

THE WISE DEVELOPERS' GUIDE TO STATIC CODE ANALYSIS

FEATURING FINDBUGS, CHECKSTYLE, PMD,
COVERITY AND SONARQUBE

How many bugs will you find?



TABLE OF CONTENTS

CHAPTER I

WELCOME TO STATIC CODE ANALYSIS,
THAT THING YOU AREN'T DOING

1-10

*Click to go
to section*
↙

CHAPTER II

SETTING UP AND RUNNING FINDBUGS,
PMD, CHECKSTYLE AND COVERITY

11-18

CHAPTER III

MAKING SENSE OF IT ALL
(AKA, THE "10000-ERROR BLUES")

19-27

CHAPTER IV

FINAL ADVICE AND A GOODBYE COMIC :-)

28-31

CHAPTER I:

WELCOME TO STATIC CODE ANALYSIS, THAT THING YOU AREN'T DOING

"The quality of your code is a weak spot in almost every software project you'll ever touch. This is because ongoing development ensures that even the bits you were once proud of become, over time, first less elegant, then rough, and finally incomprehensible."

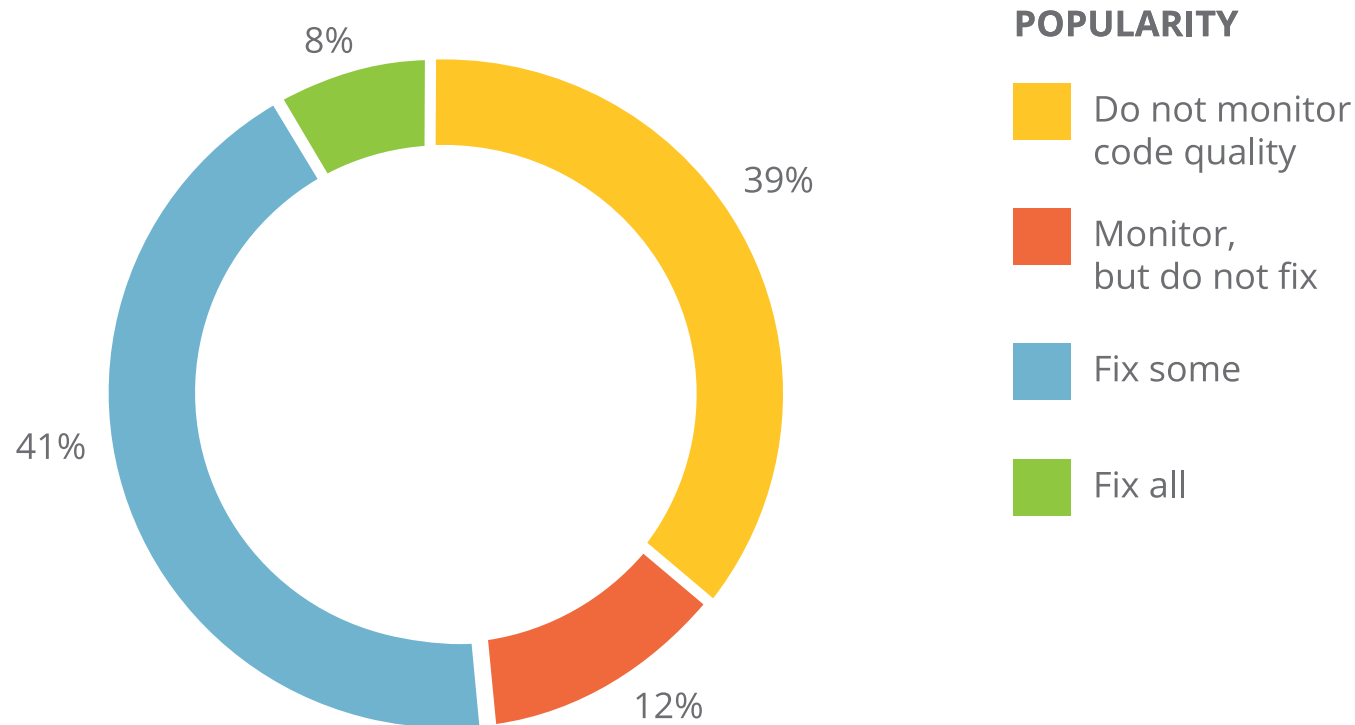


OLEG SHELAJEV,
Java Developer/Author

Why should we monitor and fix code quality issues?

If we start at the very beginning, it would be with what we know about developers and their use of the tools and practices used to analyze code quality. There are a few things we found out about how developers think about code quality analysis, from [Developer Productivity Report 2013 – How Engineering Tools & Practices Impact Software Quality & Delivery](#), which surveyed just over 1000 developers. Here's what we saw:

DO YOU MONITOR AND FIX CODE QUALITY PROBLEMS? (e.g. with Sonar)

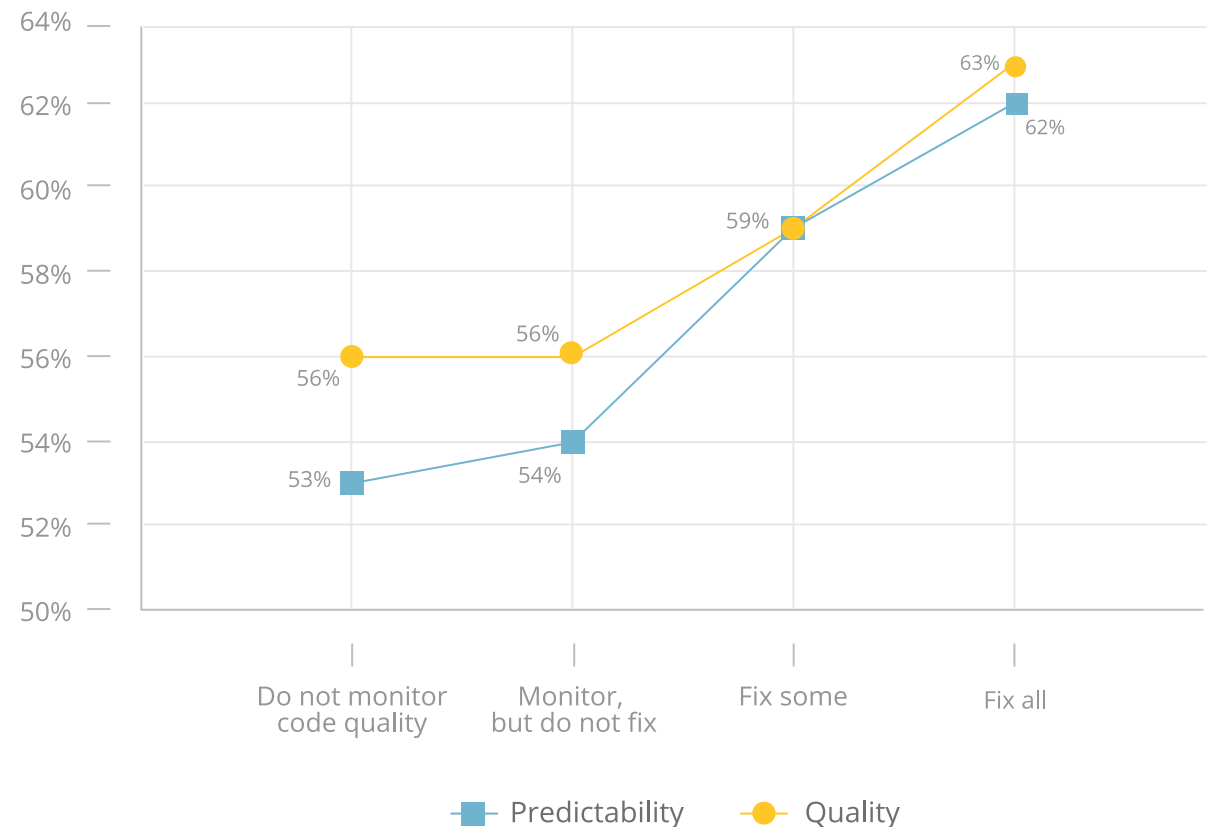


So, code quality analysis is not a terribly popular category to start with--which is probably one reason that most apps, even the best of them, suffer from creeping bugs and errors at some point. And do you know what else we found when we correlated these answers with others in the report?

Fixing code quality issues has a significant effect on, well, the overall quality of your code, as well as your ability to accurately predict when the software can be delivered to end users. Boom!

If the choice is between doing nothing and fixing all code quality issues that are identified, this is the analysis that emerged from the responses provided by the sample population:

The effects of fixing code quality issues on predictability and quality



Developers reported up to **9% better predictability** for app delivery
Developers reported up to **7% better app quality**

The point here is, monitoring and fixing code quality issues is something that is **proven to raise the quality of your application AND your ability to deliver that application to stakeholders on time**. But it's clear that the vast majority of developers aren't taking full advantage of tools designed to improve app quality.

Perhaps most developers don't know where to start. For developers, the main point can be summed up in one sentence:

How are you supposed to integrate your tool of choice into your development cycle so it can find relevant issues and allow the team to fix them?

There are many aspects of "code quality" that we can sink our teeth into, but we've decided that Static Code Analysis is an essential building block in your pyramid of tools that help improve the quality of your code. However, developers are using tools that fit into other categories as well, such as:

DYNAMIC CODE ANALYSIS

The simplest difference between Static and Dynamic analysis tools is that the former runs in the development environment and the latter needs to be active during the runtime of the application under analysis. Typical dynamic code analyzers profile your system and monitor its health. Both execution time and memory usage profilers, figuring out number of database transactions per request, the average size of an user session object, etc. require the system to be under a load comparable with the intended in production environment. Dynamic analysis tools often instrument the code

to add tracing of method calls, catching and notifying about exceptions, and any other statistics they collect.

PROFILERS

Performance is a magical term that never fails to generate interest. Figuring out why your system is slow and how to make it faster is a rewarding exercise. Combine this with the fact that you can continue optimizing forever (as something will always be a bottleneck), performance-related tasks are always picked first by developers. It just sounds so cool, and it's also measurable too.

MEMORY TOOLS

Most of existing tools that deal with memory management either provide some high level statistics in real-time, like telling you the size of the heap and the number of classes loaded into the JVM, or work in an offline mode feeding on some traces produced during a run. Garbage collector's logs, object allocation rates, ability or inability to refresh the memory taken by the classloader of your web-application, these are questions usually attacked with a tool analyzing your application's memory behavioral patterns.

MONITORING TOOLS

Monitoring tools are known to everybody, often these are the last man standing before a service goes offline because of some resource limits.

Naturally, there are a lot of questions to ask before you start to use any of the tools we discuss later, so in this report we show you what aspects are important to consider when getting started.

Static code analysis tools for source code, byte code and the big picture

Since an application's code can greatly vary, and every program can be written in lots of ways without being semantically different, most tools use some kind of a probabilistic approach, usually based on pattern matching, to determine which pieces of code should be reviewed. These hints are a real time-saver, helping you to review incoming changes and prevent bugs from propagating into the released artifacts. Here are a few categories of static code analysis that development teams can consider:

- **Source code analyzers** - CheckStyle, PMD, Coverity
- **Bytecode analyzers** - FindBugs, JLint
- **High-level project analyzers** - SonarQube, Atlas, SonarGraph-Quality

We'll go into that more later. Now, before choosing a code analysis tool, you should be able to answer the following questions:

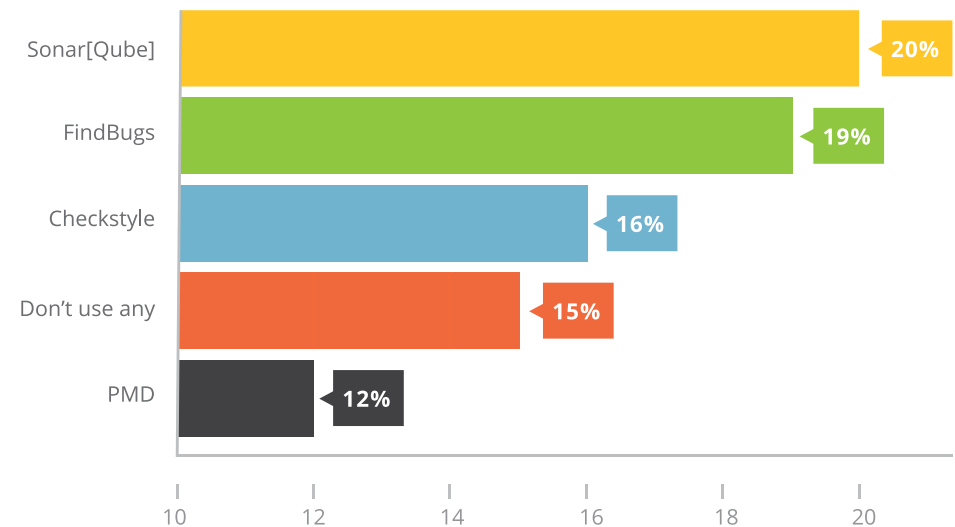
- What types of issues do you want to detect (i.e. security flaws, code style, incompetent or inefficient code, concurrency and design issues)?
- Does your tool require a fully-buildable set of sources and what are the artifacts it operates on (e.g. source code, compiled binary or linked executable)?
- What kind of reporting do you need provided: integration with IDEs, CI tools or some kind of central hub?

These simple questions are sometimes the hardest to figure out, so that is what we focus on in this report: the value you can get from setting up code analysis tools and things to look out for along the way.

Obviously, we can't look at every tool in existence, so we'll focus only on the tools that we have experience with and/or more than 10% of developers actually use: SonarQube, FindBugs, Checkstyle and PMD--all open source--plus Coverity, which represents the commercial code analysis tool space. Oh, do notice that the 4th most used method for static code analysis is NOTHING AT FREAKIN' ALL, so if it wasn't clear before now that we all have a long way to go, it should be now :-)

STATIC CODE ANALYSIS TOOL USAGE BY DEVELOPERS

(SAMPLE SIZE: 2119)





SONARQUBE

License: LGPL

Current version: 4.2 (3.7 - long-term support)

URL: <http://www.sonarqube.org/>

SonarQube is a web-application based platform to manage most of your static analysis tools. It doesn't do the low-level analysis itself, but instead integrates with tools like FindBugs, Checkstyle and PMD out-of-the-box, or with provided plugins. It then acts as a central hub for all your code analysis tools, providing you with historical insight and trend analysis for multiple projects at the same time.

We actually use this tool at ZeroTurnaround, and it gives you a great overview over your projects. It's also easy to set up and does not intrude on your build process. The all-in-one-place analytics are great to have even if you won't act on the reports it sends you. Just having access to historical data about the complexity of your code and the number of issues your tools spot in your code can show you if you're doing something right.



FINDBUGS

License: LGPL

Current version: 2.0.3 (released Nov 2013)

URL: <http://findbugs.sourceforge.net/>

FindBugs was the most-used tool in the Code Quality space according to our survey results. Created by the University of Maryland, it actually scans your code for bugs, breaking down the list of bugs in your code into a ranked list on a 20-point scale. The lower the number, the more hardcore the bug. While FindBugs also performs a cursory check on best practices, PMD is better at this and so they are often used in combination.



CHECKSTYLE

License: LGPL

Current version: 5.7 (released Feb 2014)

URL: <http://checkstyle.sourceforge.net/>

If you remember Checkstyle from past days, you'd see that it has grown from a tool that yells at your code layout issues & unused imports into a full suite that can spot class design errors, help you enforce a homogenous code style across the whole development team and compute code complexity metrics outlining where the functional bugs are likely to be found.

Checkstyle features an Ant-task or a command line call as the main methods to run the checks, although there is a Maven plugin and multiple plugins for Eclipse, IntelliJ IDEA, NetBeans, and even Emacs and Vim are included.



PMD

License: a mix of BSD-style license and Apache License version 2.0

Current version: 5.1.0 (released Feb 2014)

URL: <http://pmd.sourceforge.net/>

PMD is a source code analyzer that finds common flaws like unused objects, empty blocks, unnecessary catches, incomprehensible namings and so forth. It uses "rulesets" allow you to check your code for almost every wrong language use pattern you can come up with, and copy-paste detection to make it easier for you to follow the best practices of code reuse.

PMD is an active project and the latest release added compatibility with Java 8 language constructs like lambdas and default methods. It also is worth mentioning that PMD works not only on Java code, but JavaScript, PLSQL and various frameworks are supported. Chances are that you can check your whole project in one swoop.



COVERITY CODE ADVISOR

License: Commercial (that's right, you gotta pay for this one!)

Current version: 7.0.0 (released Dec 2013)

URL: <http://www.coverity.com/products/code-advisor/>

Coverity has made it's way to the tough enterprise world with their Code Advisor tool, for which there is no free licensing option available. They are an established static code analysis tool maker in the C/C++ and Java world, claiming to be the leader in development testing. The Coverity Code Advisor is a combination of Coverity Quality Advisor and Coverity Security Advisor, and also incorporates FindBugs as one of it's key components bundled.

Once you've collect intermediate results of your project, you can upload everything to the Coverity website for some deeper analysis. Your results are presented through a easy-to-navigate web interface and issues are tracked.

Time for some analysis... let's use Jetty!

The question of code quality is an edgy one in the community. Usually, the most vocal developers are doing everything according to the best practices, use the fanciest or the most expensive IDE and all productivity tools that our software evolution has come up with. However while these people are no doubt productive and valuable, there are also others who don't shout about being on the bleeding edge of technology, but nonetheless contribute to their projects and are worth thinking about.

We do see the value in the static code analysis and don't want this report to become just a list of tools that one could use. We want to show for real what benefits the analysis can bring to your project. Thus we decided not to speak theoretically about what types of errors can be found in a project that real developers would conceivably work on, not Spring's Pet Clinic sample application!

So we picked Jetty.



WHY JETTY?

Our basic requirements for the example app we would choose for illustrative purposes in this report were that it should be a familiar, open source project, not too small, but not too big, and also complicated enough to represent the real-life complexity of a project.

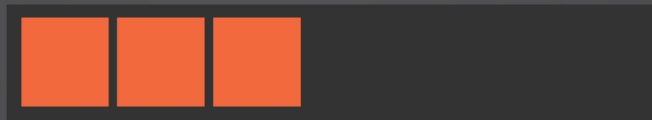
Jetty is a lightweight Java application server maintained by Eclipse Foundation. One of the main benefits of Jetty is its small size and embeddability, which open doors into many applications where a full-blown Java EE-certified server wouldn't quite fit right. In recent surveys by RebelLabs and Typesafe, Jetty use is reported by anywhere from 11 - 49% of developers, either embedded somewhere or as their primary app server.

As a project, Jetty is pretty cool: small enough to be comprehensible by our team writing this report, but definitely not trivial. It's an active project used by many, it has a permissive license and its source code is nicely available from [GitHub](#). It is also a Maven project as well, so if you're using Maven (64% of you reading this are!) then your project's build process--which we're trying to generally improve with code analysis--will share similarities with Jetty's build process.

Ok, we've covered most of the basics, so let's move forward! In the upcoming chapters, we discuss how to install and integrate your tool of choice, how to avoid getting 10,000 errors just 5 minutes after starting up, common pitfalls to watch out for, and more. We even show you ZeroTurnaround's own SonarQube installation and which plugins we use to test JRebel and LiveRebel.

THE DARK SIDE INVENTED JAVA REDEPLOYS

CHHHRR CHUUURRR



JRebel

SEE UPDATES
INSTANTLY
USE JREBEL YOU MUST

* HRRMMMM... *

SPONSORED BY: TEAMS THAT USE JREBEL

TRY IT FOR FREE!

CHAPTER II:

SETTING UP AND RUNNING FINDBUGS, PMD, CHECKSTYLE AND COVERITY

"If you set up Checkstyle with a basic set of rules defining your code style (and run it as part of your build), you spend a lot less time arguing about the placement of brackets and whitespace, and more time talking about actual software design."



TRISHA GEE,
Java Engineer at MongoDB

Installation and setup with FindBugs, Checkstyle, PMD and Coverity (SonarQube comes later)

Setting up a code quality analysis tool is sometimes a somewhat cumbersome task. Luckily, there are Maven plugins available for almost all of them: FindBugs, Checkstyle and PMD. Coverity does not have any Maven plugin, but it can be still used together with Maven. And don't worry...these tools can be executed quite easily without the Maven plugin as well.

We took the Jetty 9 source code and proceeded to set up those tools--Jetty is a fairly large and complicated project, containing over 30 different modules and tons of source code. To start off with running each tool, we modified the main **pom.xml** of this project. Luckily, there were already configurations in place for PMD and FindBugs, so all we needed to do was run these tools.

As each tool is actually looking for different types of problems in the source code and uses different rules for finding problems, we will see a variance in execution times and results.

FINDBUGS

As the name says - FindBugs is looking for bugs. It does some deeper analysis than most tools. We used the [FindBugs-Maven plugin](#), which worked well out of the box:

```
mvn findbugs:findbugs
```

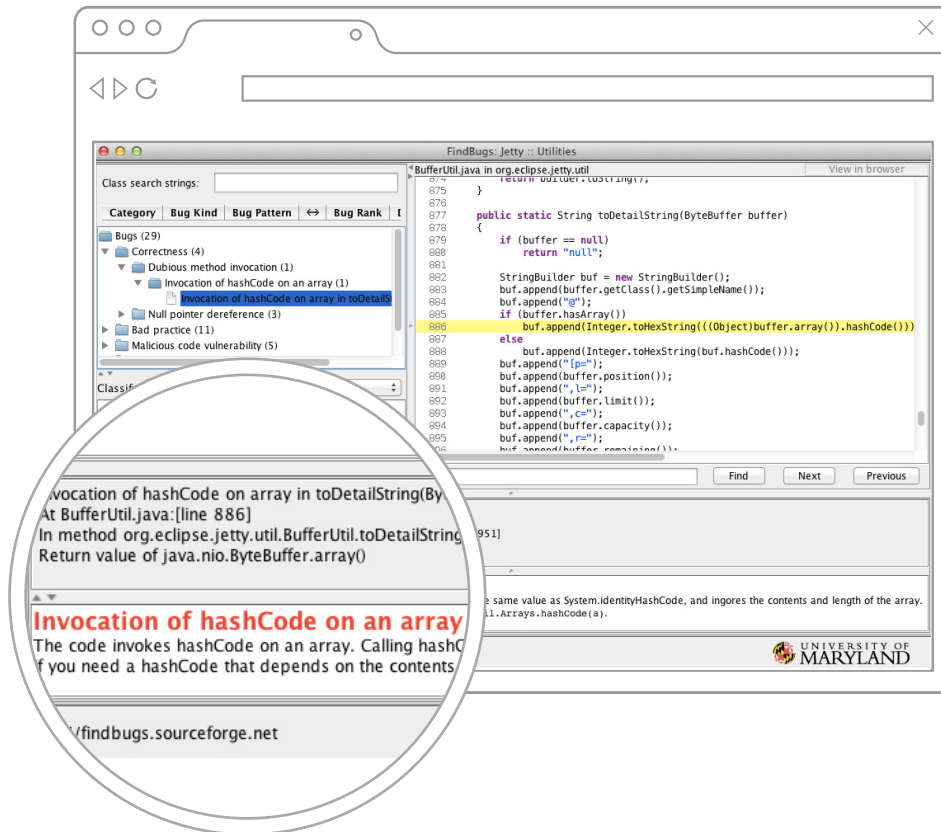
FindBugs was the slowest of the tools we tested, but for a good reason: it performs a really deep investigation of the code, going beyond coding style issues to look at actual errors in programming logic and other possible dangers (which included its share of false positives).

```
[INFO] Total time: 8:40.448s
```

FindBugs generates 2 files for results: **findbugs.xml** and **findbugsXml.xml**. With multi-module projects, these files are generated separately for each module. The good thing about FindBugs is its ability to actually map all results together with the source code, which other tools don't do as well out of the box, making it really easy to spot bugs and problems in your code.

Each problem identified also contains a fairly thorough description about the issue and a possible fix. FindBugs is a strong instructional tool for learning how to code and to do it properly, as it turns out. If a module contains any issues, a file called **findbugsXml.xml** is generated. These XML files can be easily investigated with the FindBugs GUI itself:

```
mvn findbugs:gui
```



Keep in mind that with multi-module projects, a separate **findbugs.xml** is generated for each module, so you need to actually analyze each file.

Results:

FindBugs found a lot of small optimization possibilities, mostly using the static *valueOf* method instead of *Number* constructors. This is actually a good tip; too many misuses may actually slow things down a little bit. Also lot of problems that are actually not real problems, like printing out variables that may be *null* or ignoring exceptional return values by methods that call other methods. There were also some places where method call passed null for non-null parameter. Eventually this all leads to source code, that is kind of hard to debug and understand - so the tool did some good work here.

CHECKSTYLE

Checkstyle checks the “style” of your code. It doesn’t actually find any serious bugs but it helps you make sure that your source code is well-formatted and follows good standards and practices. The main purpose of the tool is to make sure that the code adheres to a coding standard.

Checkstyle allows you to also specify rules for checking code, but more on that later. We installed the [Checkstyle-Maven plugin](#):

```
mvn checkstyle:checkstyle
```

Checkstyle was fairly quick to install:

```
[INFO] Total time: 3:38.919s
```

Running Checkstyle with Maven is fairly easy, requiring no special configuration or surprises: just run the command and you will get the report. Unfortunately, no matter how we tried, the report was kind of broken. Missing stylesheets and images made the report look a bit ugly, but it was still well readable and easy to navigate.

The main page of the HTML report contains the list of files where problems were found and the list of rules as well. As Checkstyle tries to find problems related to coding style, then of course there were a lot of problems. Mostly things on the wrong line, missing “final” keywords, missing whitespace after the comma etc. Basically the tool shows really well how careful developers are, how they use code formatters and follow coding standards. But for legacy projects like this, running a code formatter would be disastrous for its versioning history.



File	I	W	E
org/eclipse/jetty/http/spi/DelegatingThreadPool.java	0	0	74
org/eclipse/jetty/http/spi/HttpSpiContextHandler.java	0	0	67
org/eclipse/jetty/http/spi/JettyExchange.java	0	0	3
org/eclipse/jetty/http/spi/JettyHttpContext.java	0	0	45
org/eclipse/jetty/http/spi/JettyHttpExchange.java	0	0	93
org/eclipse/jetty/http/spi/JettyHttpExchangeDelegate.java	0	0	91
org/eclipse/jetty/http/spi/JettyHttpServer.java	0	0	147
org/eclipse/jetty/http/spi/JettyHttpServerProvider.java	0	0	21
org/eclipse/jetty/http/spi/JettyHttpsExchange.java	0	0	68

Results: Checkstyle found 609 errors, and indicated that Jetty’s code should be formatted more often. A lot of the code is actually not very well-formatted, including unneeded and missing spaces, problems with curly braces, etc. It looks like not all developers are using the same code formatter or these problems have come into Jetty in the past.

PMD

PMD is a very customizable tool, so the performance and installation outcome really depends on how one has instructed the tool to find problems. PMD has its own built-in rulesets, but it suggests to have your own rulesets relevant to the project, developers and frameworks. Having a framework-related ruleset is actually a good thing, because it helps you find problems that other tools cannot, like framework-specific configuration threats and issues. PMD is like a team-specific automatic code review tool in a sense, as it helps spot problems that a team thinks are important for their project.

Finding a good set of rules for PMD to run is not a trivial task. Each ruleset is very narrow and does not provide full features alone: basically for every mistake that FindBugs can show, there is one ruleset for PMD.

In fact, the [PMD-Maven plugin](#) didn't work as expected, so to make things easier we created the following scripted called "`check_jetty.sh`". The script just runs PMD with different rulesets on the same project and outputs ruleset results to a different HTML file.

```
#!/bin/bash

PMD_HOME="/Users/sigmar/Desktop/pmd-bin-5.1.0"
SOURCE_FOLDER="/Users/sigmar/Documents/jetty"
FORMAT="html"

echo "Finding unused code"
$PMD_HOME/bin/run.sh pmd -d $SOURCE_FOLDER -f $FORMAT -R rulesets/
java/unusedcode.xml > unusedcode.html
echo "Saved results to unusedcode.html"
```

```
echo "Doing basic checks"
$PMD_HOME/bin/run.sh pmd -d $SOURCE_FOLDER -f $FORMAT -R rulesets/
java/basic.xml > basic.html
echo "Saved results to basic.html"

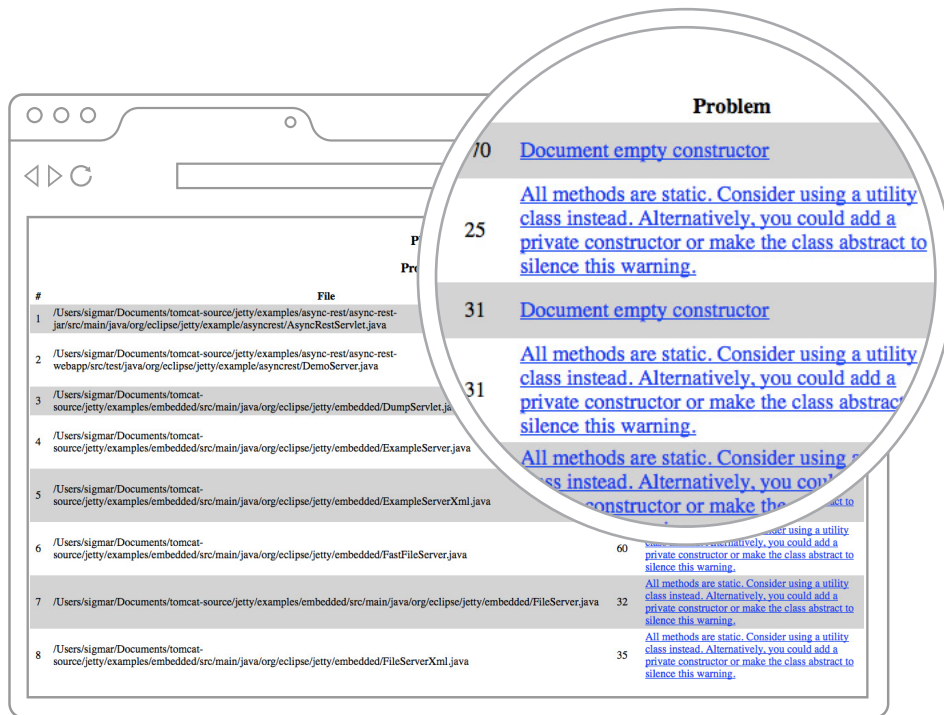
echo "Checking braces"
$PMD_HOME/bin/run.sh pmd -d $SOURCE_FOLDER -f $FORMAT -R rulesets/
java/braces.xml > braces.html
echo "Saved results to braces.html"

echo "Checking design"
$PMD_HOME/bin/run.sh pmd -d $SOURCE_FOLDER -f $FORMAT -R rulesets/
java/design.xml > design.html
echo "Saved results to design.html"
```

We timed the command and created four different reports for each ruleset:

```
sigmars-air:bin sigmar$ time ./check_jetty.sh
Finding unused code
Saved results to unusedcode.html
Doing basic checks
Saved results to basic.html
Checking braces
Saved results to braces.html
Checking design
Saved results to design.html

real    1m45.483s
user    2m53.733s
sys      0m6.297s
```



When checking the design of Jetty with PMD, many issues popped up. The most frequent problems were too-deeply nested if-statements, weird variable names, misuse of JDK APIs, switches with less than three statements, fields not declared on top of the class and also empty public constructors. It appears as though there is some smelly code still in this codebase.

Results: A total of 13200 problems?! All problems are described nicely in the HTML report with information about the source file and problem's location. The appearances of empty catch blocks, missing keywords, weird variable names, if-constructs without curly braces, empty public constructors and others were a bit suspicious.

COVERITY

Getting Coverity up and running was not so straightforward--as a commercial tools organization, their website is set up to funnel you into the warm embrace of their sales team, and being that way it took some time to realize what to do and how everything works. Here are the steps:

1. Register yourself as a Coverity user
2. Go to the "Free trial" section
3. Download the build analysis tools from the website
4. Run Coverity build analysis on your project's build results.

Thankfully, guys from Coverity were helpful and provided some guidelines and additional help how to get their stuff running; now, the fact that we needed to ask for it was slightly disconcerting at first, but then we realized that being able to call a representative and interact with someone if needed is exactly what organizations are looking for from commercial vendors like Coverity.

Fortunately, the downloaded analysis package was already pre-configured and all we needed to do was to run it on our Jetty-Maven project:

```
cov-build --dir ./cov-int mvn -Dmaven.test.skip=true -DskipTests
-Dmaven.compiler.fork=true install
```

Basically Coverity takes all the build results immediately after the build has finished and analyses them.

Which resulted like this:

```
[INFO] Total time: 5:08.476s
```

After running it all, we got a folder called **cov-int**, which we needed to compress to some well-known archive format (we compressed it as "zip"). After compressing the analysis and build results, we needed to upload the package to Coverity website:

Trial Project

Jetty 9

Upload complete! Your project has been queued for analysis and we will notify you by e-mail when your results are ready. It can take up to 48 hours for your first analysis to complete.

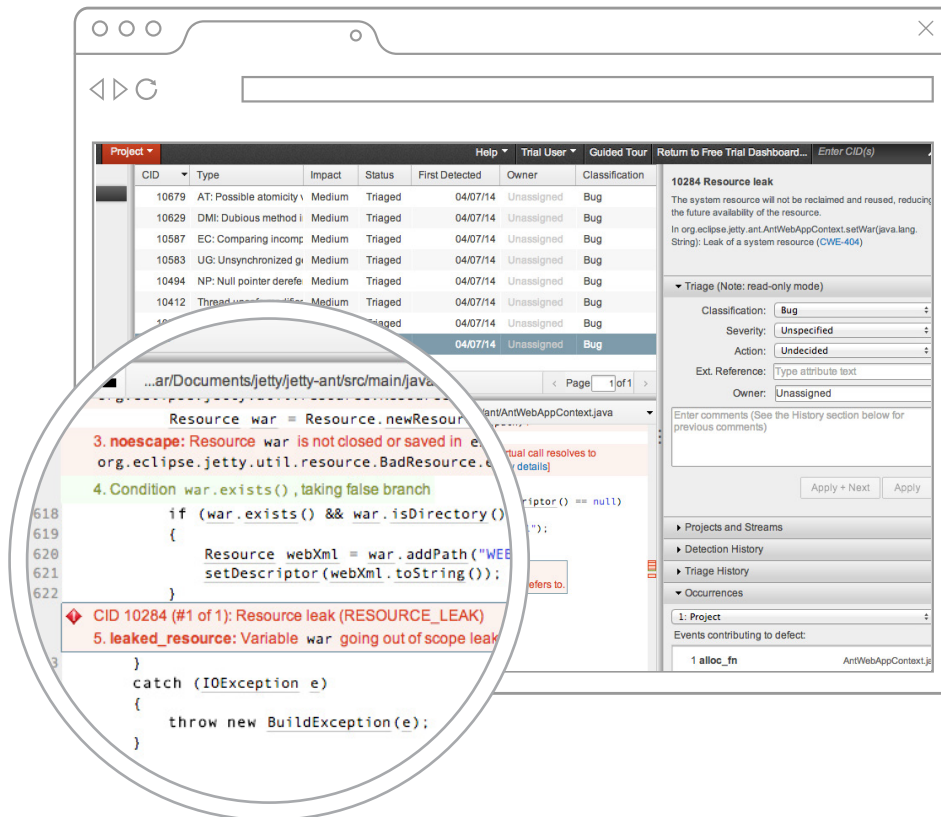
At any time you may [delete your project](#). Your source code and defect data will be [securely erased](#) from our servers.

One thing to keep in mind is that the project was queued for analysis and we had to wait for the notification once it was complete. If you need your results immediately, other tools will provide that feedback more rapidly.

All uploaded projects are listed on the Coverity dashboard, enabling you to view and analyze results any time you like.

What these results tell us...

A benefit of Coverity is that it does a much deeper analysis than other tools and it also keeps track of the issues that you've found and fixed. It works like an automated issue tracker for your bugs, with everything listed and easily navigable using the web-based UI:



Results: With the most detailed coverage, Coverity found 581 defects in total, of which 77 were considered high-risk defects. According to their suggestions, high-impact defects cause crashes, program instability and performance problems.

...is only part of the story.

Now, your next question might be *who is going to fix all these problems?* *Is Jetty's source code really such garbage?* We really doubt so. The issue at heart here is the matter of **strictness of rules defined in these rulesets**-- without any customizations, we wound up with tons of errors and not a lot of reasoning behind them.

Without telling any of these tools what to really look for, we are letting them show every possible error under the sun. In the same way that running no code analysis for your applications at all is too little too late, seeing 10,000+ errors (or false positives) -- that don't really appear to affect the end user's POV all that much -- is receiving too much useless information.

In the coming chapter, we'll show you how to customize your experience with the styles and methodologies in mind that you and your team practice. We'll see how ZeroTurnaround uses SonarQube to bring a lot of code analysis processes together, and show suggestions for FindBugs, CheckStyle and PMD (we will leave Coverity users here, since they are paying for specialists that are better than your humble RebelLabs crew to help them out whenever they need). Off we go!

CHAPTER III:

MAKING SENSE OF IT ALL (AKA, THE “10000-ERROR BLUES”)

"For us, SonarQube is a great platform for finding bugs in your code...and much cheaper than having your users do it for you!"



TOOMAS ROMER,
Founder of ZeroTurnaround

What to know about SonarQube (aka Sonar)

A great way to establish an infrastructural process of regularly running code analysis tools on your projects is to install SonarQube and enable various plugins from within it. Because SonarQube is a platform that let's you add different tools and functions to it, we left it out of Chapter 2, since it's more likely for developers to try out individual tools.

First, installation. The latest stable version is available on the [official download page](#), so go here to proceed with installing it. We are going to show how to do that on a local machine, but you most definitely want to daemonize it and run on some server. Steps to follow:

First, let's get set up (we used SonarQube 4.2 on a Mac):

```
shelajev@shrimp ~/apps/ » unzip sonarqube-4.2.zip
shelajev@shrimp ~/apps/ » cd sonarqube-4.2/bin/macosex-universal-64
shelajev@shrimp ~/apps/sonarqube-4.2/bin/macosex-universal-64 » ./
sonar.sh start
Starting sonar...
Started sonar.
```

Next, let's access the web interface of the SonarQube at <http://localhost:9000/>. Then, log in with the default credentials: *admin/admin*. Now we are able to configure dashboards and, more importantly, the plugins and rules that SonarQube makes available to you.

Now we can add Project Analysis to our currently-empty SonarQube installation. The most generic way to enable SonarQube integration on a project is to install and use [SonarQube Runner](#). You will need to configure a properties file that specifies at least your SonarQube installation location and project language:

```
# Required metadata
sonar.projectKey=my:project
sonar.projectName=My project
sonar.projectVersion=1.0
# Path to the parent source code directory.
# Path is relative to the sonar-project.properties file. Replace
"\\" by "/" on Windows.
# Since SonarQube 4.2, this property is optional. If not set,
SonarQube starts looking for source code
# from the directory containing the sonar-project.properties file.
sonar.sources=src
# The value of the property must be the key of the language.
sonar.language=java
# Encoding of the source code
sonar.sourceEncoding=UTF-8
```

Using Maven? That's good! Maven projects enjoy a more convenient way to run SonarQube... in the form of a [Sonar-Maven plugin](#), which gets updated with every SonarQube release. So executing `mvn sonar:sonar` will do the trick.

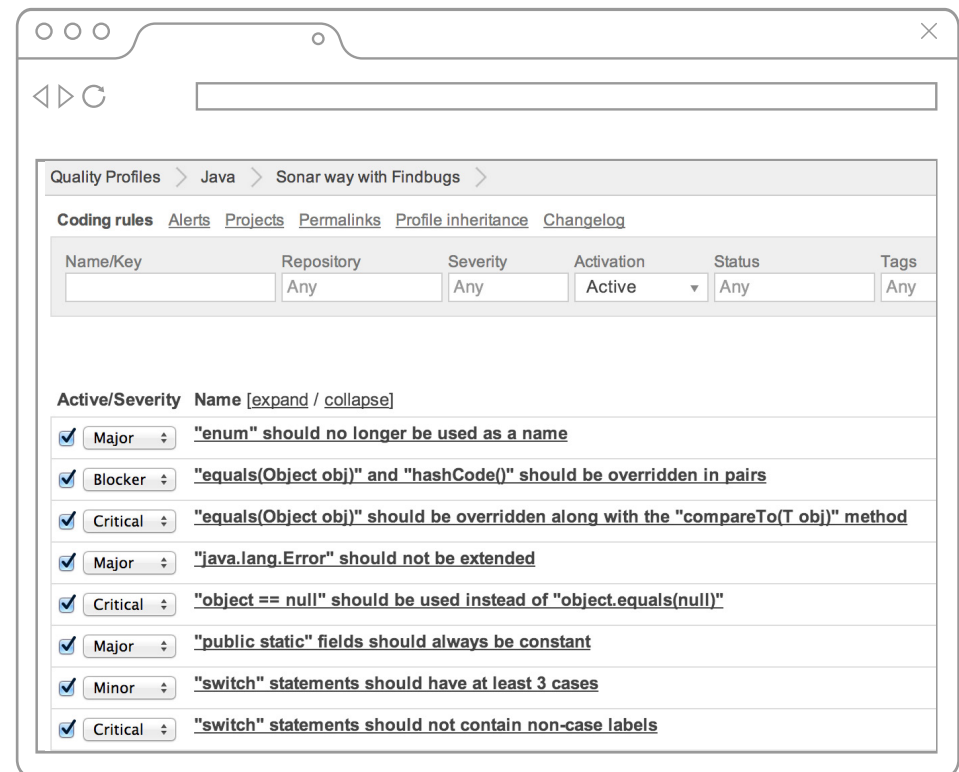
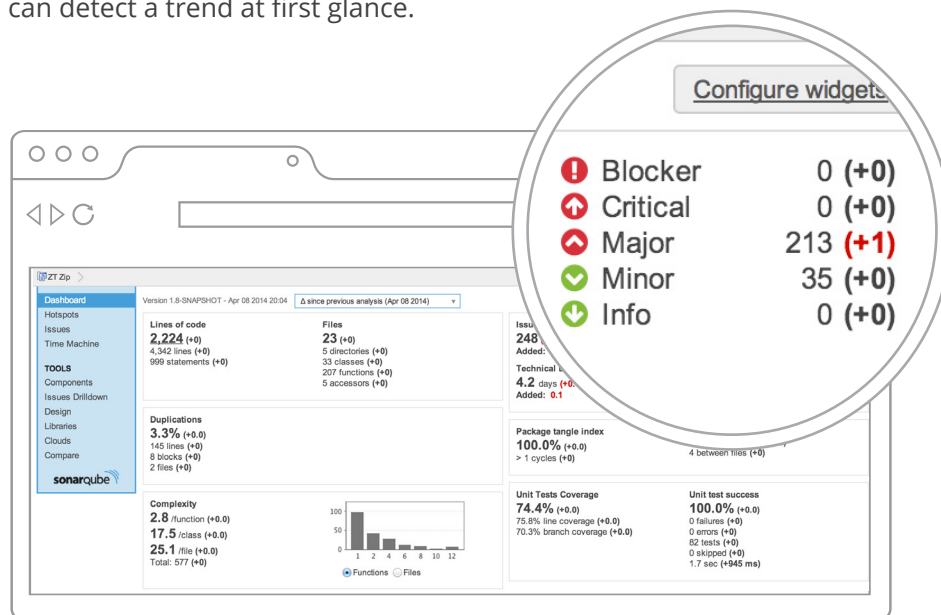
There are a couple of details to remember:

1. Always use the same version of the plugin that your SonarQube instance has.
2. Do not run SonarQube with other Maven commands, it might confuse it and influence the output.
3. Remember that `mvn sonar:sonar` will run the tests, so in order to save time you most probably want to omit it while building the project prior to running the SonarQube plugin.

The Maven plugin requires some additional configuration to establish a connection to the web platform, where it reports the results of the analysis--`sonar.host.url` and `sonar.jdbc.url` will help you with that. Other properties, like `sonar.analysis.mode`, `sonar.profile` and `sonar.dynamicAnalysis` configure the plugin to gather a certain statistics only.

If you are confused now, don't worry: the default values are sane and if you don't have any specific preferences yet, just trust the out-of-the-box configuration. Your upcoming analysis will show you enough information to proceed and you will always have an option to tweak everything later.

Here is an example of a SonarQube's dashboard showing the for another project, not Jetty. You can immediately see that it includes some basic metrics, like lines of code, complexity numbers for your code and an overview of the issues that the analysis encountered. Even better, it shows you the difference in the number of issues since the previous run, so you can detect a trend at first glance.

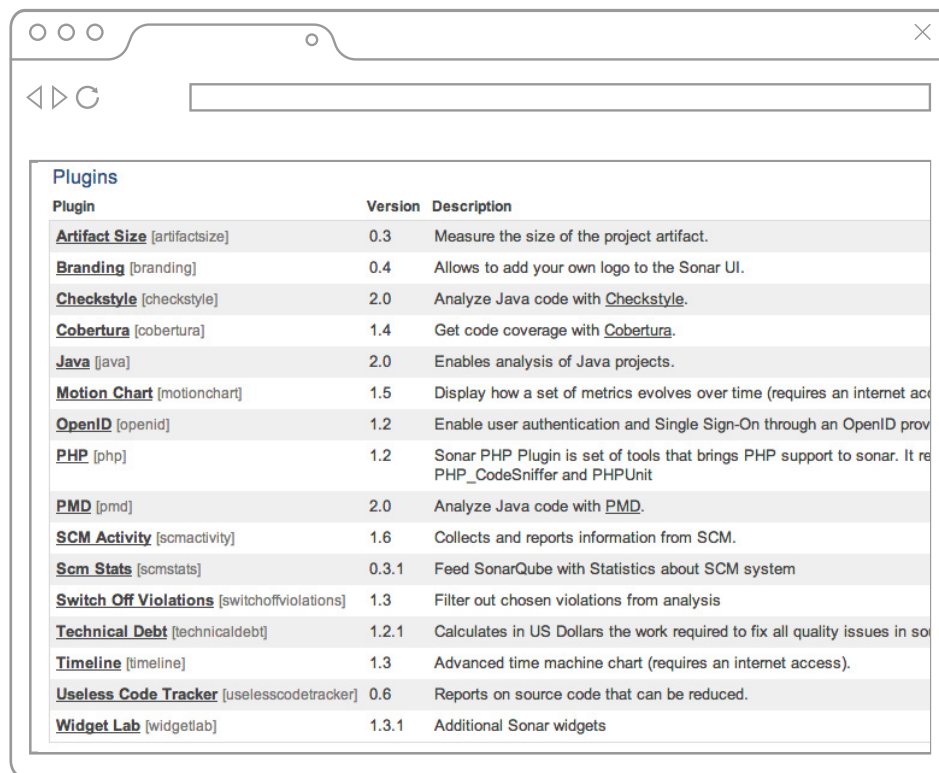


Now you can always log in into your SonarQube instance and configure the profiles. Please note that SonarQube's default profile for Java projects includes FindBugs.

The best thing is that the configuration of individual rules goes through mouse-clicks instead of writing long properties files or including a bunch of XML rulesets. PMD, we're looking at you! ;-)

We mentioned that ZeroTurnaround uses SonarQube for our Java projects JRebel and LiveRebel, as we find that it gives us the best combination of usefulness and transparency against the relative complexity of setting it up and solving weird configuration issues. Here are the plugins we have installed as of the publication of this report, which took a while to acquire as we shaped our needs over time.

As we mentioned before however, a lot of developers will get started with individual tools rather than installing a separate platform first and then adding everything in. So, you shouldn't be afraid of the individual tools. That's all coming up next...

A screenshot of a web browser window showing the 'Plugins' page in SonarQube. The browser has a single tab and a search bar. The page title is 'Plugins'. Below the title is a table with three columns: 'Plugin', 'Version', and 'Description'. The table lists 17 installed plugins, each with a link to its documentation. The plugins are: Artifact Size, Branding, Checkstyle, Cobertura, Java, Motion Chart, OpenID, PHP, PMD, SCM Activity, Scm Stats, Switch Off Violations, Technical Debt, Timeline, Useless Code Tracker, and Widget Lab.

Plugin	Version	Description
Artifact Size [artifactsizesize]	0.3	Measure the size of the project artifact.
Branding [branding]	0.4	Allows to add your own logo to the Sonar UI.
Checkstyle [checkstyle]	2.0	Analyze Java code with Checkstyle .
Cobertura [cobertura]	1.4	Get code coverage with Cobertura .
Java [java]	2.0	Enables analysis of Java projects.
Motion Chart [motionchart]	1.5	Display how a set of metrics evolves over time (requires an internet access).
OpenID [openid]	1.2	Enable user authentication and Single Sign-On through an OpenID provider.
PHP [php]	1.2	Sonar PHP Plugin is set of tools that brings PHP support to sonar. It requires PHP_CodeSniffer and PHPUnit.
PMD [pmd]	2.0	Analyze Java code with PMD .
SCM Activity [scmactivity]	1.6	Collects and reports information from SCM.
Scm Stats [scmstats]	0.3.1	Feed SonarQube with Statistics about SCM system
Switch Off Violations [switchoffviolations]	1.3	Filter out chosen violations from analysis
Technical Debt [technicaldebt]	1.2.1	Calculates in US Dollars the work required to fix all quality issues in sonar.
Timeline [timeline]	1.3	Advanced time machine chart (requires an internet access).
Useless Code Tracker [uselesscodetracker]	0.6	Reports on source code that can be reduced.
Widget Lab [widgetlab]	1.3.1	Additional Sonar widgets

Hints for customizing FindBugs

The main strength of FindBugs is that it is aimed at finding actual bugs in your code. They are ranked automatically according to severity: 1-5 - High, 5-10 - Medium, 10-20 - Low. What is good about this automatic ranking scheme is that it divides the large problem with your codebase into smaller digestible chunks that you can act upon.

This separation by severity is done by experts and is a great starting point is to limit the output of FindBugs runs on your projects. The most urgent violations are marked as **Blocker**. Here is a number of blocker rules, which our FindBugs installation contains:

- **Correctness** - A known null value is checked to see if it is an instance of a type
- **Correctness** - close() invoked on a value that is always null
- **Correctness** - equals method always returns false
- **Correctness** - equals method always returns true
- **Correctness** - equals(...) used to compare incompatible arrays
- **Correctness** - Impossible cast
- **Correctness** - Impossible downcast
- **Correctness** - Impossible downcast of toArray() result
- **Correctness** - Null value is guaranteed to be dereferenced
- **Performance** - Maps and sets of URLs can be performance hogs
- **Performance** - The equals and hashCode methods of URL are blocking
- **Performance** - Hardcoded constant database password

What immediately strikes an eye is that these are indeed the things that you don't want in your code, so checking your application for these is really a no-brainer! Just imagine how bad it could be if one of these will bring your entire system down.

If we go down one level to the **Critical** severity, we immediately get a list of 202 conditions that are thought to be bad. The general areas that these rules are divided into are:

- **Bad practices** - e.g. returning a null from a `toString()` method
- **Correctness** - e.g. like having a field that is only initialized to null
- **Dodgy** - e.g. Weird increments in the loop
- **Multithreaded correctness** - e.g. `java.util.concurrent.await()` (or variants) which is not in a loop
- **Performance** - e.g. building a String using concatenation in a loop
- **Security** - e.g. executing SQL statements with dynamically generated String parameters

And a couple of violations are not tagged with any of them.

We strongly believe that these are good rules to follow, and suggest tweaking the output of FindBugs to make it less spammy for your project. What this means is that the quality of the code there can be majorly improved, so you should probably focus on the cases which seriously affect your code.

To run an analysis with a specified threshold for severity you can just pass a property to the Maven plugin:

```
mvn findbugs:findbugs -Dfindbugs.threshold=High -Dfindbugs.includeTests=false
```

This omits lower-level warnings all together and avoids running the checks on tests. Yeah, we're guilty here of advising a somewhat-ill thing: tests should also be production-grade, but if you only have time to fix a limited amount of violations, let them be in your main application. Fix the tests in the next iteration ;-)

Running this command on the Jetty project with FindBugs leads to a much faster result. If previously the default analysis took 8.5 minutes, this trimmed version ran in under 6 minutes, mainly because it didn't look at tests. Additionally, the amount of warnings is now greatly reduced and we've set it up so that only the most blatant violations are highlighted.

If you have a multi-module Maven project, [consider configuring an additional module](#) to encapsulate the FindBugs configuration. Then you can make all modules that require the analysis use that preconfigured module to run consistent analysis across the whole project.

Another suggestion: incorporate running this version of FindBugs analysis as a part of your Continuous Integration build. It shouldn't annoy developers with obvious false positives and still keep the health of your project on a better level.

Hints for customizing Checkstyle

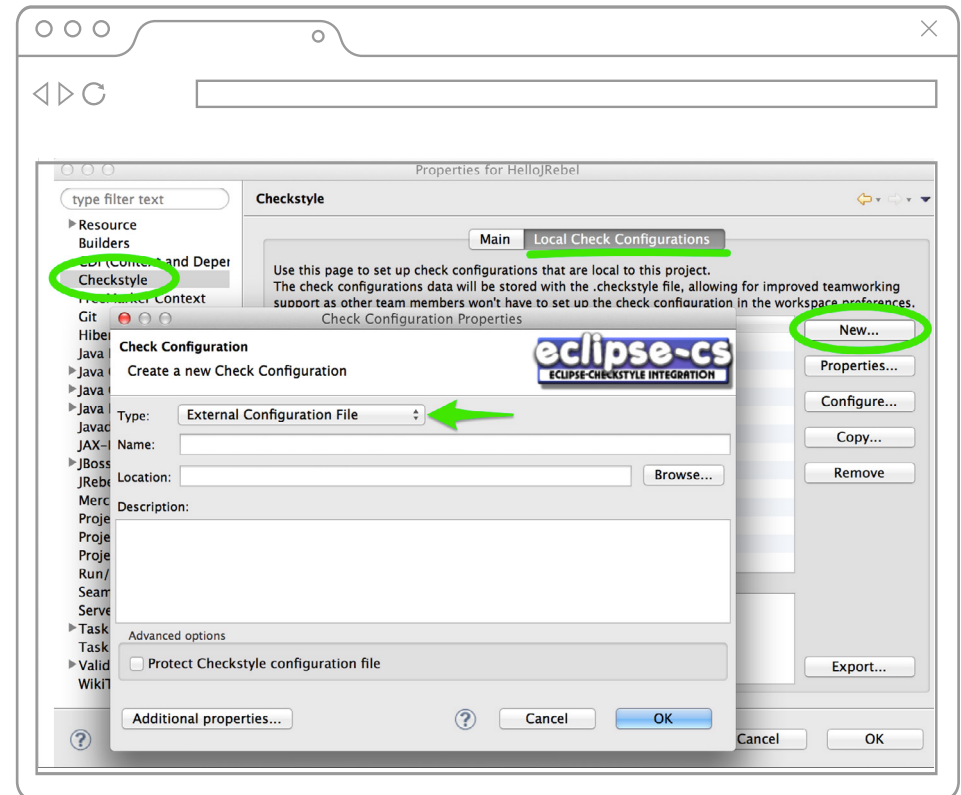
Checkstyle is the easiest tool to use from our perspective. “Code style” is a soft problem and is the easiest to agree upon among the development team. Just configure it according to your own weird requirements and enforce the rules you need upon the team :-)

In fact, an established team very likely is already using some code style guide or an automatic formatter: IDEs are good at this. The advantage of an IDE is that it is much closer to the developer making the changes, and in this way Checkstyle is not that useful. However, it is wise to include it into your CI build process in order to keep a hand on the pulse.

It’s hard to choose a correct configuration for Checkstyle, because everyone’s taste is different. We can safely advise you to follow the style of your existing codebase, and don’t make Checkstyle color every source file in your project red and give your fellow developers a reason to hate it.

When you have determined a Checkstyle configuration that fits your project you can generate Eclipse or IntelliJ IDEA formatter configurations and if you use NetBeans, import the Eclipse formatter into it.

For Eclipse users, just obtain an instance of Eclipse (duh!), and install the [Checkstyle plugin](#), from the Eclipse Marketplace. Then enable Checkstyle on a project, feeding it an external file where the prepared Checkstyle configuration is located.



Now right-click on the project, select **Checkstyle => Create Formatter Profile** and now you have an Eclipse formatter definition compatible with your Checkstyle configuration. You can share it among your team and convert to any other IDE formatter if you like.

Hints for customizing PMD

One of the first things we notice about PMD is that it's somewhat spammy. Source code has a much richer structure, than, for example, a compiled bytecode archive. This means that the rate of false positives when finding code inefficiencies is higher. The solution is to carefully examine rulesets that PMD employs when you run it and limit its output to the most severe cases.

The PMD team themselves recommend that in the best practices section:

>> Instead, start with some of the obvious rulesets - just run [unusedcode](#) and fix any unused locals and fields. Then, run [basic](#) and fix all the empty if statements and such-like. Then use the [design](#) and [controversial](#) rulesets and use the ones you like [via a custom ruleset](#).

Let's look at Jetty's specified ruleset for PMD analysis:

```
<configuration>
  <targetJdk>1.7</targetJdk>
  <rulesets>
    <ruleset>jetty/pmd_logging_ruleset.xml</ruleset>
  </rulesets>
</configuration>
```

If we remove this config from the parent XML, Maven will use the default config, so it use the prebundled *unusedcode*, *basic* and *imports* rulesets. Running just the *basic.xml* ruleset on Jetty takes less than a minute, and this time the resulting output is much smaller than the original 13000+ issues!

PMD found around 1700 issues where:

- **900** are violations of the [Useless parentheses](#) rule;
- **417** - Do not hard code the IP address
- **134** - Avoid empty catch blocks;
- **90** - complain about possibility of combining several if statements;
- **60** - Overriding method merely calls super;

There are 9 issues of the type **"Ensure you override both equals() and hashCode()"**, which is a good advice. But looking for it in this pile of less-interesting things is really annoying. However, there isn't much to do about it--you have to either strip the ruleset even more or fix those issues.

What to take away here

The most useful way to run these tools is an automatic integration with a central hub, namely SonarQube. It does make sense to go over the default settings for the individual tools and lower their verbosity, at least in the beginning. When your team gets used to the fact that their commits are automatically judged and finishes fixing existing code quality issues, you should consider enabling more rules to catch more bad code patterns.

In the next chapter, we go a bit deeper into this with a short list of suggestions (ok, common pitfalls) to keep in mind when getting started in the code analysis world.



<CODE>



</QUALITY>

CHAPTER IV:

FINAL ADVICE AND A GOODBYE COMIC :-)

Let's take a quick step back with these final reminders before you jump into action and start installing a bunch of tools to analyze all the cracks and smells in your codebase.

What we've seen so far...

In previous chapters, we look at many aspects of the gently abstract term “code quality”, starting off with what we already know about the positive effects; that analyzing code quality helps in terms of both application quality and the reliability of the application’s delivery.

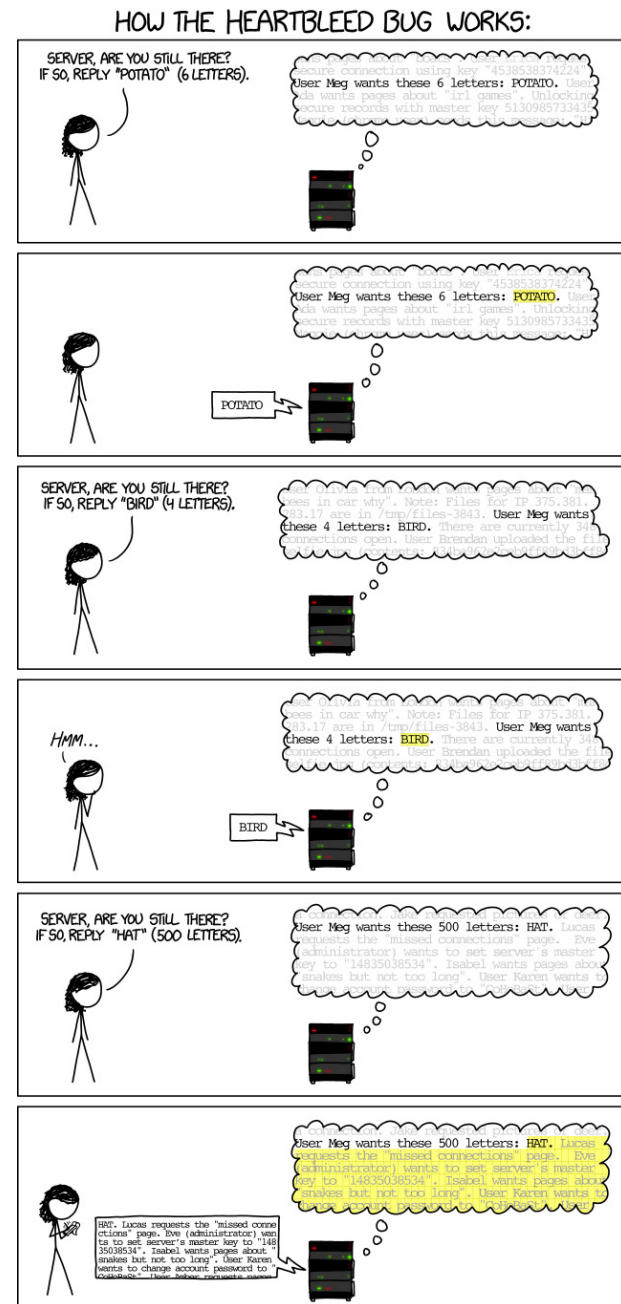
Next we looked at types of code analysis and other tool types that developers use, focusing on the static code analysis tools: SonarQube, FindBugs, Checkstyle, PMD and Coverity. We also looked at the questions you should (etc..).

We selected an open source project, Jetty, to analyze due to its popularity, licensing agreeability, plus the proximity to a complex real-world application. Then we moved on to initial set up and getting started with each tool, including preliminary results and why you shouldn't immediately get worried with them, because in the following chapter...

...We tried to make some sense of it all by showing you what a plugin-friendly management platform (SonarQube) and some useful customizations can do to help you avoid getting thousands of errors due to poorly-defined constraints in FindBugs, Checkstyle and PMD.

So, to top it all off, here is a short list of “7 Wise Recommendations”, the basic advice we'd give developers out there who want to bring a code quality tool in to enrich their codebase, including potential pitfalls and general advice.

Source: <http://xkcd.com/1354/>



7 Wise Recommendations

1. DON'T ASSUME THAT THE FIRST REPORT YOU GENERATE IS GOSPEL

Firstly, working with a code quality tool is actually quite similar to performance tuning in a way! When you performance tune an application server for example, you run a production load on it, monitor, make a change, run again, monitor... and so on. You iterate while making changes and acting on the results. This is a really important feedback loop, and it's similar with code quality tools too. Make sure that after you run your tool, you examine the results, change the rules, run again and so on, until you are happy that the results you are getting are targeting the style of defects/bugs/output you want, keeping focused on the most important area of the codebase.

2. DON'T RELY ON STATIC CODE ANALYSIS ALONE

Static code analysis is hugely advantageous to us as developers as it points to the areas of code where we might have potential problems. However, it doesn't point to every single problem, as some may be runtime problems that you really need a more dynamic tool for. Other problems in your code may still exist, since the tools available today simply aren't able to find them. You might have larger, more architectural problems, for example. Static analysis goes a long way to help you, but it's just another source of input and far from a complete solution. Hmmm, next report topic, you say?

3. AUTOMATE YOUR CODE ANALYSIS PROCESS (AND DO IT OFTEN)

"People are lazy, and developers are especially lazy. Having an automated way to check and report project's code health improves the project. Set of enforceable rules and low number of SonarQube issues also means it is quicker and easier to get new developers on the project or teaching old dogs new tricks."

- TOOMAS RÖMER, *Founder of ZeroTurnaround*

We expect many people say they use static code analysis tools, but in practice maybe they actually just run the tool each February 29th, manually, then get flustered with the configuration as well as the results, then fix a few bugs to feel good about life and email their manager. What really needs to happen is an automated code quality run that uses the same configuration every time, and runs frequently. We suggest that you make this part of your build or CI process and make sure that errors raised during code quality checks turn the build red. This makes sure that all the code that goes in has to conform to code quality standards in advance. It does take a while to get down to this, as you have to first eliminate all the bugs raised by the code quality tools in the first place, but once you're there, you really can sleep better at night!

4. MAKE THE ANALYTICS AND RESULTS OPEN TO YOUR TEAM

It's important that every team has trust, openness and clarity, and when doing something like code quality testing, there's no reason to keep the results a secret. You might want to share them over email with the team or have a known URL that is regularly updated with the latest version of the results. This way everyone can see the state of the codebase, what they can work on and potentially what they should focus on going forward to stop introducing the same problems again - that's right, it's not just about fixing the problems, it's about avoiding the same ones in the future.

5. DON'T MAKE DRASTIC CHANGES THAT CAN AFFECT OTHER DEPENDENCIES

We literally heard this: *"Wow, check out these code quality results! I could remove 20% of the errors it gives me just by adding braces around my code blocks. I'm gonna do it all now, touch virtually every class in the codebase and everyone will love me for it as I've fixed so many 'bugs'!"*

WRONG! What you're actually doing is creating a merge nightmare, particularly if you're not considerate about your timing. You're also adding a blotch on your VCS history, but that's not so important, and being a nuisance to others. Yes, it is important for your code to be consistent, but make sure you're not rushing your fixes through to get the numbers down, and remember that planning is important.

6. TAKE THE UNIQUE NATURE OF EACH PROJECT INTO ACCOUNT

No two projects are the same and each may have a different nature around it. For instance, should you always use the same set of basic assumptions as to what is good code for all Java projects? As with everything in life, there are always special considerations, edge cases and this is no different.

For example, if you're writing a Play application you should use a different set of assumptions about what you consider good code to be. Play encourages the use of public fields and direct access over private fields with getters and setters. They also use fast exceptions for passing results back, so rather than use a return statement, they'll throw an exception as a response. Weird? Sure, but you may want to take that into account before using a default set of code quality rules across it.

7. DON'T PRIORITIZE GREEN LIGHTS OVER WORKING & UNDERSTANDABLE CODE

There are many times in every developer's life when tricky decisions need to be made that compare several factors and ask that you compromise. This could include performance, readability, maintainability, code reuse, future planning etc. The list goes on.

Code quality is another metric that wants a say on how we should write our code, and of course it isn't always right. So we need to make decisions as to whether a code quality fix is actually worth it. Is fixing a SonarQube bug just going to make some particular piece of really important code less readable? If so, should you fix it? Or just add it to the ignore list and preserve your code base. There are more important people and things to keep happy than just code metric tools.

And on that note, we'll leave you with this:
Could better code analysis have prevented Heartbleed?



↙ Contact Us

Twitter: @RebelLabs

Web: <http://zeroturnaround.com/rebellabs>

Email: labs@zeroturnaround.com

Estonia

Ülikooli 2, 4th floor
Tartu, Estonia, 51003
Phone: +372 653 6099

USA

399 Boylston Street,
Suite 300, Boston,
MA, USA, 02116
Phone: 1(857)277-1199

Czech Republic

Osadní 35 - Building B
Prague, Czech Republic 170 00
Phone: +372 740 4533

Report Authors:

Oleg Shelajev, Sigmar Muuga, Simon Maple,
Oliver White

Report Designer: Ladislava Bohacova