and so facilitate communication between developers and non-developers within a team.

For existing or legacy applications with little or no automated testing in place, it can be time-consuming and difficult to retro-fit comprehensive unit tests onto the code. In addition, the tests may not be very effective, as they will tend to validate the existing implementation rather than verify the expected business behavior. One useful approach in these situations is to write automated functional tests (“regression”) tests that simulate the most common ways that users manipulate the application. For example, automated web testing tools such as Selenium and WebDriver can be effectively used to test web applications at a high level. While this approach is not as comprehensive as a combination of good quality unit, integration and acceptance tests, it is still an effective and relatively cost-efficient way to integrate automated regression testing into an existing application.

In this chapter, we will see how Jenkins helps you keep track of automated test results, and how you can use this information to monitor and dissect your build process.

# Automating Your Unit and Integration Tests

The first thing we will look at is how to integrate your unit tests into Jenkins. Whether you are practicing Test-Driven Development, or writing unit tests using a more conventional approach, these are probably the first tests that you will want to automate with Jenkins.

Jenkins does an excellent job of reporting on your test results. However, it is up to you to write the appropriate tests and to configure your build script to run them automatically. Fortunately integrating unit tests into your automated builds is generally relatively easy.

There are many unit testing tools out there, with the xUnit family holding a predominant place. In the Java world, JUnit is the de facto standard, although TestNG is another popular Java unit testing framework with a number of innovative features. For C# applications, the NUnit testing framework proposes similar functionalities to those provided by JUnit, as does Test::Unit for Ruby. For C/C++, there is CppUnit, and PHP developers can use PHPUnit. And this is not an exhaustive list!

These tools can also serve for integration tests, functional tests, web tests and so forth. Many web testing tools, such as Selenium, WebDriver, and Watir, generate xUnit-compatible reports. Behaviour-Driven Development and automated Acceptance-Test tools such as easyb, Fitnesse, Concordion are also xUnit-friendly. In the following sections we make no distinction between these different types of test, as, from a configuration point of view, they are treated by Jenkins in exactly the same manner. However, you will almost certainly need to make the distinction in your build jobs. In order to get the fastest possible feedback loop, your tests should be grouped into well-defined categories, starting with the fast-running unit tests, and then proceeding to the integration tests, before finally running the slower functional and web tests.

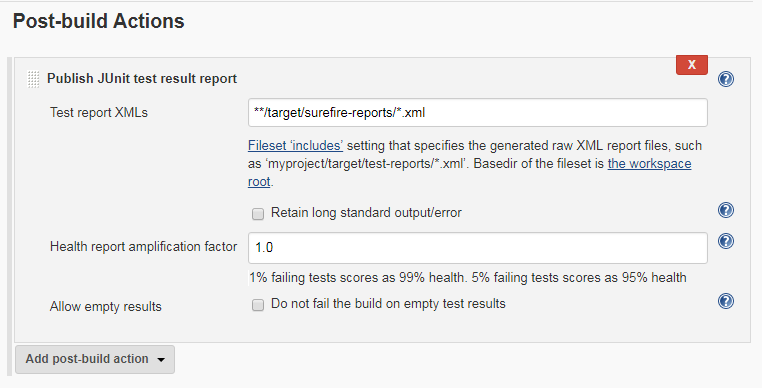
# Configuring Test Reports in Jenkins

Once your build generates test results, you need to configure your Jenkins build job to display them. As mentioned above, Jenkins will work fine with any xUnit-compatible test reports, no matter what language they are written in.

For Maven build jobs, no special configuration is required—just make sure you invoke a goal that will run your tests, such as mvn test (for your unit tests) or mvn verify (for unit and integration tests).

For freestyle build jobs, you need to do a little more configuration work. In addition to ensuring that your build actually runs the tests, you need to tell Jenkins to publish the JUnit test report. Youconfigure this in the “Post-build Actions” section (see [Figure 6-2](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch06s03.html#fig-testing-freestyle-junit-config)). Here, you provide a path to the JUnit or TestNG XML reports. Their exact location will depend on a project—for a Maven project, a path like *\*\*/target/surefire-reports/\*.xml* will find them for most projects. For an Ant-based project, it will depend on how you configured the Ant JUnit task, as we discussed above.



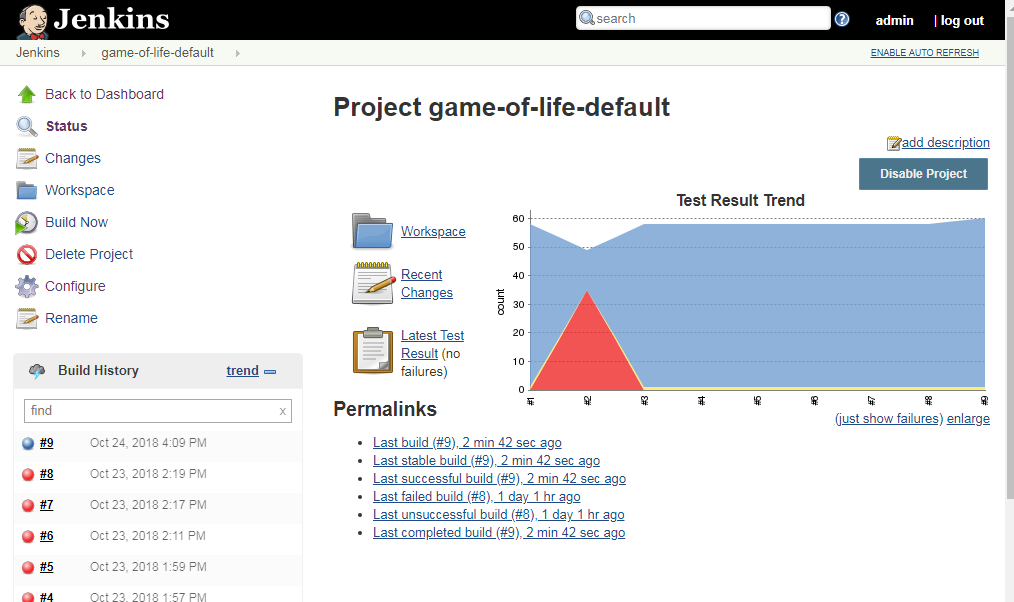


# Displaying Test Results

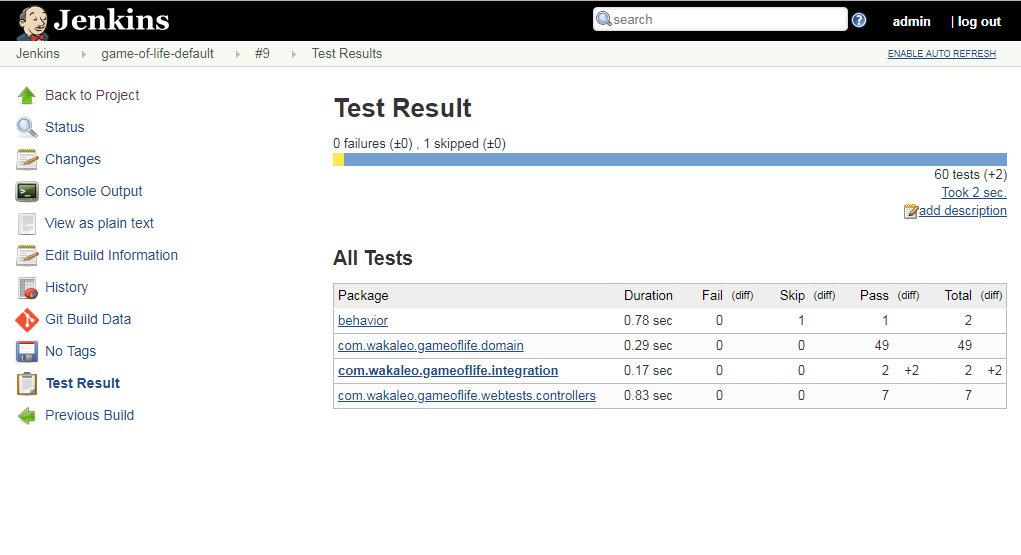
Once Jenkins knows where to find the test reports, it does a great job of reporting on them. Indeed, one of Jenkins’s main jobs is to detect and to report on build failures. And a failing unit test is one of the most obvious symptoms.

As we mentioned earlier, Jenkins makes the distinction between failed builds and unstable builds. A failed build (indicated by a red ball) indicates test failures, or a build job that is broken in some brutal manner, such as a compilation error. An unstable build, on the other hand, is a build that is not considered of sufficient quality. This is intentionally a little vague: what defines “quality” in this sense is largely up to you, but it is typically related to code quality metrics such as code coverage or coding standards, that we will be discussing later on in the book. For now, let’s focus on the failed builds.

Below we can see how Jenkins displays a Maven build job containing test failures. This is the build job home page, which should be your first port of call when a build breaks. When a build results in failing tests, the Latest Test Result link will indicate the current number of test failures in this build job (“no failures” in the illustration).You can also see how the tests have been faring over time—test failures from previous builds will also appear as red in the Test Result Trend graph.



If you click on the Latest Test Result link, Jenkins will give you a rundown of the current test results (see [Figure 6-6](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch06s04.html#fig-testing-test-result-details)). Jenkins understands Maven multimodule project structures, and for a Maven build job, Jenkins will initially display a summary view of test results per module. For more details about the failing tests in a particular module, just click on the module you are interest in.



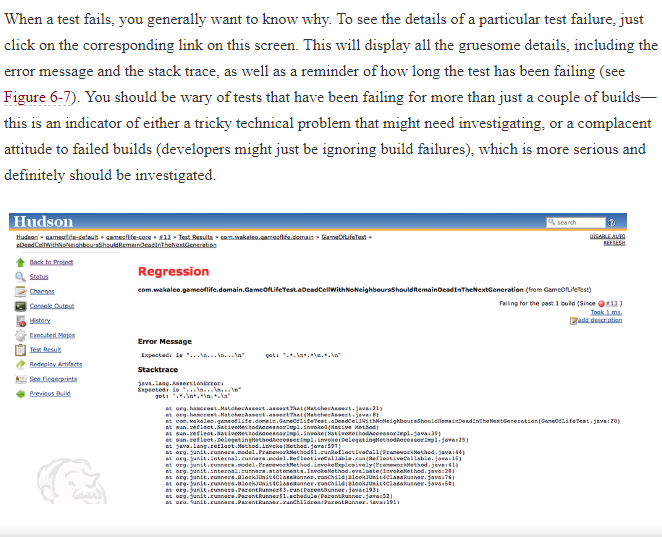
For freestyle build jobs, Jenkins will directly give you a summary of your test results, but organized by high-level packages rather than modules(as above pic).

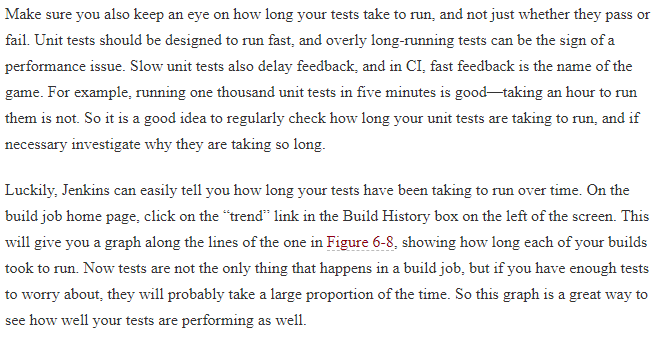
In both cases, Jenkins starts off by presenting a summary of test results for each package. From here, you can drill down, seeing test results for each test class and then finally the tests within the test classes themselves. And if there are any failed tests, these will be prominently displayed at the top of the page.

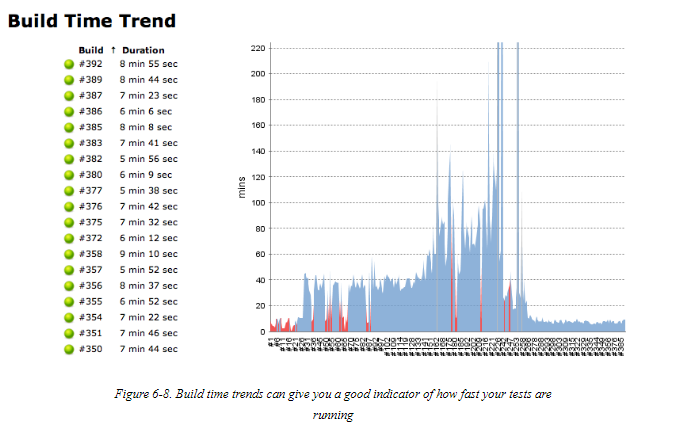
This full view gives you both a good overview of the current state of your tests, and an indication of their history. The Age column tells you how for how long a test has been broken, with a hyperlink that takes you back to the first build in which this test failed.

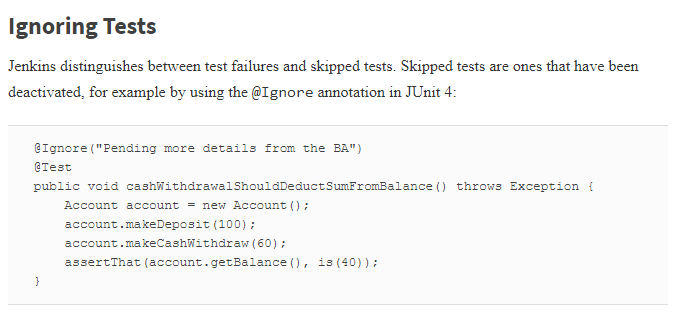


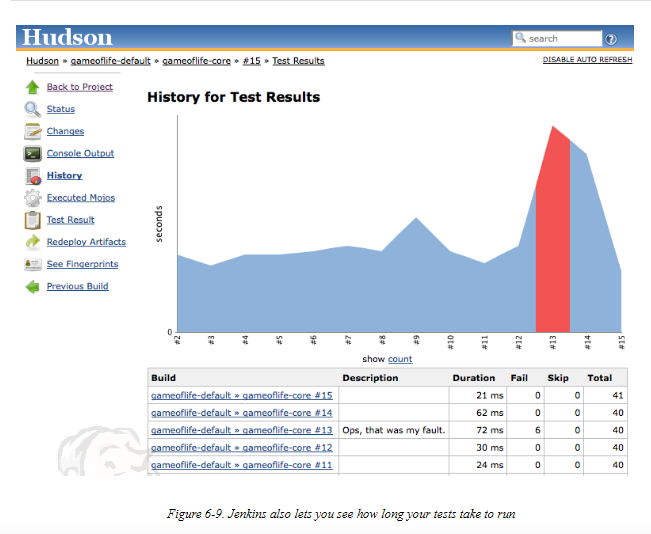
You can also add a description to the test results, using the Edit Description link in the top right-hand corner of the screen. This is a great way to annotate a build failure with some additional details, in order to add extra information about the origin of test failures or some notes about how to fix them.

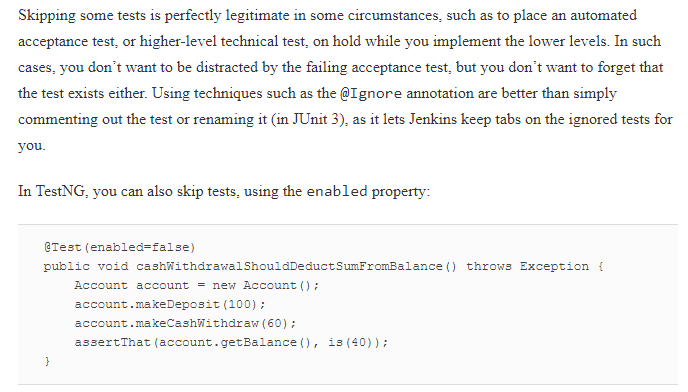


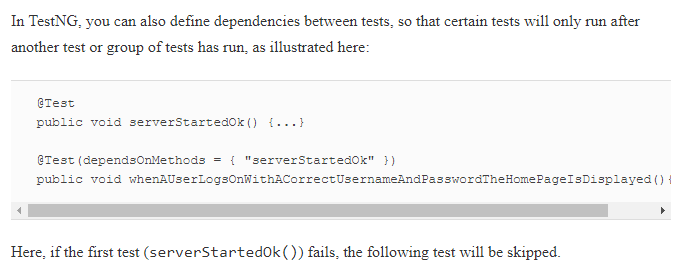


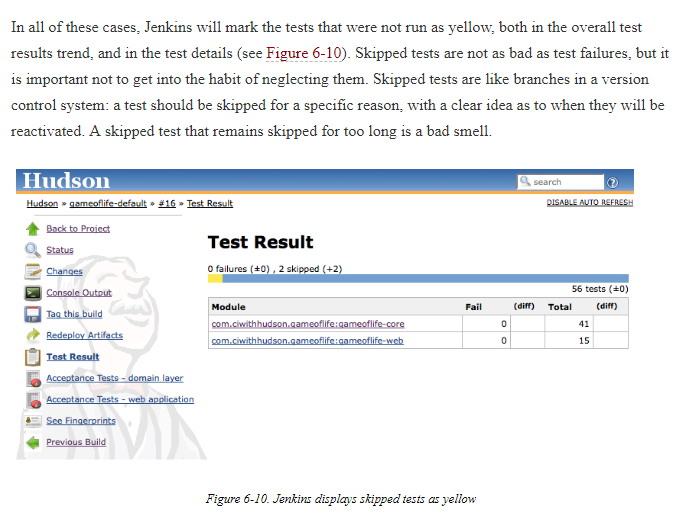












# Code Coverage

Another very useful test-related metric is code coverage. Code coverage gives an indication of what parts of your application were executed during the tests. While this in itself is not a sufficient indication of quality testing (it is easy to execute an entire application without actually testing anything, and code coverage metrics provide no indication of the quality or accuracy of your tests), it is a very good indication of code that has not been tested. And, if your team is introducing rigorous testing practices such as Test-Driven-Development, code coverage can be a good indicator of how well these practices are being applied.

Code coverage analysis is a CPU and memory-intensive process, and will slow down your build job significantly. For this reason, you will typically run code coverage metrics in a separate Jenkins build job, to be run after your unit and integration tests are successful.

There are many code coverage tools available, and several are supported in Jenkins, all through dedicated plugins. Java developers can pick between Cobertura and Emma, two popular open source code coverage tools, or Clover, a powerful commercial code coverage tool from Atlassian. For .NET projects, you can use NCover.

## Measuring Code Coverage with Cobertura

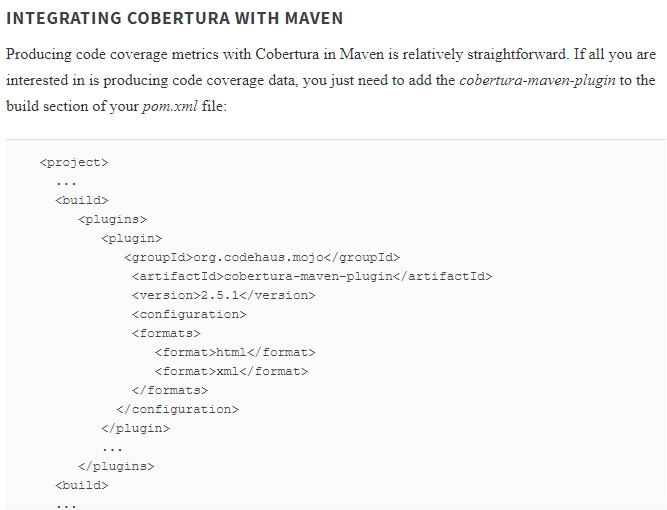
[Cobertura](http://cobertura.sourceforge.net/) is an open source code coverage tool for Java and Groovy that is easy to use and integrates well with both Maven and Jenkins.

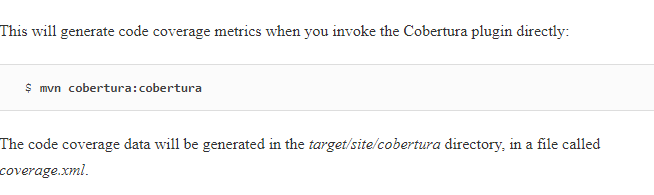
Like almost all of the Jenkins code quality metrics plugins,[[2](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch06s06.html#ftn.id2621133)] the Cobertura plugin for Jenkins will not run any test coverage metrics for you. It is left up to you to generate the raw code coverage data as part of your automated build process. Jenkins, on the other hand, does an excellent job of reporting on the code coverage metrics, including keeping track of code coverage over time, and providing aggregate coverage across multiple application modules.

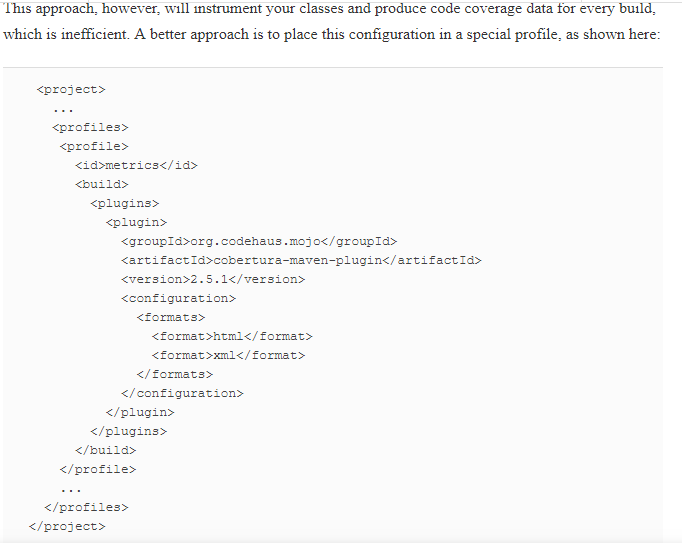
Code coverage can be a complicated business, and it helps to understand the basic process that Cobertura follows, especially when you need to set it up in more low-level build scripting tools like Ant. Code coverage analysis works in three steps. First, it modifies (or “instruments”) your application classes, to make them keep a tally of the number of times each line of code has been executed.[[3](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch06s06.html#ftn.id2621162)]They store all this data in a special data file (Cobertura uses a file called cobertura.ser).

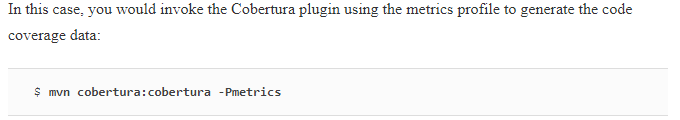
When the application code has been instrumented, you run your tests against this instrumented code. At the end of the tests, Cobertura will have generated a data file containing the number of times each line of code was executed during the tests.

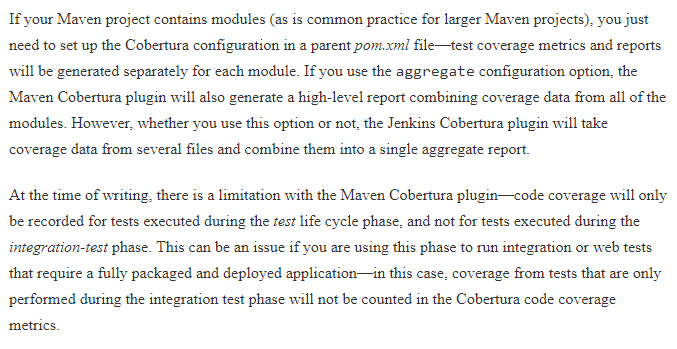
Once this data file has been generated, Cobertura can use this data to generate a report in a more usable format, such as XML or HTML.

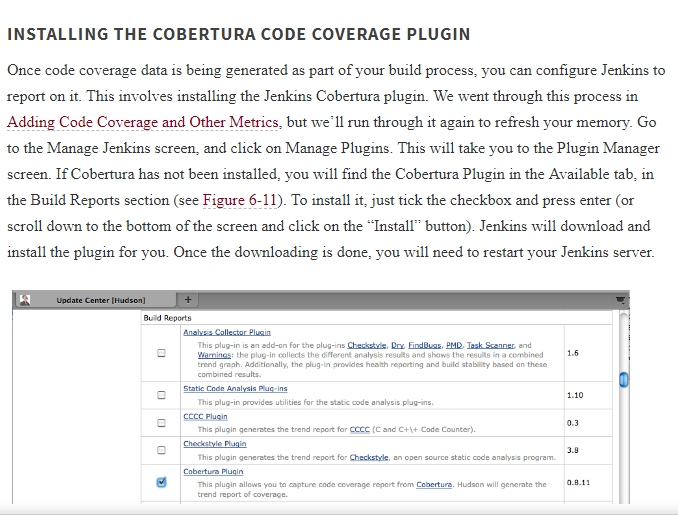






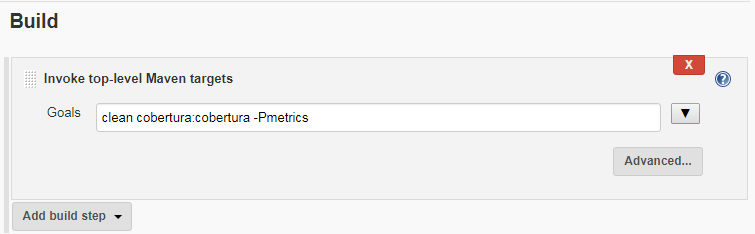


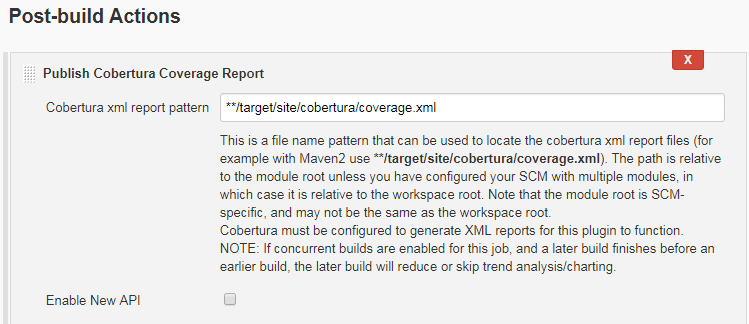




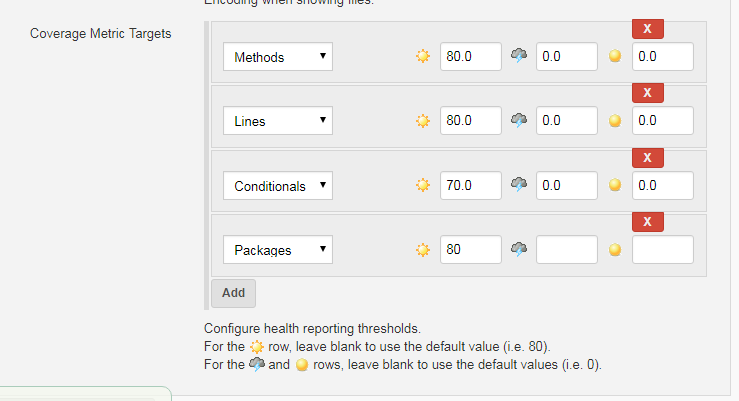
### REPORTING ON CODE COVERAGE IN YOUR BUILD

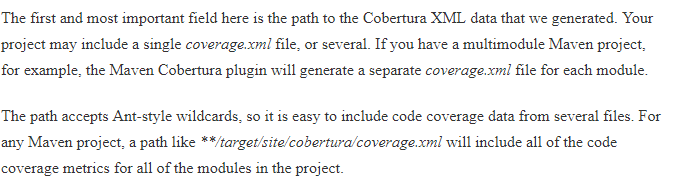
Once you have installed the plugin, you can set up code coverage reporting in your build jobs. Since code coverage can be slow and memory-hungry, you would typically create a separate build job for this and other code quality metrics, to be run after the normal unit and integration tests. For very large projects, you may even want to set this up as a build that only runs on a nightly basis. Indeed, feedback on code coverage and other such metrics is usually not as time-critical as feedback on test results, and this will leave build executors free for build jobs that can benefit from snappy feedback.

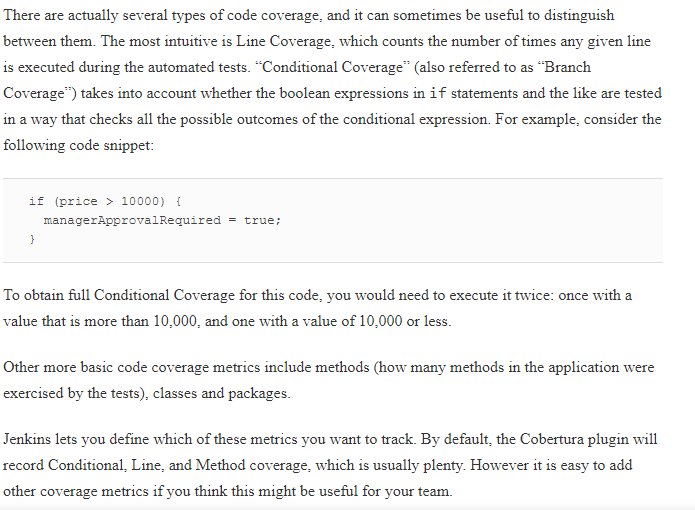




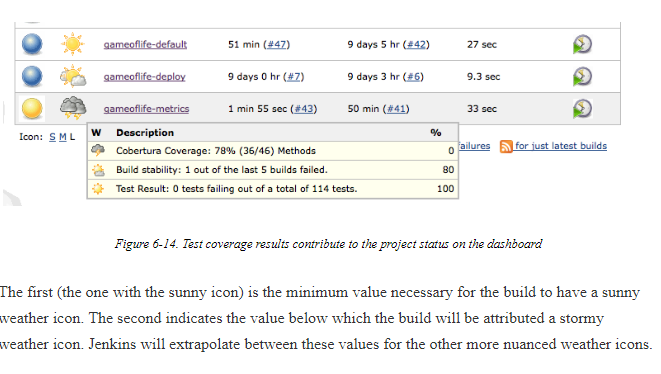
Once you have configured your build to produce some code coverage data, you can configure Cobertura in the “Post-build Actions” section of your build job. When you tick the “Publish Cobertura Coverage Report” checkbox, you should see something like above

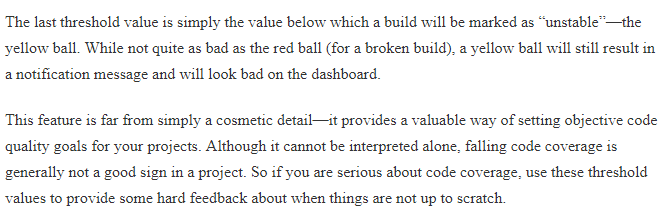


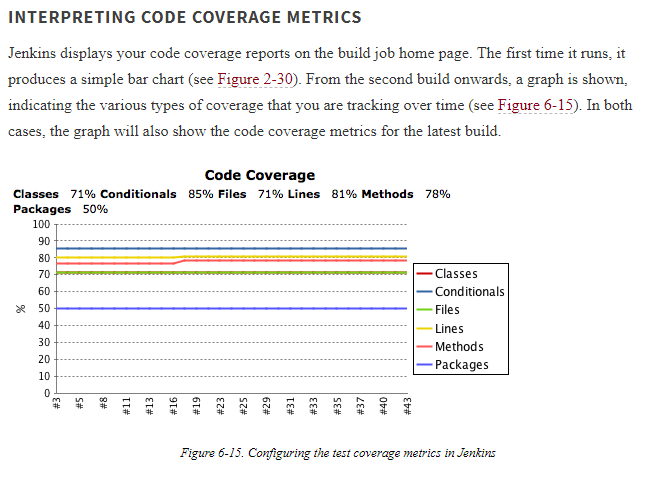


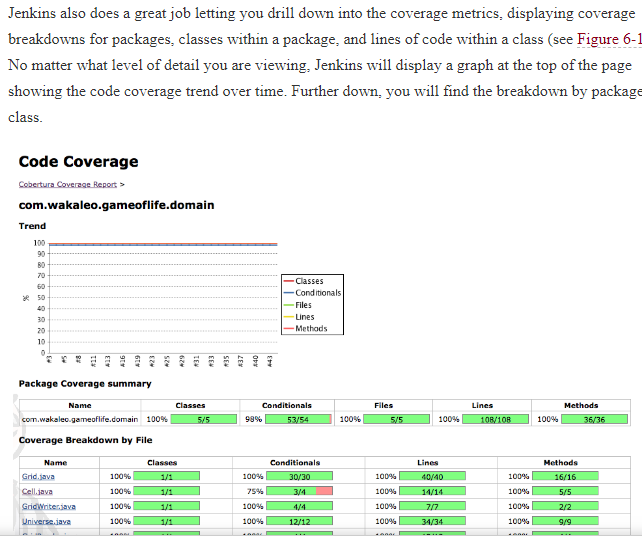


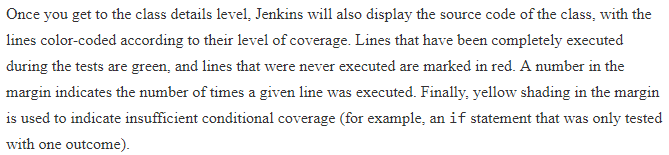
Jenkins code quality metrics are not simply a passive reporting process—Jenkins lets you define how these metrics affect the build outcome. You can define threshold values for the coverage metrics that affect both the build outcome and the weather reports on the Jenkins dashboard (see [Figure 6-14](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch06s06.html#fig-hudson-testing-coverage-stabiliy)). Each coverage metric that you track takes three threshold values.











# Automated Acceptance Tests

Automated acceptance tests play an important part in many agile projects, both for verification and for communication. As a verification tool, acceptance tests perform a similar role to integration tests, and aim to demonstrate that the application effectively does what is expected of it. But this is almost a secondary aspect of automated Acceptance Tests. The primary focus is actually on communication—demonstrating to nondevelopers (business owners, business analysts, testers, and so forth) precisely where the project is at.

Acceptance tests should not be mixed with developer-focused tests, as both their aim and their audience is very different. Acceptance tests should be working examples of how the system works, with an emphasis on demonstration rather than exhaustive proof. The exhaustive tests should be done at the unit-testing level.

Acceptance Tests can be automated using conventional tools such as JUnit, but there is a growing tendency to use Behavior-Driven Development (BDD) frameworks for this purpose, as they tend to be a better fit for the public-facing nature of Acceptance Tests. Behavior-driven development tools used for automated Acceptance Tests typically generate HTML reports with a specific layout that is well-suited to nondevelopers. They often also produce JUnit-compatible reports that can be understood directly by Jenkins.

Behavior-Driven Development frameworks also have the notion of “Pending tests,” tests that are automated, but have not yet been implemented by the development team. This distinction plays an important role in communication with other non-developer stakeholders: if you can automated these tests early on in the process, they can give an excellent indicator of which features have been implemented, which work, and which have not been started yet.

As a rule, your Acceptance Tests should be displayed separately from the other more conventional automated tests. If they use the same testing framework as your normal tests (e.g., JUnit), make sure they are executed in a dedicated build job, so that non-developers can view them and concentrate on the business-focused tests without being distracted by low-level or technical ones. It can also help to adopt business-focused and behavioural naming conventions for your tests and test classes, to make them more accessible to non-developers (see [Figure 6-19](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch06s07.html#fig-hudson-junit-acceptance-tests)). The way you name your tests and test classes can make a huge difference when it comes to reading the test reports and understanding the actual business features and behavior that is being tested

Remaining part not good at all

Performance test read from the book if required

# Help! My Tests Are Too Slow!

One of the underlying principles of designing your CI builds is that the value of information about a build failure diminishes rapidly with time. In other words, the longer the news of a build failure takes to get to you, the less it is worth, and the harder it is to fix.

Indeed, if your functional or integration tests are taking several hours to run, chances are they won’t be run for every change. They are more likely to be scheduled as a nightly build. The problem with this is that a lot can happen in twenty-four hours, and, if the nightly build fails, it will be difficult to figure out which of the many changes committed to version control during the day was responsible. This is a serious issue, and penalizes your CI server’s ability to provide the fast feedback that makes it useful.

Of course some builds are slow, by their very nature. Performance or load tests fall into this category, as do some more heavyweight code quality metrics builds for large projects. However, integration and functional tests most definitely do not fall into this category. You should do all you can to make these tests as fast as possible. Under ten minutes is probably acceptable for a full integration/functional test suite. Two hours is not.

So, if you find yourself needing to speed up your tests, here are a few strategies that might help, in approximate order of difficulty.

## Run Fewer Integration/Functional Tests

In many applications, integration or functional tests are used by default as the standard way to test almost all aspects of the system. However integration and functional tests are not the best way to detect and identify bugs. Because of the large number of components involved in a typical end-to-end test, it can be very hard to know where something has gone wrong. In addition, with so many moving parts, it is extremely difficult, if not completely unfeasible, to cover all of the possible paths through the application.

For this reason, wherever possible, you should prefer quick-running unit tests to the much slower integration and functional tests. When you are confident that the individual components work well, you can complete the picture by a few end-to-end tests that step through common use cases for the system, or use cases that have caused problems in the past. This will help ensure that the components do fit together correctly, which is, after all, what integration tests are supposed to do. But leave the more comprehensive tests where possible to unit tests. This strategy is probably the most sustainable approach to keeping your feedback loop short, but it does require some discipline and effort.

## Run Your Tests in Parallel

If your functional tests take two hours to run, it is unlikely that they all need to be run back-to-back. It is also unlikely that they will be consuming all of the available CPU on your build machine. So breaking your integration tests into smaller batches and running them in parallel makes a lot of sense.

There are several strategies you can try, and your mileage will probably vary depending on the nature of your application. One approach, for example, is to set up several build jobs to run different subsets of your functional tests, and to run these jobs in parallel. Jenkins lets you aggregate test results. This is a good way to take advantage of a distributed build architecture to speed up your builds even further. Essential to this strategy is the ability to run subsets of your tests in isolation, which may require some refactoring.

At a lower level, you can also run your tests in parallel at the build scripting level. As we saw earlier,both TestNG and the more recent versions of JUnit support running tests in parallel. Nevertheless, youwill need to ensure that your tests can be run concurrently, which may take some refactoring. For example, common files or shared instance variables within test cases will cause problems here.

In general, you need to be careful of interactions between your tests. If your web tests start up an embedded web server such as Jetty, for example, you need to make sure the port used is different for each set of concurrent tests.

Nevertheless, if you can get it to work for your application, running your tests in parallel is one of the more effective way to speed up your tests.