SonarQube IN ACTION

G. Ann Campbell
Patroklos P. Papapetrou

FOREWORD BY Olivier Gaudin

MANNING

SonarQube in Action

SonarQube in Action

G. ANN CAMPBELL
PATROKLOS P. PAPAPETROU

MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964

Development editor: Susanna Kline
Copyeditor: Tiffany Taylor
Proofreader: Toma Mulligan
Typesetter: Dottie Marsico
Cover designer: Marija Tudor

ISBN 9781617290954
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – EBM – 18 17 16 15 14 13

To the software architects, programmers, testers, project managers, executives, and end users of every piece of software ever written.
We hope this book will make your lives easier.

brief contents

contents

foreword

The software industry is still a young industry in which *software quality* means for many people "pain," "cost," "constraint," "nice to have," "one-shot effort," or "external reviews." Fortunately, with the Agile movement, the industry has started to realize during the last decade that software quality also means "fun," "built-in," "rewarding," and "higher productivity." Ann Campbell and Patroklos Papapetrou belong to the latter group, and they strongly believe that software quality should be a daily concern shared by all stakeholders in the industry for long-term success.

Software quality is divided into external and internal quality. *External quality* looks at how well the software fulfills its functional requirements: in other words, whether you're building the right software. *Internal quality* looks at how well the software is designed/implemented to constantly welcome new changes: in other words, whether you're building the software right. Industry statistics show that on average, 80% of the cost of software is spent on maintenance; there is considerable variability depending on internal quality. This makes internal quality a key component for the future cost of software.

This is the reason why managing code quality of applications has become a major concern for any company that builds or is involved in building software. Traditional approaches to managing code quality propose to test code from time to time, mainly at the end of a development phase. In the best case, this approach leads to delays and re-work; in the worse case, it leads to the shipment of poor-quality, expensive-to-maintain software. There is therefore an urgent need for a new approach: one that clearly gives ownership of code quality back to the development team; one that emphasizes quality throughout the development phase and has a shorter feedback loop to ensure

rapid resolution of quality problems; in short, a model that builds in quality from the start, rather than considering it after the fact.

This is the mission we have set ourselves at SonarSource: to provide tooling for support of this new approach called Continuous Inspection. This is what we believe we have achieved with SonarQube, the open source platform to continuously manage technical debt. SonarQube has a large ecosystem, is widely adopted, and has a very large community. Ann and Patroklos are part of this community and among the most active members, contributing not only by their feedback but also by expanding the ecosystem. When they approached me with the idea of writing a book, I was thrilled, because this is clearly something that is missing in the SonarQube ecosystem. Having Ann and Patroklos writing it also meant it would have some great insight from the community and, more important, that it would contain the end-user perspective on the solution.

This book will be your companion in your journey with SonarQube. It will take you from why you should use SonarQube to installation, configuration, administration, and utilization of services, up to extending the platform. You can use it either by reading through from A to Z or as a support reference for information about a specific topic.

But that isn't all! Ann and Patroklos also discuss the process surrounding the tool, challenge existing and missing functionality, and provide numerous tips for using SonarQube, all based on their own experience. Whatever your level of familiarity with the product, you'll learn from this book. This is what, in my opinion, makes this book a unique source of information for a successful implementation.

Enjoy!

OLIVIER GAUDIN
CEO AND COFOUNDER
SONARSOURCE

preface

"Would you like to help me write a book about Sonar?" My reaction was immediate: "Yes!"

I knew Patroklos Papapetrou from the Sonar mailing list, and I was aware that he was pitching *Sonar in Action* (now *SonarQube in Action*, to match the technology's new name) to Manning. What I didn't know was that he wanted a coauthor. Because I was a native English speaker and active (and helpful) on the list, he thought of me. I had only been a member of the list for about six months, but I'd been aware of Sonar since late 2008 when my boss came across a mention of Sonar and asked me to evaluate it.

I was coding in Java at the time, but I had started my programming career with Perl and C. Lint was your friend, and bugs were found the hard way—by the users. So I found Sonar intriguing. It promised to scan each line of code and point out all kinds of things that were wrong or could go wrong. But to use it, you had to be building with Maven. Unfortunately, we were in an Ant-build shop. Sonar was off the table.

Fast-forward to early 2010. Sonar was approximately three years old, but already it was gaining broad acceptance among community and enterprise users and being downloaded more than 2,000 times a month. Patroklos had found the Sonar website while researching software quality tools, and it was a classic boy-meets-software story. (Cue the sappy music.) It didn't take long before he was in love and Sonar was one of his favorite tools.

Meanwhile, I had begun moving our Ant builds to Jenkins (it was still called Hudson then), and I stumbled across the Sonar plugin for Hudson. It works differently now, but at the time, it performed a shallow "Maven-ization" of a non-Maven project and ran an analysis. Hmmm. Maybe Sonar was back on the table.

I installed Sonar and the plugin on my localhost and ran an analysis. When I poked around in the results, I didn't understand everything I was seeing, but I knew I liked the way it presented issues in the context of the offending code. And because Sonar had a web-based front end, instead of having to send quality reports to people, I could send the *people* to the *reports*! When I showed my colleagues, they agreed that what we were seeing was good stuff, so we teamed up on our management.

We pretty quickly got first-level management's buy-in to pilot Sonar, and we started talking about it to anyone who would listen. Pretty soon other teams were approaching me to set them up "with that Sonar thing." We were seeing a viral adoption. After only a couple of months, management at the next level up said that everyone needed to be on Sonar by the end of the year.

By this time it was early 2011, and Patroklos was an active member of the Sonar community. He had spent 2010 telling everyone *he* knew about Sonar via his articles and blog posts. He also implemented his first Sonar plugin that year. By August of 2011, he was such a prominent figure in the small Sonar community that another publisher approached him to write a book about it. He was flattered, but didn't have the time to do it justice, so he turned it down.

But although Patroklos didn't write that book, he didn't forget the idea. When his schedule cleared at the end of the year, he approached Manning about writing *Sonar in Action*. A few short months after that, we were on our way.

Our goal in writing this book has been to condense the SonarQube lessons we learned in the last few years, combine them with whatever wisdom we can lay claim to from our combined 30 years of programming, and put a bow on all of it for you. The first time I ran a Sonar analysis, I didn't understand some of the things I was seeing, but we don't want that to happen to you. We'll tell you not only what SonarQube's metrics mean, but also why you should care and (unless it's really, really obvious) what steps to take in your code to get started fixing what's wrong. We'll help you plan a strategy for tackling your technical debt, and we'll show you how to make SonarQube a part of your routine rather than something extra you have to remember to do. We'll guide you in twiddling SonarQube's knobs so you can tune it to get the best experience in your environment. And finally, in case you feel the need, we'll show you how to get started writing your own SonarQube plugin.

Over and over, I've seen this in action: good programmers are passionate about quality code. Show us what the problems are, and we'll be almost compelled to fix them. In the past, the hard part has been pinpointing the problems. With SonarQube, the only hard part is finding the time to deal with them. Code quality used to be hard. Now it's easy. Welcome to SonarQube.

ANN CAMPBELL

acknowledgments

When we signed up to write this book, we didn't know how much effort it would require. It was for both of us our first authoring attempt, and although Ann had considerable experience with documentation and journalism, Patroklos's most recent writing had taken place a decade earlier. Without the assistance of many people, this book would definitely not be in your hands or on your screen right now. It's much more likely that we would still be working on it.

From day one of the project, help has poured in, and all of it has been not only appreciated but essential. If we've forgotten anyone, please accept our sincere apologies. In no particular order, we'd like to thank the following people for their work and support.

Manning Publications

Many thanks to publisher Marjan Bace for accepting the initial proposal of two new and inexperienced authors. He helped the book take its first—and most important—step.

Thanks to Christina Rudolph and Michael Stephens, who were the first two people we talked with about the book. They guided us as we improved our proposal, and they explained the publishing process in detail.

Hillary Clinton said it takes a village to raise a child. It turns out that it takes one to produce a book, too. Clearly, our development editor, Susanna Kline, deserves a shout-out for her continuous support and help. Her patience and encouragement were invaluable.

Thanks to Bert Gates and Cynthia Kane for teaching us how to write our first chapters following "Manning style."

Thanks to Candace Gillhoolley for her marketing efforts on the first MEAP release. She is a master of social media.

Knowing that we didn't have to worry about every semicolon because a copyeditor would come after us has been tremendously freeing; our thanks to Tiffany Taylor. And thanks to many other members of the Manning team who helped us behind the scenes: Maureen Spencer, Kevin Sullivan, Olivia Booth, Toma Mulligan, Mary Piergies, Dottie Marsico, and Janet Vail. And special thanks to our technical proofreader, Craig Smith, for his final review of the manuscript shortly before it went into production.

The SonarSource team

Of course, without SonarQube itself, the book wouldn't have been possible. That puts the folks at SonarSource squarely at the front of the line for thanks for their incredible free and open source contribution to software quality. More than simply providing the software, they've actively supported this book. SonarSource CEO Olivier Gaudin and Product Director Freddy Mallet reviewed each chapter, offering invaluable feedback and insight. The fact that we have their blessings and support … wow. We couldn't ask for more. Without their help, we wouldn't have been able to publish this book with the most updated material.

The reviewers

Thanks to our MEAP readers and to the reviewers who read the manuscript at various stages during its development and gave helpful comments and feedback to our editors and to us: Alex Garret, Bobby Abraham, Brandon Campbell, Chris Baxter, Christopher Taylor, Gregor Zurowski, Jason S. Shapiro, Javier Garcia Martin, Joshua White, Mark Elston, Michael Hüttermann, Mikkel Arentoft, Rashid Jilani, Reinhard Prehofer, Robert Wenner, and Steven Hicks.

Ann Campbell

I'd like to thank my husband, Charles Nix; my dog; and the rest of my family for their patience with me this past year as I concentrated on writing rather than on … really, anything else in my life.

I'd also like to thank my high school English teacher, Dr. Richard L. Handlesman ("Doc" to his students), for forcing me to write an essay every two weeks; and my mother, Polly Campbell, for teaching me to write them (whether I wanted to learn or not).

Patroklos Papapetrou

I'd like to thank my loving and beautiful wife, Anna, for her patience all these months, especially during the weekends. Without her encouragement, I wouldn't have managed to finish my part of the book. Thanks to my sons, Panagiotis (age 4) and Charis (age 2), who understood that sometimes Daddy couldn't play with them or go to the park. You can have me back now! Thanks to our families for their patience as well and for sometimes watching the kids to let me work on the book.

about this book

Welcome to *SonarQube in Action*. This book is aimed at turning all the tedious and sometimes hard-to-understand stuff about source code quality and software metrics into an exciting experience. It aims to become the Holy Bible of software quality: a reference for every development team that wishes to improve their source code. You'll see that metrics are meaningful and affect several aspects of your software's health. In this journey, SonarQube will be our pilot. SonarQube is an open source platform for continuously measuring, managing, tracking, and improving source code's quality.

How this book is organized

We begin each chapter of the book by describing a real problem/situation, and then we talk about the features of SonarQube and the relevant metrics that help you address and eventually solve that problem. We elaborate by providing some theoretical background, we discuss best practices (if any), and we end each chapter by talking about relevant—to the chapter's topic—SonarQube plugins and how you can take advantage of them.

Now it's time to list in detail the book's content.

Part 1 gives you an overview of SonarQube, explains the seven axes of quality (like the seven deadly sins of software development), and sets the stage for the following parts. We introduce SonarQube's key features and benefits and discuss the core metrics that SonarQube calculates.

- Chapter 1 sets the scene, introducing the core concepts of SonarQube. We begin by showing you what you should expect to see when you analyze a project

using SonarQube for the first time. We briefly discuss the different metrics presented in SonarQube's dashboard. At the end, we present the Technical Debt plugin.

- Chapter 2 introduces the topic of code issues. You'll learn, among other things, where they come from and how they're related to bugs or potential bugs.
- Chapter 3 is all about testing (unit and integration). It describes the importance of code-coverage metrics and how they're calculated, and it gives you some tips for improving the test quality and coverage of your code.
- Chapter 4 focuses on duplicated code by illustrating the problems that may arise and the resulting impact on the quality and maintainability of your source code.
- Chapter 5 deals with a topic which is rarely considered by development teams as a quality factor: documentation. You'll find out when and why it's a good practice to document your code, and we'll present a proposed documentation strategy that fits any development process.
- Chapters 6 and 7 talk about design and complexity. Although some may argue that they're more or less the same thing, we've chosen to split them in order to provide more examples and illustrate their value in code quality.

Part 2 discusses how you can get the best out of SonarQube, where it fits in any development lifecycle, and how to make it part of your everyday work life. It also introduces the concept of Continuous Inspection, which is the ultimate target when talking about software quality.

- Chapter 8 discusses several approaches for improving the quality of your source code. You can pick one or all of them based on your experience. Then we'll take you on a tour of all the possible data perspectives that SonarQube offers, and we finish by explaining the concepts of history and trending.
- Chapter 9 delves into the details of Continuous Inspection. We'll talk about integrating SonarQube with Jenkins, and you'll learn about the star feature of differential views that lets you track quality evolution over time.
- Chapter 10 deals with the popular practice of code reviews and explains how you can benefit from SonarQube. You'll find out how issues are associated with reviews, how to track them, and what SonarQube features let you plan your work with action plans.
- Chapter 11 talks about integrating SonarQube with Eclipse. Enjoy most of the SonarQube advantages without leaving your IDE by following the step-by-step guide provided in this chapter.

Part 3 covers several administrative topics and gives you ideas about customizing and tuning SonarQube to make it suitable for any kind of project. In an enterprise environment with a SonarQube installation that hosts several projects, it's a good idea to adjust many of SonarQube's predifined settings to fit your needs. This part of the

book also teaches you step by step how to extend SonarQube by writing a custom plugin.

- Chapter 12 explores security concepts, including users, roles, and groups. You'll learn how to delegate authentication and authorization to external systems (LDAP, OpenID, and so on).
- Chapter 13 deals with managing coding rules and organizing them in quality profiles. You'll also discover how you can create your own rules or edit existing ones and trigger alerts when metrics fall below a threshold.
- Chapters 14 and 15 discuss global and project administration, including filters, dashboards, and user notifications. The latter also provides a simple path for adopting Continuous Inspection by discussing useful SonarQube features that will assist you in this direction.
- Chapter 16 is dedicated to teaching you how to extend SonarQube. Although it's not possible to cover everything in a few pages, we provide a complete example of implementing a real SonarQube plugin. We also give you some insights into adding support for new programming languages.

The book also has two appendixes that will help you with the basics, especially if you're a SonarQube newbie:

- Appendix A focuses on installing SonarQube in Linux and Windows.
- Appendix B provides all the necessary details to run your first analysis with SonarQube.

One last thing—don't expect to find correct code in this book. Chapter 16 is the only exception to that rule, because it deals with writing plugins. Most of the examples intentionally illustrate bad habits in coding, and their purpose is to point out what you should avoid. Nevertheless, in some cases we've included a refactored version to show you that by using SonarQube, you can begin to understand these nasty metrics and dramatically improve the quality of your code.

How to use/read this book

Each person has their own reading style, and we can't force you to change it for this book. But we can still give you a couple of ideas on how to get the most out of this book.

Every chapter is organized in such a way that you can read it separately from the rest. We do suggest that you read chapter 1, especially if you're not an experienced SonarQube user, because it's an overview of SonarQube and introduces some basic ideas you may need when reading the rest of the book.

If you decide to read the book sequentially, you'll find that each chapter is connected to the previous one, and the chapters flow smoothly, without gaps. But again, you can skip any chapter and come back later if you want to.

We did our best to ensure that this book will become a reference for you whenever you need to learn or remember anything about SonarQube or its computed metrics.

Who should read this book

Believe it or not, source code quality is a topic that targets almost everyone who participates in a software project. Although we provide several code examples, you don't need to be a code expert to read this book. You also don't need to be familiar with Java, because the code listings and snippets are so simple that anyone with basic programming skills can understand them. Besides, don't forget that most of the examples in the book show you poor or bad code, to illustrate techniques and habits you should avoid. We do expect that you have some basic knowledge about software quality metrics.

The book is aimed at the following professionals:

- *Software engineers (developers, designers, architects)*—This is the book's primary target audience. Software engineers live a day-by-day battle to achieve software quality, hunting and fixing bugs, adding new features, and designing and redesigning the logical architecture of the system. Not to mention that all these things have to be done within strict deadlines and constantly changing business requirements. This book will help you spot the parts of the software that need your attention so you can take immediate action.
- *Quality assurance staff and testers*—QA stuff nowadays plays a valuable role in software engineering. In most cases, these people are part of the development team, and it's up to their judgment whether a product should be released. If you fall in this category, this book will teach you how to track the quality of the software under devlopment in an easy and comprehensive way, how to define criteria and thresholds for critical metrics, and, eventually, how SonarQube can assist you in decision making.
- *Project/Product managers and team leaders*—The era of project/product managers and team leaders sitting in an office, isolated from the rest of the development team, has passed. Managers exist to do more than read weekly reports and track down timelines and deliverable. They must have a clear view of the software and especially its quality in order to assist team members and get them on the right track. This book explains all the quality axes without unnecessary technical details. It provides you with a guide to how you can automatically track quality measures in source code over time and improve the development lifecycle by introducing new practices such as code reviews and Continuous Inspection.

Code conventions and downloads

All the source code in the book, whether in code listings or snippets, is in `a fixed-width font like this`, which sets it off from the surrounding text. In most listings, the code is annotated to point out the key concepts, and numbered bullets are sometimes used in the text to provide additional information about the code. We have tried to format the code so that it fits within the available page space in the book by adding line breaks and using indentation carefully.

Source code for all the examples and the plugin from chapter 16 are available at www.manning.com/SonarQubeinAction. If you want to get the most updated source code for the plugin—remember, it's a real one, so the latest version is likely to be different from the code shipped with the book—it's available at the following GitHub link: https://github.com/ppapapetrou76/sonar-redmine-plugin.

What this book doesn't do

This book should not be considered a user or administration guide for SonarQube. If you just want to learn how to use SonarQube, the online documentation at http://docs.codehaus.org/x/EoDEBg should be sufficient.

This book also doesn't explain the underlying tools with which SonarQube integrates, such as PMD, FindBugs, Checkstyle, and so on. You're encouraged to visit the corresponding websites to learn more about their purpose.

In some chapters, we include tips and best practices for refactoring as well as some introductory material. But this book doesn't teach you how to refactor your code.

Author Online

The purchase of *SonarQube in Action* includes free access to a private web forum run by Manning Publications, where you can make comments about the book, ask technical questions, and receive help from the authors and from other users. To access the forum and subscribe to it, point your web browser to www.manning.com/SonarQubeinAction. This page provides information on how to get on the forum once you are registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the authors can take place. It is not a commitment to any specific amount of participation on the part of the authors whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions lest their interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

About the authors

G. Ann Campbell has 15 years of experience in Perl, C, C++, Java, and web technologies on variously sized and organized teams, and she has spent far too much time achieving code quality the hard way without SonarQube.

Patroklos P. Papapetrou is a Java architect, an experienced software developer, and an Agile team leader. He's an active SonarQube community member and contributor.

about the cover illustration

The figure on the cover of *SonarQube in Action* is captioned "Habit of a Bonze of China in 1700." A bonze is a Buddhist monk. The illustration is taken from Thomas Jefferys' *A Collection of the Dresses of Different Nations, Antient and Modern* (4 volumes), London, published between 1757 and 1772. The title page states that these are hand-colored copperplate engravings, heightened with gum arabic. Thomas Jefferys (1719–1771) was called "Geographer to King George III." He was an English cartographer who was the leading map supplier of his day. He engraved and printed maps for government and other official bodies and produced a wide range of commercial maps and atlases, especially of North America. His work as a mapmaker sparked an interest in local dress customs of the lands he surveyed and mapped, which are brilliantly displayed in this four-volume collection.

Fascination with faraway lands and travel for pleasure were relatively new phenomena in the late 18th century, and collections such as this one were popular, introducing both the tourist as well as the armchair traveler to the inhabitants of other countries. The diversity of the drawings in Jeffreys' volumes speaks vividly of the uniqueness and individuality of the world's nations some 200 years ago. Dress codes have changed since then and the diversity by region and country, so rich at the time, has faded away. It is now often hard to tell the inhabitant of one continent from another. Perhaps, trying to view it optimistically, we have traded a cultural and visual diversity for a more varied personal life. Or a more varied and interesting intellectual and technical life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Jeffreys' pictures.

Part 1

What the numbers are telling you

In part 1 of *SonarQube in Action*, we'll start by giving you an overview of SonarQube in chapter 1: what it offers, what you can do with it, and why you're interested in giving it a try. Then, in chapters 2 through 7, we'll walk you through each of SonarQube's Axes of Quality for an in-depth look at what each axis means. We'll show you the Axes' impact on the quality and maintainability of a code base, and we'll look at how to begin fixing what SonarQube has pointed out in your applications.

1 An introduction to SonarQube

This chapter covers

- Why SonarQube
- Running your first analysis
- The Seven Axes of Quality
- Languages SonarQube covers
- Interface conventions

For as long as software developers have been writing code, we've been asking ourselves and our teammates, "Did we do it right?" Until fairly recently, there weren't a lot of good answers.

Unless you worked for NASA, the answer was "Well, it compiles." Or, "Um, it seems to work." And then there's the perennial favorite: "The users aren't complaining."

Sometimes that was enough. Until the users did start complaining. Or until we had to add new features. Which is when we realized just how "not right" we had done it.

More recently, people have tried to answer these questions with automated test suites. But how do you know you've written enough tests? What about the things tests can't cover?

As much as developers have struggled to understand when they've "done it right," their bosses have struggled even more. It's easy enough to evaluate salesmen (product sold), and lawyers (cases won), and factory workers (whatzits produced with acceptable quality). But how do you evaluate a coder?

In the past, people have been so stuck for an answer that they've resorted to the factory worker model. Only instead of whatzits, lines of code were counted. Not even "lines of code with acceptable quality," just "lines of code." Because measuring quality was hard.

Now it's not. Welcome to SonarQube.

1.1 Why SonarQube

Imagine that your CEO's aunt is also a customer. It's not a big account; in fact, it's tiny. But she makes his favorite pie, so her opinion matters more than it should. Unfortunately, the last release had a couple of bugs that mattered to her, so *he*'s been on the warpath ever since. He's started ranting about quality and demanding numbers. He says that if you don't come up with a way to measure quality and show improvement, *he will*. The glint in his eye says you won't like it.

Now what? Now it's time for SonarQube, which will help you manage your code quality, instead of letting your code quality (and Aunt Betty) manage you.

SonarQube is a free and open source "code quality platform." It gives you a moment-in-time snapshot of your code quality today, as well as trending of *lagging* (what's already gone wrong) and *leading* (what's likely to go wrong in the future) quality indicators. For test coverage (a leading indicator), a score of 50% may not look great, but what was it last month? If you're up from 35%, it's high-fives all around. Down from 70%? Time to shape up.

SonarQube doesn't just show you what's wrong. It also offers quality-management tools to actively help you put it right: IDE integration, integration for Jenkins, a popular Continuous Integration server, and code-review tools.

SonarQube's commercial competitors in the code-quality space offer some of those things too (depending on which one you're looking at); but they seem to focus their definition of *quality* mainly on bugs and complexity, whereas SonarQube's offerings span what its creators call the *Seven Axes of Quality*. We'll cover them in more detail soon, but in brief, SonarQube addresses not just bugs but also coding rules, test coverage, duplications, API documentation, complexity, and architecture. Some of the other players in this space also hit unit-test coverage and API documentation, but no one else seems to address all Seven Axes.

Because it's free, it would be easy enough to say, "Why not SonarQube? Might as well try it." Although that's a perfectly valid reason to give it a shot, once you put it through its paces you'll see that even if it weren't free, SonarQube would be well

worth the investment. That's because software quality is something every system stakeholder cares about, not just end users.

NOTE SonarQube hasn't always been called that; it used to be named just plain Sonar.

From a tester's standpoint, SonarQube is worth attention because it will help you pinpoint the spots where automated testing is thin or nonexistent. It may also help target manual penetration and security testing.

From a developer's standpoint, SonarQube is worth the effort because it helps you grow as a coder. From language-specific subtleties to thread safety and resource management, SonarQube can show you what you're getting wrong—or doing sub-optimally—and point you in the right direction for fixing it. That guidance isn't just for the folks fresh out of school. Experienced programmers can learn from SonarQube, too, even if it's only that their super-elegant code will be unreadable to the new guy. Plus, let's face it; everyone has off days, and SonarQube helps coders find their goofs and fix them quickly.

From a software architect's standpoint, SonarQube is worth the time because it helps you keep an eye on whether your cleanly delineated initial design is being degraded over time with creeping dependency cycles. It can show you whether the internal coding rules are being followed, and it can help you spot rising complexity that needs to be refactored.

From a project management standpoint, SonarQube is worth the focus because testing alone isn't enough. It can only show whether software does what it's supposed to do: its level of *external* quality. On the other hand, SonarQube analyzes and fosters *internal* quality: whether an application will run optimally and be readily maintainable and extensible down the road.

From a business standpoint, SonarQube offers a strong ROI because its acquisition and setup costs are low, and its intuitive interfaces mean that very little training is required. Add to that the fact that its adoption within an organization is typically viral, and you've got a minimal investment that produces what quickly become significant results.

Finally, *from a management standpoint*, SonarQube is worth the investment because it gives you metrics. Like the stereotypical charts that salesmen are measured by, with SonarQube in the fold, you've now got trending available on abstract measures of code quality. It even offers charts, like the one in figure 1.1.

In nearly every industry, serious leaders track metrics. Whether it's manufacturing defects and waste, sales and revenue, or baseball's hits and RBIs, there are metrics that tell you how you're doing: if you're doing well overall, and whether you're getting better or worse.

Now we've got those metrics for software, packaged and presented through a standardized, centralized, server-based quality platform (nothing to install client-side!) that uses code-quality tools already respected in the industry, such as FindBugs, PMD,

Figure 1.1 Trending is a core feature of SonarQube, with changes represented in a variety of formats, including this spark line-style graph that can be added to any dashboard.

and JaCoCo for Java; and Gallio, Gendarme, and FxCop for C#. Those results are presented in a fairly intuitive web front end that also offers RSS feeds of the Alerts raised when the quality thresholds you set are crossed.

1.1.1 Proven technologies

If SonarQube uses existing tools, you might ask why you need it at all. Why not just run those tools alone? There are a couple of reasons. The first is that the tools generate laundry lists of potential problems. They don't generate metrics, and, for the most part, they don't offer tracking or trending from analysis to analysis.

Numbers aside, of course those tools *can* be run without SonarQube. But are they? Any developer can download FindBugs or Gallio and point it at his code, but does he? How often? With what settings? Are they the same ones that the coder in the next cube is using? And what's the visibility of those quality reports to the team at large? SonarQube answers all these questions by applying standardized rule sets, not just from analysis to analysis on a given project, but potentially across your entire stable of projects. It also offers its own rules and measurement algorithms, such as an enhanced duplication-detection mechanism that shows you cut-and-paste not just in a given project, but across projects as well.

With all that going on behind the scenes in SonarQube, you might think you need a PhD to use it. In fact, analysis is easy to set up, and the SonarQube interface is surprisingly intuitive, as you'll begin to see shortly. Additionally, SonarQube's review functions and integration for the Eclipse IDE make it easy to transparently manage the fix for whatever SonarQube tells you might be wrong.

1.1.2 Multilingual: SonarQube speaks your language

If you've heard of Sonar or SonarQube before, it may have been in the context of Java. SonarQube is written in Java, and it started as a way to measure the quality of Java projects, but it's no longer limited to analyzing just Java. There is an ever-growing list of languages you can analyze through SonarQube by adding plugins, many of which are provided by the SonarQube community and offered for free.

THE SONARQUBE COMMUNITY

The SonarQube community has two mailing lists: one for users and another for developers. You can search the archives or get instructions on joining either one here: http://sonarsource.org/support/support/.

Similarly, the other "language of SonarQube" is also easily changed with plugins. The interface is in English by default, but the community supports a number of localization plugins. So if you and your teammates are more comfortable in French, Spanish, Greek, Chinese, or Japanese than English, you can accommodate them to make it as easy as possible to use SonarQube.

Whatever language you speak, whatever language you code in, once you decide to begin using SonarQube you can manage your code quality from a proactive, rather than a reactive, standpoint. Start using it regularly, and you can begin to manage your quality, rather than letting your quality manage you.

1.2 Running your first analysis

At this point, we'll assume that you're bowled over by the possibilities and chomping at the bit to get started. It's not hard. Next we'll look at installation, then we'll give you a high-level walk-through of running an analysis, and we'll finish with a look at the SonarQube interface.

1.2.1 Installation considerations

First you'll need to download SonarQube, set up a database for it, and turn it on. All that's covered in appendix A, but before you jump in, there are a few things you'll want to take into account as you plan your setup.

See it on the web

If your interest is piqued but you're not ready to install SonarQube for a trial run yet, take a peek at Nemo, SonarQube's public instance: http://nemo.sonarsource.org. Here you'll see analyses of many open source projects and get a feel for what SonarQube can show you on large and small code bases alike. Just be aware that this is a showcase for SonarSource, the company behind SonarQube, so what you'll see is a tricked-out version of what you'll get from SonarQube out of the box.

In addition to having customized the interface, the SonarSource guys are also using Nemo to show off their commercial plugins for SonarQube. The ones you're most likely to notice on this site are the Developer Cockpit plugin, which is the only way to get metrics not on projects but on the developers behind them, and the Views plugin, which lets you aggregate quality metrics across multiple projects. Although SonarQube itself is free, you won't get everything you see on Nemo without spending some money.

Ideally, you're going to put SonarQube and its database on two separate hosts (to split the CPU load) that are side by side on the network. Network I/O is one of the biggest determinants in how long an analysis takes, so make sure you've got as fat a network pipe between the hosts as possible.

Once you've set up your database, configured SonarQube, and started it, open a browser and head to port 9000 of your SonarQube host. If you're running it locally, then your target URL is http://localhost:9000. When your browser is able to connect, SonarQube is up and running.

At this point, you're ready to begin collecting those metrics we've talked so much about; you're ready to run your first analysis. If you have a choice of where to run it from, then the closer you can get to your SonarQube server and database, network-wise, the better. Running it on the SonarQube host is best of all.

If you're in a Maven-centric Java shop, then you've got a few things to add to your pom.xml file, detailed in appendix B, and you're good to go. You'll also want to refer to appendix B if you need to integrate analysis into an Ant script. But the third option, SonarQube Runner, which we cover briefly in the next section and in detail in appendix B, is tidy enough that you may be tempted to `<exec>` it from Ant.

1.2.2 Analyzing with SonarQube Runner

Before you can run your first analysis, you need one more download: the SonarQube Runner, which actually runs the analysis. The analysis is accomplished by setting some properties and firing off the sonar-runner executable.

Project-specific values are set at runtime, either on the command line or in a properties file. The simplest possible project properties file would look something like the following listing.

Listing 1.1 sonar-project.properties

```
# required metadata
sonar.projectKey=my:project
sonar.projectName=MyProject
sonar.projectVersion=1.0

# required path to source directories
sources=srcDir1,srcDir2
```

A comment, which will be ignored

Project's unique ID, used internally

Project name to display to users

Listing 1.1 assumes that the project under analysis is written in Java and that you've set the database connection values in the global properties file as described in appendix B. It omits tests, the compiled byte code, and dependencies, all of which you'd want included in a full analysis. But if you've put sonar-runner in your path, it's runnable as follows:

```
cd [path to project root]
sonar-runner
```

Execute these steps, and your analysis begins immediately, with each step logged to the console and prefixed with a timestamp. The time to run an analysis varies by project size and, as mentioned earlier, network speed.

While you wait for your analysis to complete, you may want to flip to appendix B, which gives you a fuller idea of the analysis properties you can set. We've made the list as complete as we can, but SonarQube is under constant active development, so it's possible to have new properties with each release. Each language-specific plugin also typically adds its own properties to the mix.

1.2.3 Analyzing multilanguage projects

If you've browsed ahead to the analysis properties in appendix B and you have a project that uses multiple languages, you might be scratching your head at this point. For instance, a typical web project might use Java or C# for the back end and JavaScript on the client side, but appendix B shows that the `sonar.language` property (which specifies which language to use in a project analysis) only accepts one value. "Wait," you're saying. "I've got Java, JavaScript, and XML in my project, and I can only analyze one of them?"

Yes and no. You can analyze all of them. You can't do it all in one analysis.

Each SonarQube project is about a single language. So what you'll need to do is set up a separate analysis—a separate SonarQube project—for each language. This means that you'll have a SonarQube Runner properties file for each language, and you'll run an analysis for each properties file. (Maven folks, you'll still run your primary analysis through Maven, but you'll want to use SonarQube Runner for the rest.)

But whether you're using SonarQube Runner, Maven, or Ant to run your analysis, you need to make *sure* you give SonarQube a way to differentiate between, say, your Java evaluation of a project and the JavaScript analysis of the project. Don't assume that SonarQube will pick up on the difference in the specified language. It *only differentiates projects by* `projectKey`, *not* by `projectKey` and language. Reuse the same `projectKey` for each language, and each subsequent analysis against a new language will overwrite the previous one (which will do bizarre things to your quality trends). You may want to vary `sonar.projectName` as well, but that's only for your own convenience. It's not as crucial, and it can easily be changed later if you find yourself getting confused. In chapter 14, we'll show you how to set up filters (project lists) for each language, so if you wanted to keep the names the same, you could still easily tell each aspect of, say, Project Blue apart by whether you're looking at the list of Java analyses, the JavaScript results, or the Flex list.

1.2.4 Seeing the output: SonarQube's front page

Once your analysis is complete, you've got your first set of metrics. Point your browser back to port 9000 of your SonarQube host to begin seeing them. When you arrive, you'll find yourself looking at something much like figure 1.2, which shows a welcome message on the left and, on the right, two different views of the results of an initial local analysis of the open source Apache Tomcat project.

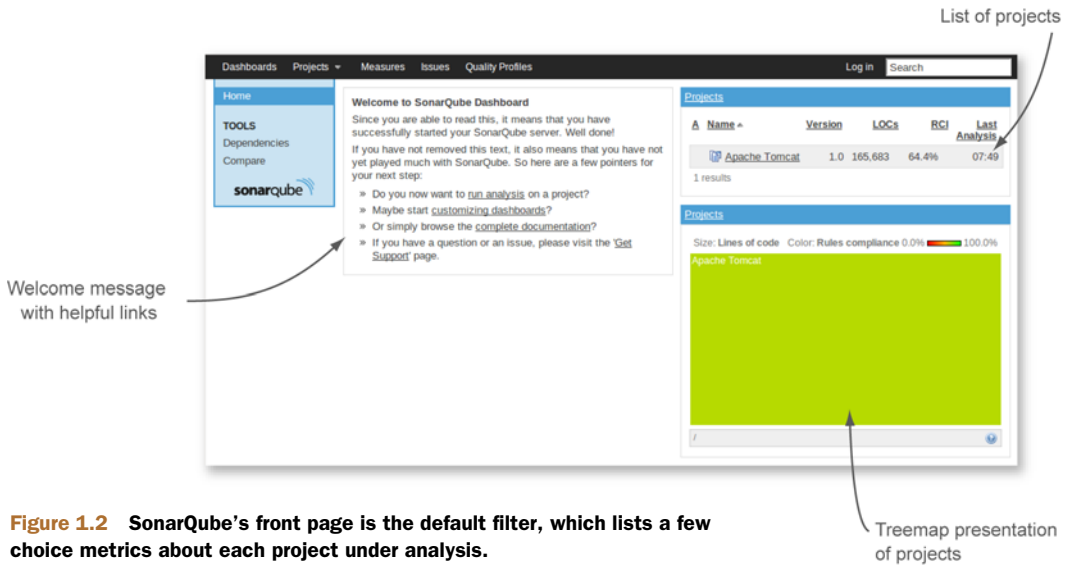


Figure 1.2 SonarQube's front page is the default filter, which lists a few choice metrics about each project under analysis.

Figure 1.2 shows SonarQube's default front page, which is your 30,000-foot view of code quality. The *widget* (box) at upper right lists all projects under analysis, with some key data points for each. The widget at lower right is a treemap of those same projects. With one project in SonarQube, it's a solid, colored box; but as you'll see in later chapters, that changes as you add more projects.

1.2.5 Drilling in: the dashboard

In either the project list or the treemap, click-through on the name of your project to reach the project dashboard. From the 30,000-foot view at the front page, you've just dropped to 10,000 feet. The view from this altitude looks something like figure 1.3.

There's a lot to take in on the default project dashboard (other dashboards are available), and it's difficult to generalize about it as a whole except to say that you'd like to see the bar graph weighted to the left. Each box here is called a *widget*. Generally, widgets are focused representations of a single facet of code quality. The widgets on the default dashboard show three types of metrics. The first kind is like a golf score; lower is better. The second is like bowling; high score wins. And the third is like age, which could go either way depending on your perspective, but which is just a value-neutral report of the current state.

Before we move on to the metrics, let's linger a moment on golf and bowling. It's no accident that we'll be comparing the metrics ahead to those two games. On the face of it, they each have you competing against other players, but unless you're on a professional tour and playing for prize money, you're *really* competing against yourself. You want to play better today than you did yesterday. It's the same thing with SonarQube; it's not about benchmarking, it's about having better code today than you did before.

Now let's look at some metrics.

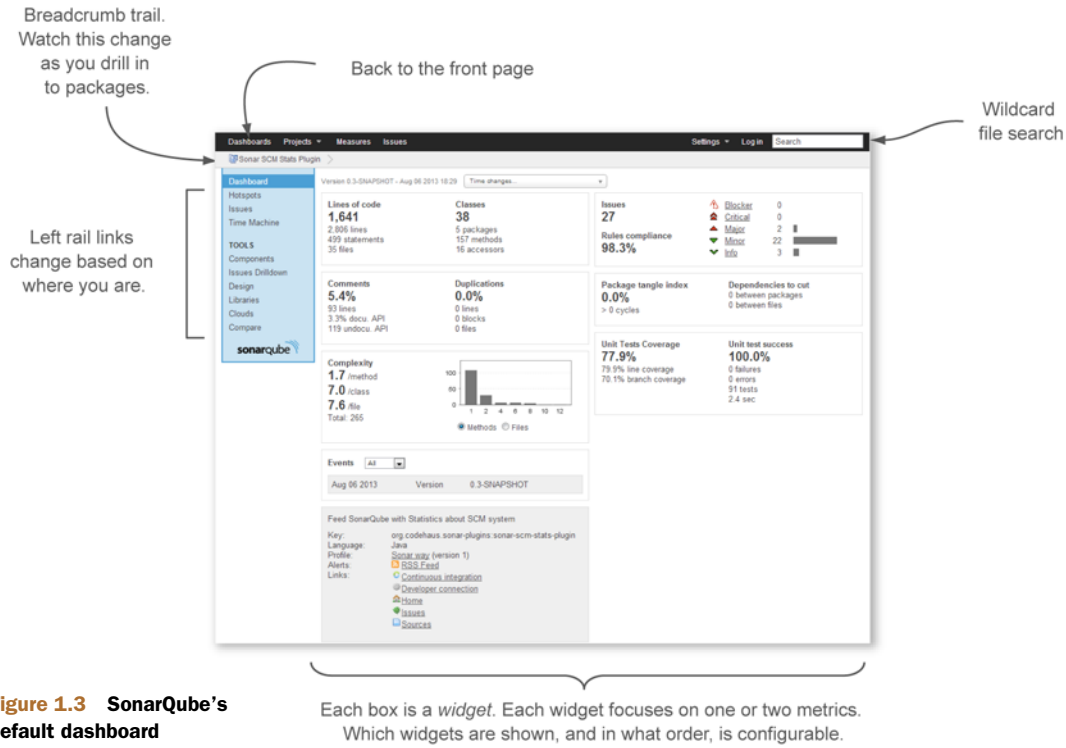


Figure 1.3 SonarQube's default dashboard

Size

The size metrics widget at upper left falls in the neutral category. It tells you how many lines of code, methods, classes, and packages it found during the analysis, as shown in figure 1.4.

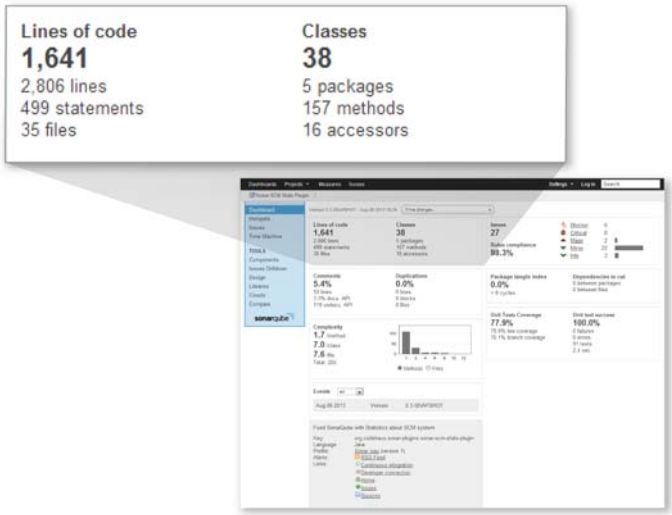


Figure 1.4 The size metrics widget shows how many lines of code, methods, classes, and packages were found during analysis.

We should make a distinction at this point between lines of code (often referred to as LOC) and physical lines, which SonarQube also reports. The number of physical lines in your project is a raw count of the number of times someone presses the Enter key, whether or not there's any content on the line. Lines of code, on the other hand, is meant to be a count of the number of "working" lines in your project. The LOC definition is language-specific, but SonarQube calculates it for Java by subtracting comments and blank lines from physical lines.

Why do you care? Because we're about to start showing you some percentages, and LOC is the bottom number in most of them.

EVENTS

The two widgets in the bottom of the left column are also neutral. Second from the bottom, the events widget gives you a quick list of events recorded on the project, as shown in figure 1.5.

There are several types of events, one of which is a change to the project version string that you pass in to the analysis. That's the lone event shown in figure 1.5. With the first project analysis, SonarQube recorded a version string "change," from nothing to something.

Events are significant, at least in part because they flag an analysis snapshot for long-term retention. Every time an analysis is performed, a snapshot of the project state is taken. That could quickly add up to a lot of snapshots, bloating the database, but SonarQube's rigorous housekeeping routines keep that from happening. Those routines are preconfigured to sane but tunable defaults. For more on that, see chapter 13.

DESCRIPTION

The description widget at lower left is a brief curriculum vitae of your project, showing the language and ID you set during analysis. It also shows the name of the rule set, or *profile*, applied in the last analysis, as shown in figure 1.6. Because multiple rule sets exist, knowing which one was used can make a difference.

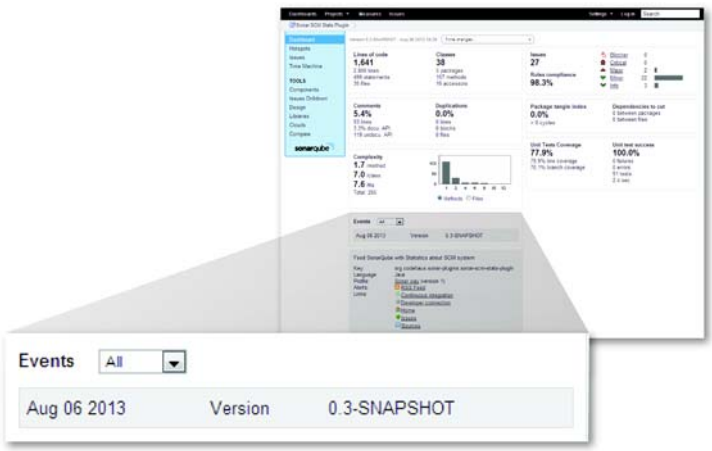


Figure 1.5 The events widget gives a quick list of events recorded for the project.

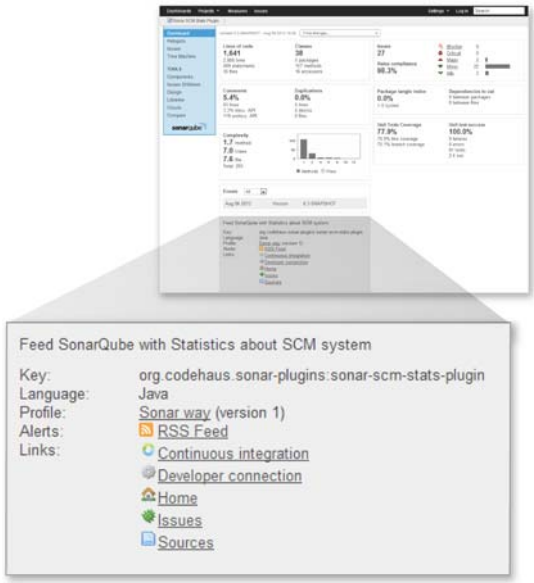


Figure 1.6 The description widget shows basic data about your project and its last analysis.

Finally, the description widget ends with a link to an RSS feed of Alert events on the project. We mentioned earlier that there are several types of events. One type of event relates to Alert thresholds you can set on a rule profile. When those thresholds are crossed in either direction, an Alert is raised (chapter 13 covers setting Alerts). For instance, you may choose to set an Alert when test coverage falls below 80%, or when the number of Blocker-level issues exceeds 0. These rule-based events are what the RSS feed gives you.

1.3 Seven Axes of Quality

The remainder of the widgets on the default dashboard relate to what the creators of SonarQube call the Seven Axes of Quality (and sometimes the Seven Deadly Developer Sins). First, let's be clear: axes here is the plural of *axis*. This has nothing to do with the seven dwarves or the pickaxes they carry into the mine every day. Instead, think geometry, where an axis is a line against which you measure distance or height (as in achievement).

Here's where we roll up our sleeves and dig in to the quality metrics of a project that we promised you at the beginning of the chapter. The axes that SonarQube measures a project against are as follows:

- Potential bugs
- Coding rules
- Tests
- Duplications
- Comments
- Architecture and design
- Complexity

In this section, we'll give you a high-level understanding of what each axis is about. In the rest of part 1, we'll go into detail on the metrics for each axis and give you a little practical advice on how to start patching any problems SonarQube shows you.

1.3.1 Potential bugs and coding rules

The issues widget, shown in figure 1.7, is a two-for-one. The creators of SonarQube list potential bugs and coding rules as separate axes, but for reporting they group them together under issues. Generally, you can consider issue counts as lagging quality indicators; they show what's already gone wrong.

Taken together, potential bugs and coding rule infractions span a continuum, from setting up a logic path through the code that's guaranteed to lead to a null pointer dereference, to not putting the open curly brace on the line the team has agreed to. Teetering between the two are things like flouting industry-standard naming conventions and writing one-line conditionals without using curly braces.

Given those examples, it's clear that some issues are worse than others. That's why SonarQube ranks them at different severities: Blocker, Critical, Major, Minor, and Info. The rules-compliance percentage you see at lower left in the issues widget gives perspective. It's based on the number and severity of issues versus the lines of code in the project. Whereas the issues counts are golf-style metrics, the rules-compliance index is like bowling: higher is better.

Looking ahead, chapter 2 covers the importance of issues, even the ones that don't seem all that critical at first blush. Later, in chapter 10, we'll talk about issue management, and in chapter 13, we'll show you how to make the priority SonarQube places on an issue line up with your own.

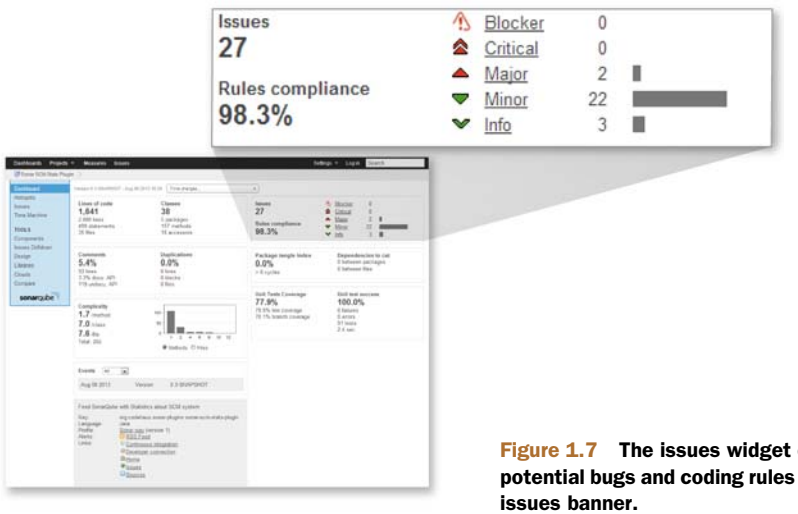


Figure 1.7 The issues widget combines potential bugs and coding rules under the issues banner.

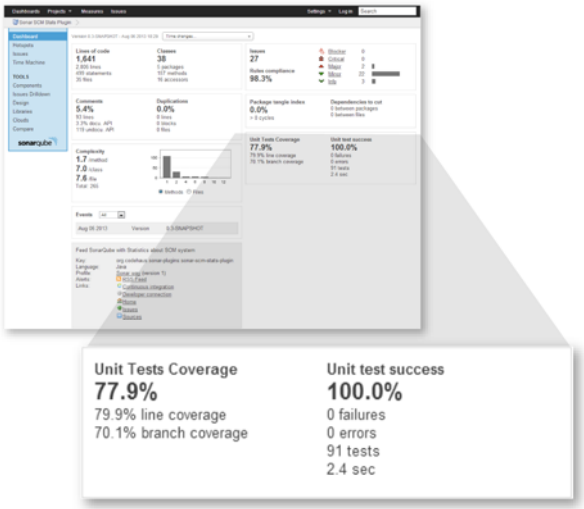


Figure 1.8 The test coverage widget reports on how well your code base is covered by unit tests, and how those tests are doing.

1.3.2 Tests

Next in the list is unit-test coverage, which is a bit like double-entry bookkeeping, in that each unit of work in your program should ideally be balanced by a test verifying that it works correctly. In fact, in test-driven development, the test side of the books is always entered first.

The test coverage widget, shown in figure 1.8, shows how well that coverage equation balances and whether your tests are passing, failing, or erroring-out. The percentages in this widget are bowling-style metrics, the failure and error counts are like golf, and the test count and duration are neutral. All the metrics here are leading indicators; if they head south, your quality may follow. For an in-depth look at unit tests, see chapter 3.

1.3.3 Comments and duplications

The comments and duplications widget is another two-for-one. To return to sports, comments are like bowling, and duplications are like golf. Both are leading quality metrics (nothing's gone wrong yet, but it could). The widget is shown in figure 1.9.

COMMENTS

There are two main types of code comments: the ones inline in any method (public or private) that are intended to notate some detail of the code logic, and the ones outside a public method that are intended to communicate how and why to use it (the API comments). The first kind is often referred to as a *code smell*. Like house guests and leftovers, this kind of comment tends to get stale. The logic changes, but the comments don't; or the comments get separated from what they refer to. The second kind of comment (the API documentation) is what's measured by SonarQube.

This is a measure of maintainability. It looks at how often you're going to make the caller of your method read the code to understand what she's getting into, versus

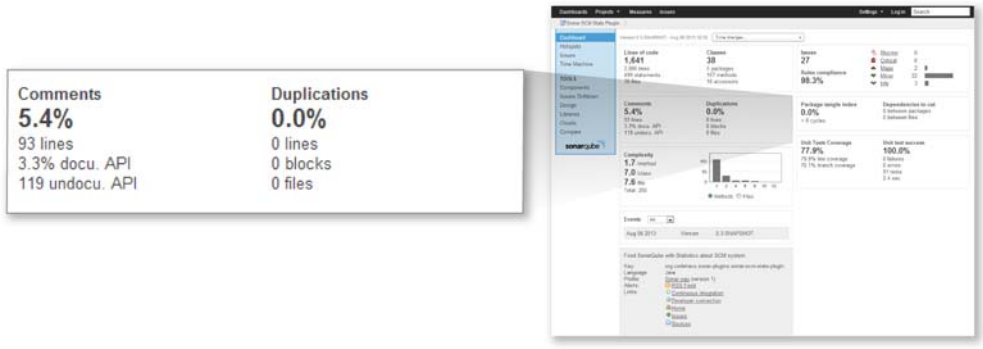


Figure 1.9 The comments and duplications widget covers two quality axes, showing both how well your public methods are documented (high scores are good) and how many duplications you need to eliminate (high scores are bad).

reading intentional documentation that (ideally) explains what should be passed in, what will be returned, and perhaps even what will happen in between.

Comments are measured because they're part of what makes a system easy (or not) to work on. They're measured with the idea that coders should spend their time writing the systems their users need, not trying to figure out what the last guy thought he was doing when he coded the method you need to call. We'll go in-depth on comments in chapter 5.

DUPLICATIONS

On the face of it, code duplications may not seem like a big deal. And at first, they might not be.

The problem is that although copy-paste, the source of duplications, is often expedient, it's not efficient in the long term. Somewhere in the same book that Murphy's Law came from is the truism that the more places a chunk of logic has been duplicated into, the more likely it is that it will need to be changed, probably with a high level of urgency or criticality.

That's why duplications are something you want to get on top of as quickly as possible, which is what SonarQube's duplications metrics let you do. Chapter 4 covers this topic in detail.

1.3.4 *Architecture and design*

Winston Churchill said, "However beautiful the strategy, you should occasionally look at the results." He wasn't talking about software quality, but he could have been.

One side of the architecture and design axis is the tidiness of a program's architecture. Not the way it was originally charted out: undoubtedly, the original plan had a Zen-like elegance and simplicity. What SonarQube measures is how it was implemented—how clean it is today. Do classes in package A include classes in package B, and vice versa? If so, either they should have been one package to start with, or you've got a big mess to sort out. Either way, you've got some cleaning up to do.

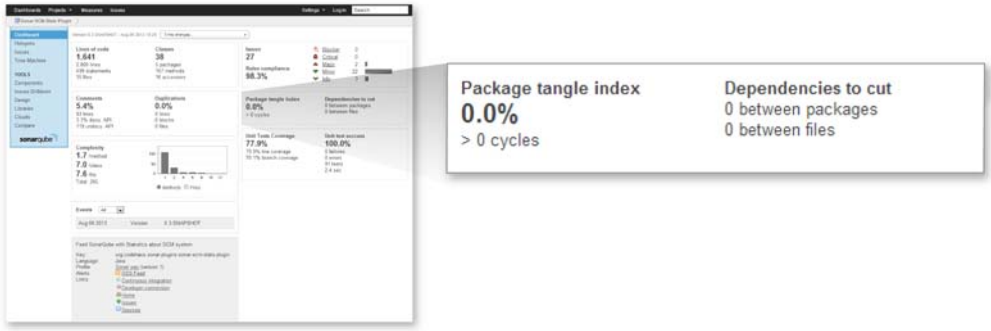


Figure 1.10 **The package design widget shows how clean your design implementation is, giving you high-level figures showing how much work needs to be done to make the implementation as clean as the original design undoubtedly was.**

From that perspective, whether architecture is a leading or lagging quality indicator is up for debate. Is this a measure of what has already gone wrong in the implementation (lagging), or an indication of how hard the code base will be to understand and maintain in the future (leading)? Either way, it deserves attention, and unless you've caught it early, it isn't likely something that can be cleaned up in an afternoon. The package design widget, shown in figure 1.10, gives you the high-level view of that cleanup in golf-style numbers.

The other side of the architecture and design axis is addressed by the LCOM4 and response for class widgets, and we're cheating a little by showing them to you here. They're no longer on the default dashboard because the creators of SonarQube think the concepts behind them are "too hard." Chapter 6 will make them seem easy, though. They're shown in figure 1.11. In a nutshell, you want to see these graphs weighted to the left because that means your classes are small and simple.

Because these are golf-style metrics, we'll use a golf example. Consider a `Ball` object. It should bounce. And maybe roll. But that should be about all it has to know how to do. Start layering in things like handicap calculation, club selection based on wind speed, distance to the pin, and grass friction, and you've probably gone too far. That's what the LCOM4 number is about. How many responsibilities does a given class have? One isn't the loneliest number in this case, it's the perfect number. Anything over two is definitely a candidate for refactoring.

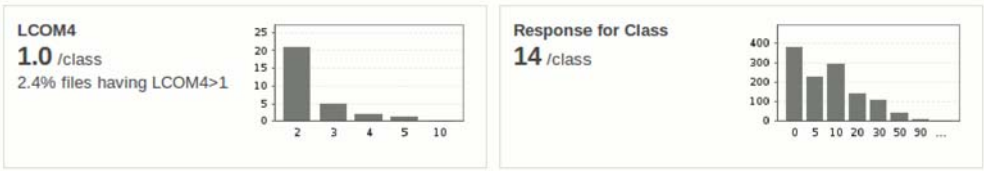


Figure 1.11 **The LCOM4 and response for class widgets show how your code stacks up from an object-oriented design perspective. Ideally, both graphs would be weighted to the left, meaning that the classes in your program are small and simple.**

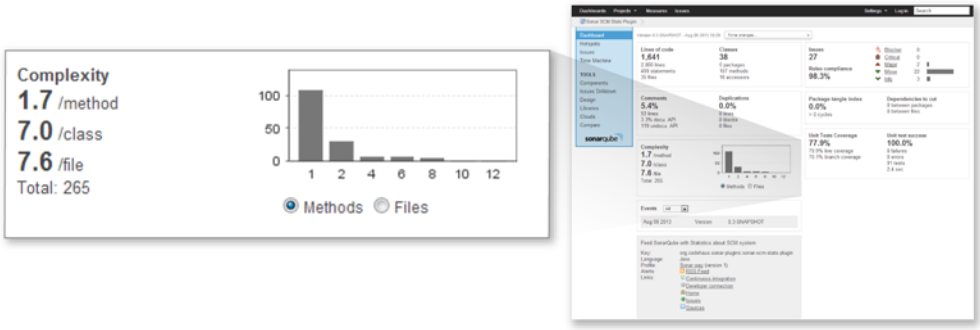


Figure 1.12 The complexity widget shows the distribution in your program of high-complexity methods or classes. The more complex a program is, the more difficult it becomes to maintain; so, ideally, this graph will be weighted to the left.

Response for Class (RFC) is also in the Keep It Simple, Smiley (KISS) realm, but a bit more esoteric. Something of a corollary to LCOM4, it's a measure of how many interactions a class initiates within itself or with other classes. For instance, if the `Ball` class reaches out to the `Grass` object to read friction, calls the `Green` object to see how far away it is, calls `getWindSpeed()` from the `Weatherman` class, and maybe even asks `Golfbag` for its list of clubs, it's initiating interactions with a lot of other classes. Not only is this a red flag from a design perspective, but it's also likely that the `Ball` class is harder to understand and therefore harder to maintain than it should be. One study of C++ programs even showed that as RFC went up, bug density did, too.

Architecture, design, and complexity are covered in depth in chapter 6 at the file level and in chapter 7 at the package level and above.

1.3.5 Complexity

Oddly, the explanation of the complexity axis is fairly simple. It's a little more complicated than this, but essentially, these metrics are about how many pairs of curly braces (real or implied) your method has. The premise of this leading indicator is that the more pairs of curly braces there are, the more complex the logic is. And the more complex the logic, the harder it is to understand and maintain. That means complexity is another golf-style metric—lower is better—and the complexity graph, shown in figure 1.12, is another that you'd like to see weighted to the left.

Now that you've seen the quality metrics SonarQube offers for its first language, Java, it's time to talk about what SonarQube offers for the rest of its languages.

1.4 The languages SonarQube covers

Earlier we said that SonarQube can analyze multiple languages. Now it's time to spell out what those languages are and what kinds of metrics are available for each one. Table 1.1 lists most of the other languages, but because the list of language plugins is always growing, we won't claim it's exhaustive. For each language, you'll see the

license model and the types of metrics (and therefore the quality axes) that are available for it.

Table 1.1 Languages SonarQube can analyze

Language	Paid/Free	Metrics
ABAP	Paid	Size, comments, complexity, duplications, issues.
C	Free (but closed source)	Size, comments, complexity, duplications, issues.
C++	Free	Size, comments, complexity, duplications, issues.
C++	Paid	Size, comments, complexity, duplications, issues. (This is not a misprint. There are both free and paid plugins for C++.)
C#	Free	Size, comments, complexity, duplications, tests, issues. C# analysis is made available by a suite of plugins, many of which rely on external tools that you'll need to install separately. On the other hand, this is the one case where you get to pick and choose which underlying tools to apply in your analysis. You'll need to install those underlying tools separately, as well.
Cobol	Paid	Size, comments, complexity, duplications, issues. Language-specific metrics such as outside and inside control-flow statements and LOC in data divisions.
Delphi	Free	Size, comments, complexity, design, duplications, tests, issues.
Drools	Free	Size, comments, issues.
Flex/ActionScript	Free	Size, comments, complexity, duplications, tests, issues.
Groovy	Free	Size, comments, complexity, duplications, tests, issues.
JavaScript	Free	Size, comments, complexity, duplications, tests, issues.
Natural	Paid	Size, comments, complexity, duplications, issues.
PHP	Free	Size, comments, complexity, duplications, issues.
PL/I	Paid	Size, comments, complexity, duplications, issues.
PL/SQL	Paid	Size, comments, complexity, duplications, issues.
Python	Free	Size, comments, complexity, duplications, issues.
Visual Basic 6	Paid	Size, comments, complexity, duplications, issues.
Web (JSP, JSF, XHTML)	Free	Size, comments, complexity, duplications, issues.
XML	Free	Size, issues.

The paid language plugins listed here all come from SonarSource, the originators of SonarQube. Regardless of the author of the plugin, most can be installed from within SonarQube, and chapter 14 will give you the details. Once you've installed your language plugins, configure your sonar-runner.properties file (be sure to specify the language under analysis with the `sonar.language` property!), and run your analysis. You don't have to do anything else; it just works. It's that simple.

Through the rest of the book, the majority of examples are Java-centric. But please keep in mind that unless we explicitly state that something's Maven-only or Java-only, it applies to other languages as much as it applies to Java (assuming the language plugin supports the metrics in question).

1.5 Interface conventions

If you've got the SonarQube dashboard in front of you, you've noticed that most of the metrics on it are links. And if you're even mildly curious, which we're betting you are, you've clicked-through on a few of them, from the 10,000-foot view at the dashboard to the low-level intricacies of the issues themselves. So at this point, we want to explain some of the interface conventions you're seeing and that you'll continue to notice as you work with SonarQube.

1.5.1 Hierarchy: packages and classes in a metric drilldown

Every metric on the dashboard clicks-through to a metric *drilldown*, which is designed to help you find the specific files (and sometimes the specific lines) that you need to work on to start moving those project-level metrics the dashboard reports. A typical drilldown is shown in figure 1.13.

Figure 1.13 Drilldowns in SonarQube start with the chosen metric and its value in the top row. That's followed by a row of hierarchical widgets: modules (if any—there aren't any here) in the left-most box, then directories/packages, then classes/files. Each widget's contents are sorted by its metric value. Click any module or package to filter widgets to its right.

There are some variations on the theme, but generally you'll see the metric in question at the top of the page followed by a row of hierarchical widgets. If it's a multi-module project, then this row will have three widgets, with the modules on the left and packages/directories in the middle. For single-module projects, there are only two widgets, with packages/directories on the left. In either case, a list of files is shown on the right.

Each widget is sorted by metric value with the worst first. You can click a module to filter the directory and file lists, or click a directory/package to see only its files. With or without module and package filtering, you can click a filename at any time to see its details.

1.5.2 File details

When you click a file, the file detail view, shown in figure 1.14, is added to the page below the module/package/file hierarchy. You see the full filename at the top, followed by a series of links, which act like tabs. Which link is selected depends on the metric under examination.

Some of the metrics, such as duplications, have dedicated tabs in the file detail view that are tailored to that metric's clear communication. Other metrics, such as complexity, take you straight to the Source tab. When you end up at the Source tab, it's typically because the metric in question relates to the file as a whole, rather than a small section of it, and there's no good way to zoom in on the issue. Instead, SonarQube tells you what you're looking for and then shows you the source so you can see for yourself how complex the class is or how documented or undocumented the API is.

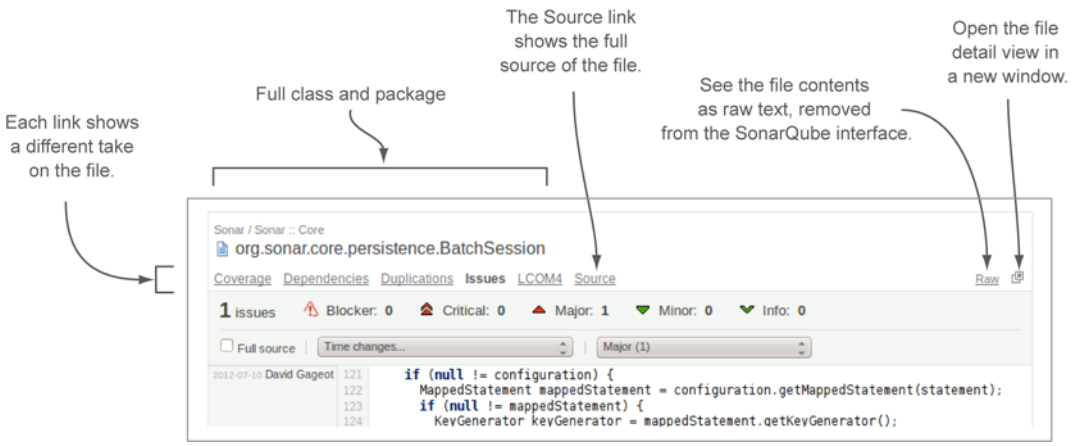


Figure 1.14 The file details view offers a consolidated spot to see most of the metrics SonarQube's gathered on a particular file. The Source tab shows the file's full contents.

1.5.3 Trend arrows

The final interface convention to show you is the trend arrows you'll see throughout the interface after your second analysis, starting with the front page. They come in three colors—red, green, and grey—and if you guessed that the colors mean bad, good, and neutral, you're right.

> **When changes take effect**
>
> As we show you how to take full advantage of SonarQube in the coming chapters, it will be important to keep in mind that almost none of the changes we'll show you how to make to SonarQube will take effect *until the next analysis*. That's because SonarQube mostly shows you metrics, and metrics are only calculated during analysis; so you can twiddle settings all you like, but they won't affect what's already been calculated and stored. To see the effect of your changes, you'll need to reanalyze.
>
> Just installed a new plugin and eager to see the results? Wait until the next analysis. Tweaked your settings and looking for the change? Wait until the next analysis. Moved your project from one rule set to another? Wait until… Oh, you get it. Okay.

Trend arrows show the 30-day trend of a given metric. You won't see them after the first analysis of a project because there's no trend yet. Make some code changes and re-analyze, and they should pop into view. An arrow alone shows a moderate change, and an arrow with a line indicates a strong one. Figure 1.15 shows a project's size metrics widget, with strong to moderate increases in all metrics.

Metrics that are expressed as percentages, such as the rules compliance index, or averages, such as complexity/method, aren't eligible for trend arrows. Otherwise, when you don't see an arrow, it means there has been no change, or only a weak one.

Those with sharp eyes have noticed that we've only scratched the surface when it comes to the options in the SonarQube interface. For instance, a double-handful of left-rail links were spread across the front page and in the project dashboard that we haven't even touched on yet. Don't worry, we'll get there—eventually.

Figure 1.15 Trend arrows are used throughout the SonarQube interface to indicate the 30-day trend of a metric. Red, green, and grey arrows indicate bad, good, and neutral changes. Arrows alone show moderate increases. An arrow with a line shows a strong increase.

1.6 Related plugins

Out of the box, SonarQube is a pretty incredible tool. But there are plugins that can make it even better. We'll end almost every chapter with a list of plugins related to the functionality discussed in the chapter, and we'll tell you how they enhance the relevant aspect of SonarQube.

If you decide to add any of these plugins, you'll find that installation is pretty easy for a logged-in administrator from within SonarQube (see chapter 14 for details). Now we'll talk about our first couple of plugins: Technical Debt, and Views.

1.6.1 Technical debt

At the heart of SonarQube is the concept of *technical debt*: the cumulative cost of work that's been put off or done poorly enough that it needs to be refactored. Because measuring technical debt is a core principle of SonarQube, it's not surprising that its creators would come up with a set of technical debt metrics—in dollars and days. What is surprising is that it's not included in the core functionality, but available instead as a plugin.

Once you've installed the plugin and restarted SonarQube, add the widget to your dashboard and run a new analysis. When it's done, you'll see something like what's shown in figure 1.16 on your dashboard.

The dollar and man-day values come directly from the other metrics on your dashboard. How many issues and duplications do you have? How much of your API is uncommented? How out-of-hand are your design and complexity?

The Technical Debt plugin takes all those numbers and multiplies each one by an estimate of how long it will take to fix an average issue of each type. That gives an hour figure, which is easily turned in to man-days. For the dollar number, the plugin multiplies by the configured cost per man-day. Simple, but often painful, calculations.

The percent figure is called the *debt ratio*. It's the total debt, divided by the total possible debt—the worst-case scenario—times 100. Presumably, it's included to ease the sting of the other numbers.

Each of these calculations is based on tunable estimates. The daily rate of a developer defaults to $500, but it's easily changed. Time to fix an average coding issue defaults to six minutes, but again, it's tunable. Chapter 14 will show you how.

Figure 1.16 Technical debt is designed to communicate the full liability of unaddressed issues in your code base clearly—what they could potentially cost you, and where they're coming from.

1.6.2 Views

Although SonarSource offers the Technical Debt plugin for free, it charges for the Views plugin. What makes it compelling enough that you may want to pony up is the cross-project aggregation it offers.

The Views plugin's functionality is most likely to appeal to managers and executives who need quick visibility of SonarQube's leading and lagging quality metrics across multiple projects in a group, or across multiple groups. There's no other way to get this cross-project view except to manually update spreadsheets, a tedious and error-prone process.

Instead, the Views plugin offers those aggregations not just on "report update day," but at a whim at the filter (list of projects) level. It also provides a dashboard for each aggregation that shows totals and percentages calculated across every member of the collection. Once you start drilling in to an aggregate view (clicking-through from the dashboard), you'll get aggregate drilldowns that let you identify culprit projects while retaining the ability to continue drilling down to see the granular issues in the projects and files themselves.

1.7 Summary

Measuring software quality used to be hard. Instead of even trying, people did silly things like measuring lines of code instead. Then the creators of SonarQube, the SonarSource folks, came along and used existing tools that find problems in software and derived metrics from their output, making quality trackable and trendable. They added their own tools and metrics as well. They wrapped all those metrics in SonarQube's intuitive interface, added review functionality and IDE integration, and … gave it away.

The beneficiaries of this generosity are the developers, testers, code architects, and managers whose teams use SonarQube. Oh yes, and end users too.

SonarQube was originally written to analyze Java, but plugins extend the offerings to an ever-growing list of other languages. Each SonarQube "project" is about a single language, but you can use multiple properties files to analyze every aspect of a complex project. For instance, you can get quality metrics not just for the Java back end of a web application, but for its XML and JavaScript, too.

In this chapter we've walked through your first SonarQube analysis using the SonarQube Runner and a basic properties file. Once the analysis was complete, we looked at the results on SonarQube's front page, the default filter, and drilled down to the details on the project dashboard.

The default dashboard is centered on SonarQube's Seven Axes of Quality:

- Potential bugs
- Coding rules
- Tests
- Duplications

- Comments
- Architecture and design
- Complexity

After getting a high-level understanding of each axis, you saw a few interface conventions that you'll be seeing regularly; the package/class hierarchy in metric drilldowns, the file detail view that's added below it, and the trend arrows you see on filters and dashboard widgets.

In the next chapter we'll focus more deeply on those first two quality axes, with a look at issues and why they should never be ignored.

2 Issues and coding standards

This chapter covers

- Looking at your issues
- What issues mean
- Where issues come from

If you're only using your users' bug reports to measure the bugginess of your code, then you're only seeing a tiny sliver of the picture—because users can only report what they can perceive. Any user will recognize a program crash, but what about gradual performance degradation caused by unclosed database connections?

The report that comes back from the users typically sounds like "sometimes it gets slow and we have to restart." Which could, of course, mean anything.

In this chapter, we'll look at issues: programming errors that users aren't necessarily noticing…yet. Pay attention to coding rule issues, and you can head that bug report and many others off at the pass by preventing problem code from ever reaching the user.

The term *issues* covers two of SonarQube's Seven Axes of Quality: potential bugs, and coding standards. We'll start with what SonarQube tells you about issues

on the dashboard and in the issues drilldown. Then we'll look at why each issue is a potential problem, even the ones you might be tempted to shrug off.

Next we'll take a brief look at rule profiles—the sets of rules against which your code is measured in an analysis—and how to change the defaults. We'll round out the chapter with a summary of issues-related plugins that might be of interest.

2.1 Looking at your issues

Bob is one of your best customers, and he says the app is crashing when he opens the Edit window. But only sometimes. Your testers can't reproduce the crash, but you have to believe Bob because he sent you screenshots.

The best suggestions at this point are a line-by-line audit of the code, which could take weeks and still not produce anything, or flying someone out to watch over Bob's shoulder as he works.

Neither idea is popular with upper management. Now what?

Now it's time to let computers do what they do best: automate intricate, detailed, and mindlessly tedious tasks. It's time to turn to SonarQube for a look at your issues.

After your first analysis, you'll likely see hundreds of issues on your SonarQube dashboard. Each one represents some anti-pattern in your source or compiled byte code. Not every issue is a bug, but it's something that needs further attention. Also, addressing issues is the quickest road to higher-quality software.

Let's start from the dashboard, where you see something that looks like figure 2.1. This is the issues widget, which reports on your project's adherence to a rule profile. At upper left is the raw count of issues. To the right is the issue breakdown by severity. At lower left in the widget is the project's Rules Compliance Index, a calculation based on the number and severity of issues versus the number of lines of code in the project. It's calculated with the formula shown in figure 2.2.

The Rules Compliance Index is a gut-check type of metric that gives perspective. After all, 465 issues in a project with 10,000 lines of code is entirely different from the same issue count in a 500-line project. The problem with Rules Compliance is that it can fluctuate without the number of issues ever changing. For instance, if you add lines of issue-free code, your index goes up. At a glance, you might think you've improved the code base, when all you've really done is dilute the problem. For that reason, we won't spend more time on the Rules Compliance Index.

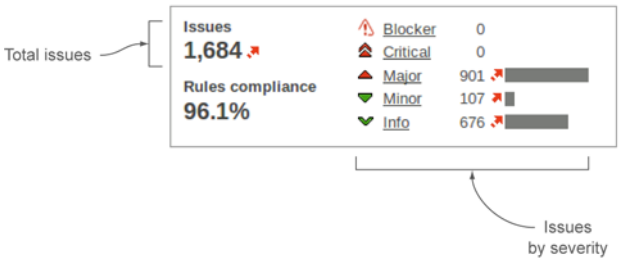


Figure 2.1 The issues widget gives the total number of issues, the breakdown of that count by level of severity, and the Rules Compliance Index.

$$100 - \left(\frac{(\text{Blockers} * 10) + (\text{Criticals} * 5) + (\text{Majors} * 3) + \text{Minors}}{\text{Lines of code}} \right) * 100$$

Figure 2.2 The Rules Compliance Index is the Weighted Issues score (your counts of issues multiplied by severity factors), divided by the number of lines of code in the project, turned into a percentage, and subtracted from 100.

Instead, we'll focus on the issues themselves. To see what the issues in your project are, you can either choose Issues Drilldown in the left menu or click-through in the issues widget on the Issues total at upper left or on one of the individual issue severities. Figure 2.3 shows you the issues drilldown with the Major severity highlighted. In chapter 1, we showed you a metric drilldown for complexity that looked a lot like the bottom half of figure 2.3. That's because the package and file/class hierarchy are standard interface features you'll see repeated from drilldown to drilldown.

What's specific to the issues drilldown is the severity and rule hierarchy across the top. If you got here by clicking Issues Drilldown in the left menu, then none of the severities are highlighted, and you're seeing all issues in the Rule box at upper right, grouped by severity (worst first) and sorted by count.

To narrow the list of rules, click a severity. It will not only filter the rules at upper right, but also limit the package list to only packages with issues of the selected severity, and similarly limit the class list to only classes with issues of the chosen severity.

It works the same way with the rules. Click a rule to filter the packages and classes to show only ones with issues of that rule. And of course, clicking a package filters the

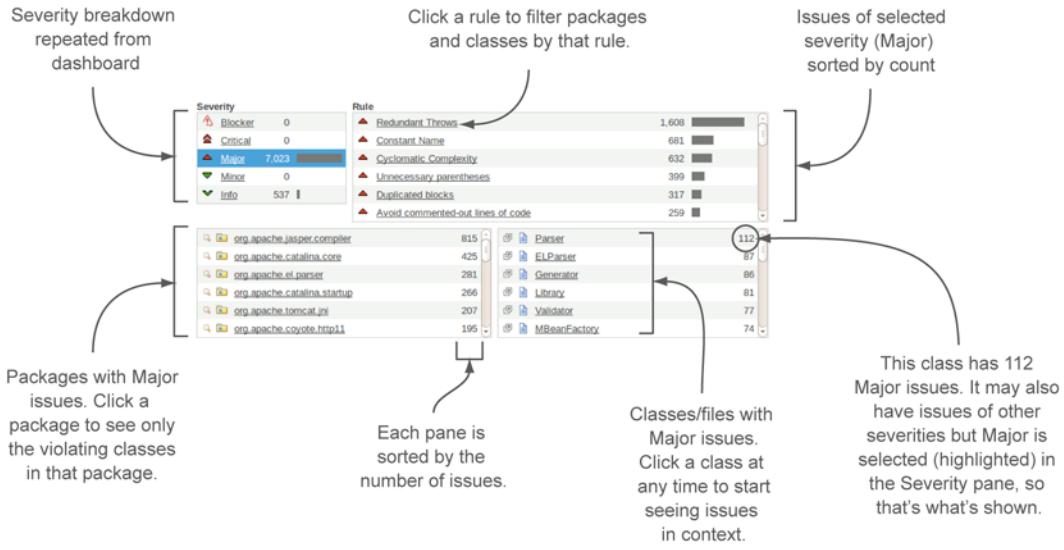


Figure 2.3 The four widgets you first see in the issues drilldown represent a hierarchy: severity, rules, packages, and classes/files. Each widget after Severity is sorted by the number of hits against the items listed. Click at any level to filter the widgets lower in the hierarchy. Click a file or class name to begin seeing issues in that file at any time.

class list by that package. If you find that you've filtered too much, you can re-click your choice at any level to remove that filter and the ones below it in the hierarchy.

At any point, whether you've filtered the list or not, you can click a filename to begin seeing the issues in the file detail view, which is added to the bottom of the screen when you click a filename. If you did click a rule first, you're only shown issues of that particular rule in the file you've chosen. Other issues may exist, but they're filtered out initially. To see them, the easiest thing to do is consult the All Issues drop-down at upper right of the file detail view, shown in figure 2.4. You can use it to show all the issues in the file, or all the issues of a particular severity or of a particular rule.

For each issue, SonarQube not only shows what the problem is and where to find it, but also gives a few lines of context around each issue. If the context shown by default around the issue isn't enough for you to understand the problem, you can use the Full Source check box at upper left in figure 2.4 to show the issues in the context of the whole file.

By default, SonarQube shows you the "what" and the "where" of a issue. If you're using the SCM Activity plugin, it can also show you "who," as shown in the left column in figure 2.4.

SCM stands for Source Control Management, and the plugin integrates with many of the popular source control repositories, such as Git, Subversion, and CVS. It doesn't come bundled with SonarQube, but it's maintained by the authors of SonarQube, and it's a free download.

Now that you know how to find your issues, let's dig in to what they are.

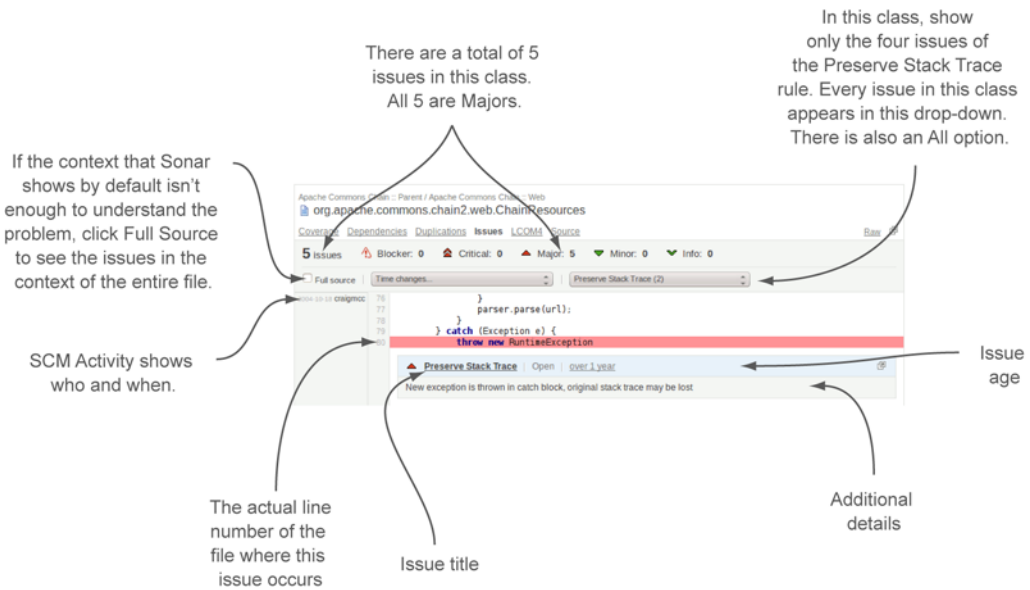


Figure 2.4 Once you click a file, you're shown the issues in that file with a few lines of context on each side. If you have the SCM Activity plugin in place, you see not only the problem code and its line number, but also who checked it in and when.

2.2 What issues mean, and why they're potential problems

The issues shown earlier were recorded during a SonarQube analysis. As one part of the analysis, SonarQube compares your code to a set of rules. When a rule is broken, an *issue* is marked against the line where it occurred. Issues are reported at one of five severities: Blocker, Critical, Major, Minor, and Info, and the severities generally mean what you'd think they would.

Severities are attached to the rules being checked, not to the issues themselves. So you won't see SonarQube report one issue of rule A as a Blocker and another issue of rule A as a Minor.

> **NOTE** There are exceptions. Two rule profiles might include the same rule at different severities, and it's possible to change the severity of an individual issue. See chapters 10 and 13 for details.

For the sake of discussion, it's useful to break the issues SonarQube reports into six general categories. They're listed here in approximate order of importance:

- Bugs
- Potential bugs
- Indications of (potential) programmer error
- Things that may lead to future programmer error
- Inefficiencies
- Style inconsistencies (future productivity obstacles)

Very often, issues that represent bugs and potential bugs show up in SonarQube as Blocker or Critical issues. After that, you can't necessarily match up the severity of an issue to the categories we've outlined here. That's particularly true because the severity of an issue is editable (we'll show you how in chapter 13), and the importance (severity) that one person or team places on a rule may differ from another's. To make this clearer, figure 2.5 gives a general overview of severities versus categories.

For instance, some things that would fall under "inefficiencies" or "future programmer error" are flagged by SonarQube as Major issues, and others show up as Minor. So although the severities are important, we're going to set them aside for the purpose of this discussion.

Next we'll look at each category and tell you why it's important. We'll also give you a few examples of issues in the category, but don't think that these lists are

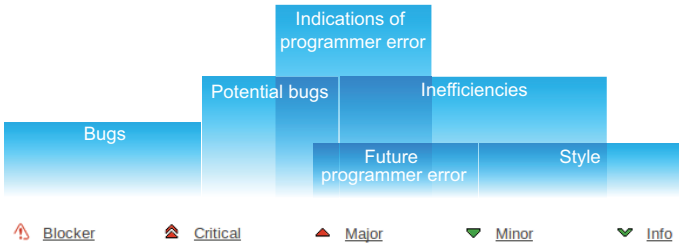


Figure 2.5 SonarQube's issue severities match up only loosely with the categories we'll use in this chapter to discuss the kinds of issues that are flagged.

exhaustive. Unfortunately, the lists of things you can get wrong are far longer than we have space for.

2.2.1 Bugs

The first (and worst) category is bugs. Issues in the bugs category are guaranteed to be problems. Your users might not be complaining about them, but if so, that's because they just haven't noticed them yet.

Things in the bugs category include the following:

- Logic errors that would lead to null pointer exceptions
- Failures to close file handles or database connections
- Bad behavior in a multithreaded environment
- Methods designed to check equality that always return false (or true)
- Impossible class casts

Figure 2.6 shows one such issue flagged in SonarQube.

Any piece of code flagged with the issue shown in figure 2.6, "Correctness–Null pointer dereference," is guaranteed to be a problem. Full stop. No horsing around.

Sure, the particular example of reversed logic shown in figure 2.6 is trivial enough to verge on silly, but it's the kind of thing that's easy to do when your mind is elsewhere. And trivial or not, this example represents a very serious category of issue, because these bugs are always guaranteed to have unpleasant results for the user.

At best, issues in the bugs category result in performance degradation that eventually requires a restart, and Murphy's Law dictates that it will come at an inconvenient time. At worst, your users are probably dealing with weird program behavior and error messages, or outright program crashes.

2.2.2 Potential bugs

As with bugs, potential bugs represent actual problems in the code. Often, though, they're conditional problems, ones that will only happen some of the time—which is probably how they get past the programmers' own testing.

Figure 2.6 **Null pointer dereferences fall into the bugs category.**

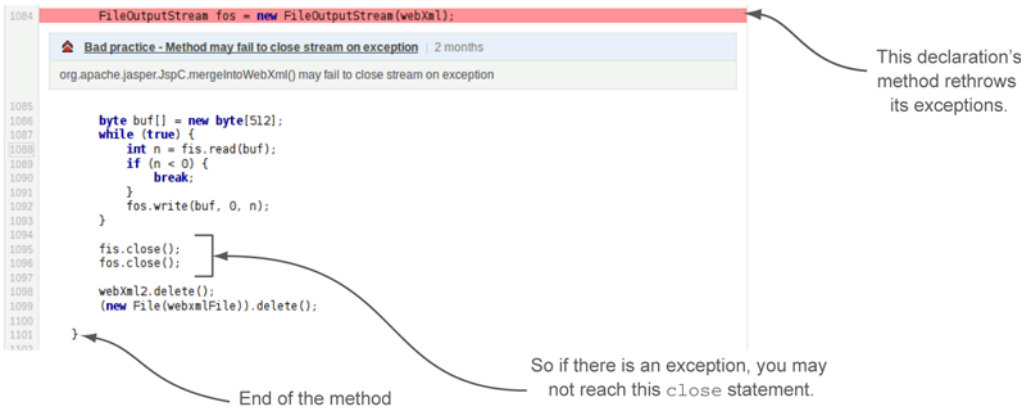


Figure 2.7 If everything works well, on line 1096 this method will close the `FileOutputStream` that was created on line 1084. But if an exception is encountered in the middle, that resource leaks instead. If this happens once or twice, you may never notice. Let it go on long enough, though, and you'll bring your application to its knees.

You're even less likely to get coherent bug reports from the users about these problems, but they're problems nonetheless. Figure 2.7 shows one such issue.

We had to use the Full Source check box we mentioned earlier to see enough of the code to understand the problem shown in figure 2.7. Once we did, we saw that the code will work fine some of the time, when everything works as it should and no exceptions are thrown. But when an exception *is* thrown, this code starts leaking resources like a sieve, because the method re-throws all its exceptions without pausing to clean up after itself first. This is the sort of thing that may happen only 1% of the time—a corner case—but once it's pointed out, it's easy enough to fix, and there's really no reason not to.

In addition to potential resource leaks, the potential bugs category includes things like these:

- *Potential* null pointer exceptions, which happen only under certain conditions
- Null checks that dereference the items they're checking
- Math operations that use the wrong precision or lose precision

Now that we've covered the two issues categories for things that are flat-out wrong, what's left are the categories with problems that typically are just as serious but that may not be quite as urgent.

2.2.3 Indications of (potential) programmer error

Unlike the issues in the first two categories, the ones in the indications of programmer error category aren't *guaranteed* to cause problems. And at first blush, they may tempt you (or developers you work with) to dismiss the entire enterprise as trivial, because they flag code where there's nothing technically *wrong*. Figure 2.8 shows an example.

If you skimmed past it, take a minute to actually read the code in figure 2.8. On line 22, a conditional checks whether the day in question is Tuesday. Then, on line 24,

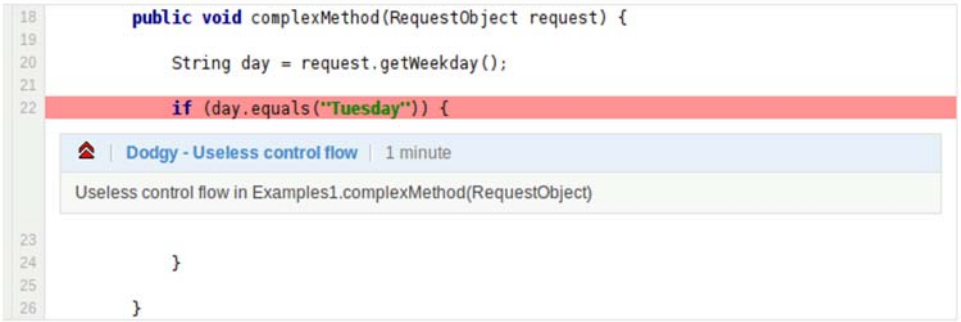


Figure 2.8 **An empty conditional may mean the coder forgot to do something.**

the conditional block is closed. There's nothing in between the open curly brace and the close curly—except the issue block that SonarQube overlays into the code.

The code shown in figure 2.8 will run just fine. But … don't you have the sense that something is missing? At the very least, you're probably wondering what the programmer *meant* to do on Tuesdays.

That's why this gets flagged by SonarQube. Not because there's anything demonstrably wrong, but because there's an indication that the developer might have made a mistake.

For instance, what if it should have looked like this?

```
public void complexMethod(RequestObject request) {
  String day = request.getWeekday();
  if (day.equals("Tuesday") {
    processPayroll();
  }
}
```

Clearly, forgetting to process payroll is a serious omission.

A small subset of issues in the indications of error category has to do with null checks: null-checking a variable that has already been dereferenced, re-null-checking something you already know is null, and re-null-checking something you already know isn't null. Coders who cut their teeth on C may bridle at these, because they've long lived by the motto "Null-check early; null-check often." The reason SonarQube flags these null-check instances, and the reason they're included in this category, is that they look like evidence of rearranging the code without really reading it.

Other issues in this category include

- Comparing objects with `==` or `!=` rather than the `.equals()` method
- Conditionals that assign rather than compare, such as `if(count=1)`, which always returns true, versus `if(count==1)`, which is truly a conditional
- Unused members or methods
- Catch blocks that swallow exceptions rather than logging or passing them on

Next we'll move from the issues that tell you a coder may have tripped up to the ones that mark the bumps in the road ahead—where coders may trip in the future.

2.2.4 *Things that may lead to future programmer error*

Issues in the future programmer error category strike some as purely questions of style. Others will recognize them for the traps that they are. Figure 2.9 gives an excellent example.

Figure 2.9 shows a conditional without curly braces. Like some of the previous examples, figure 2.9's brace-less conditional will run just fine. And many coders prefer *not* to use curly braces around one-line `if` statements (and `else` statements, and `where`s, and so on). They argue that it makes for cleaner-looking code. Certainly for the alert and focused programmer, the code shown in figure 2.9 is clear, concise, and readable.

The problem is that not every programmer is alert and focused. The next guy who comes along may be more used to one of the languages where the indents count and not the curly braces, or he may be focused more on what's for lunch than the trivial change he needs to make, and he may do something like this:

```
if (day.equals("Saturday") || day.equals("Sunday")
    playLotto();
    sleepLate();
```

If you're paying attention only to the indention, like our hungry coder, you may think this program now has you sleeping late and playing the lottery only on weekends.

In fact, because only the first command after a brace-less `if` statement is conditionally executed, what it actually does is have you play lotto on the weekends and sleep late every day of the week. Now you've not only squandered all your money on lottery tickets, you've also started missing work.

Then there are the rules that check naming conventions. They make sure your class names start with capital letters and your method names don't. They look for constant variables that are named in all caps and make sure your non-constant variables aren't.

These may seem like questions of style, which are covered shortly, but they're included here because straying from these industry-standard conventions will lead

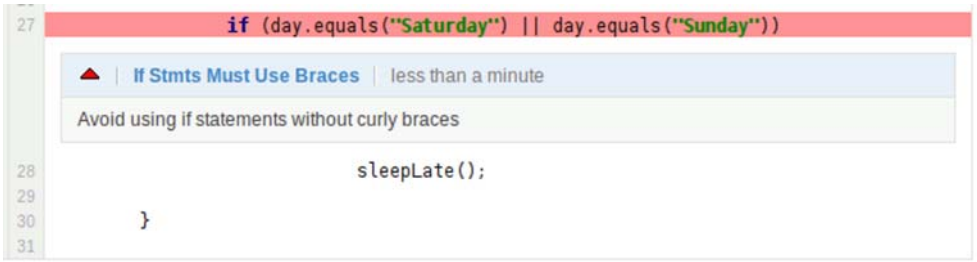


Figure 2.9 Conditionals without curly braces could lead to future programmer error.

experienced programmers to make false assumptions about code they're not familiar with—assumptions that at best will waste their time and at worst will lead to the kind of subtle bug that can be very difficult to track down.

Also in the future programmer error category are the things that will make the next person's job harder to do when she gets assigned to maintenance on your project:

- Classes that try to do too much and need to be chopped up
- Methods that are too long and complex
- Conditionals with too many clauses

Future programmer error issues can make it difficult to maintain existing code correctly and efficiently. The issues in the next category can make it difficult for your program to run efficiently.

2.2.5 Inefficiencies

No issues in the inefficiencies category will keep your program from performing correctly. Nor will they lead to future problems. What they *will* do is keep your program from performing at peak efficiency. And although that may not be a concern today, it could become important as your program's user base grows or its share of the CPU shrinks.

A set of issues in this category that deserves special mention, at least in Java, is the inefficient use of strings. Using strings poorly is sadly easy to do in Java and particularly likely for coders who come to Java from other languages. For instance, concatenating strings in a loop is standard operating procedure in some languages. But it's such a bad idea performance-wise in Java that a special rule checks for it. Figure 2.10 shows another rule in the strings group.

Also in this category are

- Unneeded import declarations
- Suboptimal use of the special Java math objects like `BigDecimal`
- Unused members and methods

If you're alert, you noticed that "unused methods" is listed in the indications of (potential) programmer error category as well. There are many issues that could span multiple categories, and this is one of them. In fact, many people would lump everything in the next category, style inconsistencies, into the future programmer error category.

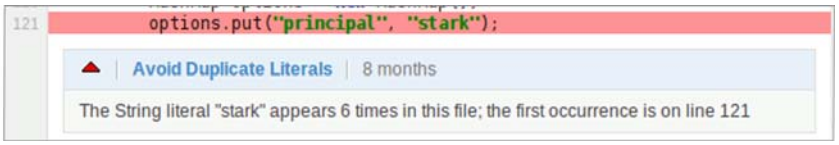


Figure 2.10 Strings are easy to abuse in Java, and SonarQube offers several rules to ensure that you're using them efficiently.

2.2.6 Style inconsistencies (future productivity obstacles)

The issues in this category are the stuff of which holy wars are made. For instance, which line does the left curly brace go on? This one?

```
if (day.equals("Monday")) {
```

Or this one?

```
if (day.equals("Monday"))
{
```

To add spaces?

```
if ( day.equals( "Monday" ) )
```

Or not?

```
if(day.equals("Monday"))
```

The idea behind these style rules is the same one behind consistent placement and presentation of traffic signals; once you learn the system, your eye zooms past the presentation ("There's a sign up ahead") and straight to the critical data being presented ("It's red; I should stop!").

Start changing sign shapes or colors, and it takes longer to get to the meat of the matter. Similarly, developers spend most of their day reading code. The extra time required to take in code that's not formatted in the style the team agreed to, multiplied by the number of files they read and the number of developers in an organization, could work out to a lot of lost productivity.

We've listed the bug categories here in order of the quickest bang for the buck, which is also (in some minds) their order of importance. But others would argue that the categories we put at the end of this list are just as important—even more so in the long term—as the ones at the start, because they relate to the maintainability of the code. Left unfixed, these issues could make your code difficult to maintain, favor the introduction of bugs (because of low readability and high complexity), and, little by little, make the business reticent to make changes because they're costly and usually end up breaking something.

Now that you've got a feel for the kinds of rules SonarQube gives you, we'll look at where they come from and how SonarQube organizes them.

2.3 Where do issues come from?

You may be aware that very good open source tools are available to analyze code and find anti-patterns. You may have even used them in the past. But when's the last time you remembered to do it? Even if you're analyzing regularly on your own, what about your teammates? Are you all using the same tools? With the same options? Chances are, the answer is no.

Standardizing the answers to those questions is one of the great benefits SonarQube offers. In Java and C#, it provides a few rules of its own, but the majority of the rules SonarQube uses come from the major rules engines: FindBugs, PMD, and Checkstyle for Java; FxCop, Gallio, Gendarme, NDeps, and StyleCop for C#. For most other languages, the rules come directly from within the plugin.

2.3.1 Picking a rule profile

Even though SonarQube uses multiple external rules engines for Java and C#, it doesn't turn on all three tools full-bore. Instead, it packages selected rules from each tool into what it calls *profiles*. Multiple profiles can exist per language, and one profile is always set as a default for that language. The default for a language almost never includes every single rule that's available for the language. New projects and projects that haven't been specifically pinned to a rule profile (*unassigned* projects) are measured against their language's default.

For Java, SonarQube provides three profiles: Sonar way, Sonar way with FindBugs, and Sun checks. It sets Sonar way as the default.

The Sun checks rule set is a small one, weighing in at only 58 rules. All it does is check source code against the Sun Java coding style conventions. For example, it checks member- and class-name capitalization, curly brace position, and the use of spaces.

The difference between the other two (the two versions of Sonar way) is the inclusion or exclusion of FindBugs rules. It's a big difference. Checkstyle and PMD scan your uncompiled .java files. By contrast, FindBugs (and some of the SonarQube-native rules as well) runs against your compiled byte code, the .class files. During a SonarQube analysis, if FindBugs is invoked (if any of its rules are included in the profile), it performs a static analysis of every possible path through the program: no stone unturned, and no execution necessary. Similarly, the C# suite also includes static analysis of compiled code.

The bugs from static analysis tools like FindBugs are the "best" ones, the ones you *really* want to catch, because those tools find the problems that are the most likely to lead directly to bad program behavior. For instance, the Java issue "Correctness - Null value is guaranteed to be dereferenced" comes out of FindBugs, and without FindBugs in your profile, you'll have a much harder time finding and fixing these rotten apples.

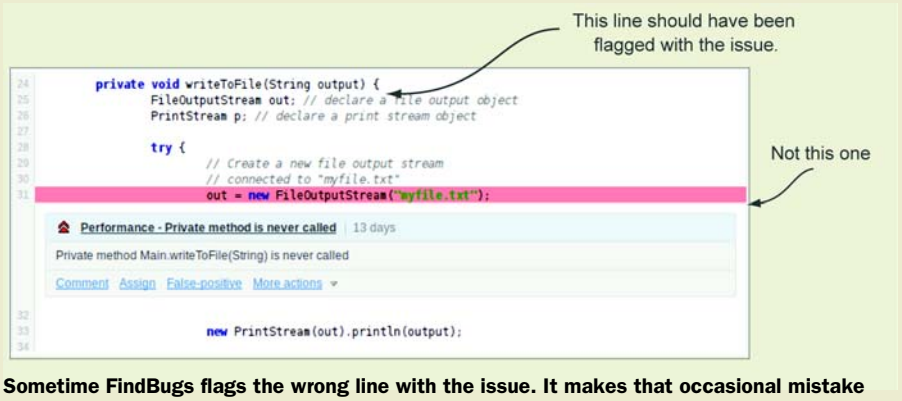
Clearly, you want to use the FindBugs rules if you have access to the .class files during analysis. Unfortunately, because that access can't be assumed, SonarQube's out-of-the-box default is the less demanding Sonar way profile.

Assuming you have the .class files handy, you should use the Sonar way with FindBugs profile instead of the out-of-the-box default. The most expedient way to do that is to change SonarQube's default Java profile.

FindBugs presents minor challenges

FindBugs analyzes your compiled byte code, not the Java source. This means you need to be aware of two things.

First, the issues it finds may sometimes be shown in SonarQube attached to the wrong line of code. When that's the case, look one or two lines above and below the line SonarQube flags, and you'll find the real source of the problem (as shown in the following figure).



Sometime FindBugs flags the wrong line with the issue. It makes that occasional mistake because it's working from the compiled byte code, not the .java files. When this happens, look a line or two above and below to find the real problem.

Second, for FindBugs and SonarQube to be able to make any kind of connection at all between a issue and a line of source code, you need to have `debug` turned on when you execute your `javac` command. (It's off by default if you're using Ant.) This has the effect of embedding line-number information in your byte code. FindBugs can then use this information when recording the issues it finds. (Of course, this implies that it's not FindBugs that occasionally gets line numbers wrong, as we said earlier, but `javac`.)

Be aware that turning on `debug` in your compile will have the added effect (typically considered a benefit!) of adding line numbers to any stack traces, thus making it easier to track down and fix your exceptions in production.

2.3.2 *Viewing profiles and changing the default*

Changing a language's default profile is as easy as the click of a button for a logged-in administrator. The Configuration link at upper right on the screen takes any user to the list of profiles, as shown in figure 2.11.

This means any user can see what the default profile is for a language (the one with the green check mark) or peruse the profile a project is being measured against (it's listed in the Description widget on the dashboard). But only administrators see the columns of controls shown on the right in figure 2.12.

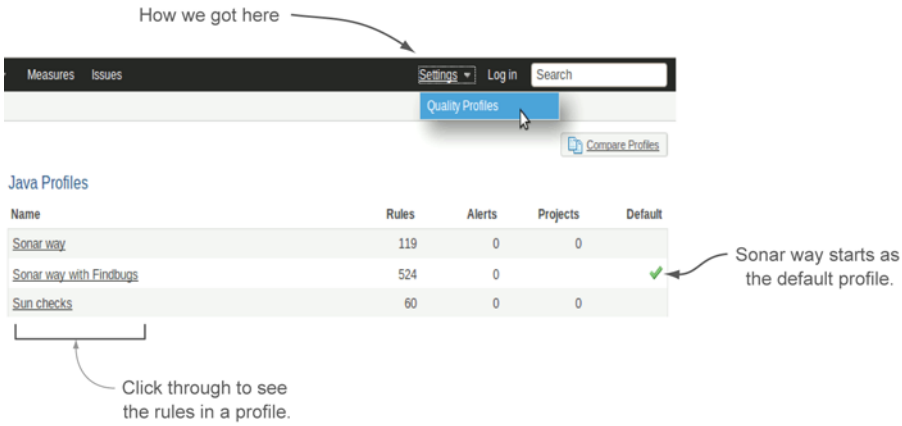


Figure 2.11 **Any user can see the rule profiles by using the Configuration link at upper right of the screen.**

Your first analysis of a project will run against its language's default profile, which we hope you've just changed. Once it's established in SonarQube, an administrator can easily assign a project to a different profile, and all subsequent analyses will run against it even if it's not the default.

Even if your project's preferred profile *is* currently set as the default, it's not a bad idea to also explicitly assign the project to that profile. To do that, start by clicking-through on the profile name and then choosing the Projects tab, as shown in figure 2.13. Once your project is assigned to the proper profile, you'll still get the analysis you expect, even if the default changes.

When you begin analyzing your projects against a rule profile, you may find there are rules in the profile or rule-severity assignments you don't agree with. One option is to switch to another profile, but a better option is customizing your rules. For details, see chapter 13.

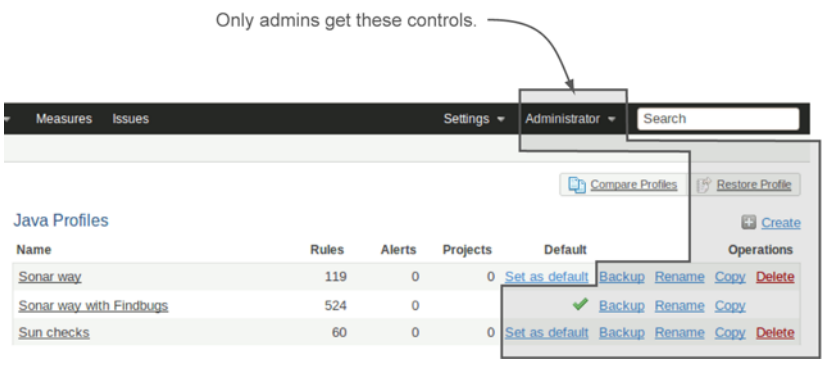


Figure 2.12 **Administrators have the ability to create and edit rule profiles and change which profile is the language's default.**

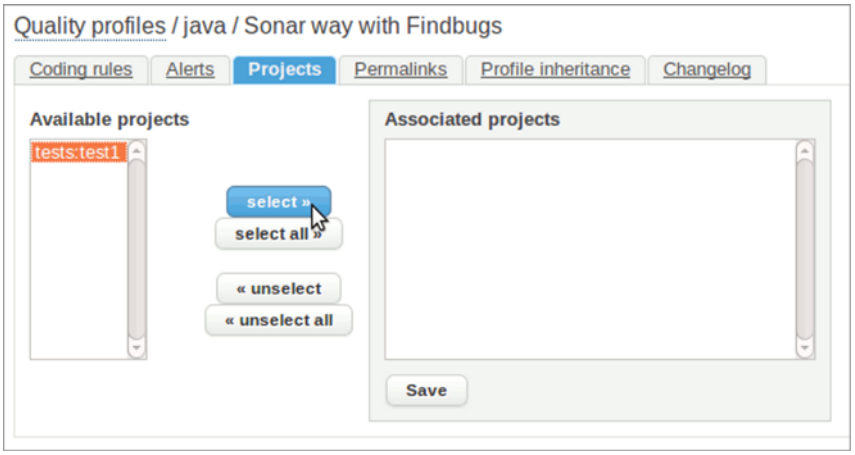


Figure 2.13 **Projects can be assigned to a non-default profile for analysis.**

2.4 Related plugins

A number of plugins are available that add rules, but we're going to focus here on something different: the SCM Activity plugin, which enhances the presentation of your issues. Here's a brief rundown.

2.4.1 SCM Activity

The SCM (Source Control Management) Activity plugin integrates with your source control repository to attach committer information to each line of code, as shown in figure 2.14. This free plugin works with a number of mainstream SCM tools, such as CVS, SVN, and Git, to show you not only who last checked in changes to a given line of code (not necessarily the person who added an issue), but also how old that change is.

Figure 2.14 **The SCM Activity plugin adds committer and commit-date information to the file detail view. This screenshot shows a section of code, starting at line 54 of the file, that was checked in on 7/26/2012.**

Once you install the SCM Activity plugin and restart SonarQube, you'll need to make configuration changes on a project-by-project basis to begin seeing commit data. The configuration page for the plugin does a good job of walking you through what to configure and how to do so.

There are just a few things to keep in mind. First, after installation, this plugin is on by default. If you're in a Maven shop, with your SCM server info embedded in your pom.xml, then this is peachy. For everyone else, this can be a pain, because if the plugin is enabled for your project (the default) but your SCM server info isn't available (also the default), your analysis will fail. You'll either want to switch the global configuration for this to be off by default, or be aware that your analyses will fail until you either toggle this off at the project level or tell SonarQube where to find its source control server.

Second, there are plugin configuration inputs for the login and password to connect to your SCM. You can leave these blank if your SCM URL encapsulates those credentials, as it does in a typical CVS URL, for example.

Once you've finished application configuration and saved your changes, kick off another analysis. This first one with the SCM Activity plugin on board can take significantly longer than you're used to. Essentially, the plugin is reading in the entire history of your project. Each analysis after the first one will take longer than analyses did before you added SCM Activity, but not nearly as long as this setup analysis.

2.5 Summary

SonarQube scans your program code for anti-patterns and reports each instance as an issue. Not all issues are bugs, but every issue needs further attention.

Rules come from existing tools like FindBugs, PMD, and Checkstyle, as well as from SonarQube itself. Rules are packaged into profiles, and for each language SonarQube can analyze, there will be at least one profile. Among the profiles for a language, one will always be marked as the default. An administrator can easily change which is the default profile. If the default profile isn't appropriate for your project, an administrator can easily assign the project to a different one.

We've focused on issues and coding standards in this chapter, but SonarQube also offers rules that check (and flag) problems with complexity, duplications, documentation, and unit testing—in other words, other Axes of Quality. As we cover those topics in the next few chapters, starting with testing in chapter 3, keep in mind that you also have the option to raise issues when the code isn't meeting your standards for those axes.

3 Ensuring that your code is doing things right

This chapter covers

- Knowing how much of your code is doing things right
- Explaining metrics on a file level
- Configuring your favorite code-coverage tool
- Integration testing (IT)

These days, unit testing is a standard practice for many teams. But if you're among the holdouts, the first part of this chapter should convince you that it's the right time to get started with unit testing.

Whether you're just getting started with unit testing or you're already an experienced test writer, SonarQube can help you track how much of your code is covered by unit tests. And more important, it will help you pinpoint the gaps in your testing, so you can close them. Two types of things can go wrong with unit tests: runtime issues (such as failing or erroring-out) and rule issues. We'll talk about both in this chapter.

SonarQube's great for everyone, not just coders. If you're on the QA side of the house, you'll find that SonarQube helps *you* do a better job, too, by showing you

what the developers have already covered with unit tests and what they've left undone. Now you can focus your efforts where they'll do the most good, on the features that are poorly covered by unit tests.

Management and the rest of the project stakeholders all have something to gain from SonarQube as well. We know you care about software quality, and test coverage is an important element of quality control. We'll show you how easy it is to track and improve your coverage with SonarQube.

We'll start from the project dashboard and look at the high-level testing metrics. Then we'll drill in to the source code and show you how to spot files with problematic tests and low coverage. When we're done, you'll never think of testing the same way again.

The topics we'll cover here don't require you to be an expert in unit and integration testing, but you may want to check out some of the following Manning titles if you'd like more information:

- *Effective Unit Testing*, by Lasse Koskela (2013), www.manning.com/koskela2
- *JUnit in Action*, 2nd edition, by Petar Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory (2010), www.manning.com/tahchiev
- *The Art of Unit Testing*, 2nd edition, by Roy Osherove (2012; 3rd edition forthcoming), www.manning.com/osherove2/

The examples shown in this chapter are based on JUnit. But we've tried to keep them as simple as possible so that TestNG fans or non-Java readers won't find the chapter hard to follow. TestNG and JUnit are the most popular unit-testing frameworks with adequate online documentation (http://junit.org, http://testng.org). You can check them for useful resources and use cases.

3.1 Knowing how much of your code is doing things right

What's the biggest reason for the eternal brawl between programmers and testers? Is it (from a coder's perspective) because testers are exaggerating whiners who magnify tiny issues in otherwise bulletproof code? Or is it because (from a tester's point of view) programmers are lazy slobs who would have tried to downplay the gash in the Titanic?

What about when customers find issues in a production environment? Someone always asks: whose fault is it that the bug made it to production? The developers who coded it? The testers who didn't catch it? You won't find an answer here—or even an argument either way. It's just not worth it. What's more important is figuring out how to keep it from happening again.

The reason bugs slip into production is that time is a limited resource for everyone on the team. You need to find a way to spend it wisely. Theoretically, developers write unit tests for every method they implement, to prove that their code works. But do they really? On the other hand, the QA team should re-test all the system functionality for each iteration or release. That's almost impossible, though, so they usually pick

some subset of features for re-testing. What are the criteria for picking those features? There probably are some, but it can seem as random as tossing darts at a feature list.

Now comes the "what if" scenario. What if you knew which source files were already covered by unit tests? Then the QA folks could focus their limited testing time on the functionality that was only partially covered by unit tests or not covered at all—making far, far better use of their time than the feature testing lottery you may have had in the past. If you agree, then you're reading the right chapter. Start SonarQube's engines, and get the testing coverage metrics you need.

3.1.1 Understanding unit-test metrics

We'll start from the default dashboard, with the widget that displays SonarQube's unit-test metrics. After your first analysis, you'll see something like figure 3.1, which shows Code Coverage on the left and Unit Test Success on the right.

Many readers will be familiar with the code-coverage numbers, but unit-test success may need some explanation. For the time being, keep in mind that test-coverage metrics report on how much of your code is tested, whereas unit-test success provides quantitative information about your unit tests.

Figure 3.1 SonarQube's testing widget on the default dashboard

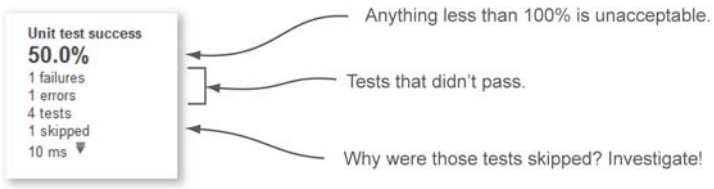


Figure 3.2
Unit-test metrics

Let's group these metrics into a category and call them *unit-test metrics* (see figure 3.2). Starting from the right side of the widget, the first and most important metric is the unit-test success density. Any number below 100% is unacceptable and should trigger a red alert. Below the unit-test success density are some of the numbers that feed the density calculation. Table 3.1 gives a detailed explanation of each metric. We decided not to list them in the order they appear in the widget, but in a way that will help you better to understand success density, which is computed based on the other metrics.

Table 3.1 Unit-test metrics

Metric	Description
Failures	Absolute number of assertions that failed. If a test includes more than one assertion (which is considered bad practice), then those found after the first failed assertion are never executed—which is one reason multiple assertions per test is bad practice.
Errors	Absolute number of tests with errors. A test with one or more errors is a test that hasn't completed its assertion. For instance, an unexpected exception has occurred during its execution.
Tests	Absolute number of tests executed by SonarQube during the latest project analysis.
Skipped	Absolute number of tests that weren't executed. For example, in JUnit 4.x, these tests are annotated with `@Ignore`; and in TestNG, they're annotated with `@Test(enabled=false)`. Similar attributes exist in most xUnit frameworks.
Ms	Time (in milliseconds) needed to execute all tests by SonarQube during the latest project analysis.
Unit-test success density	A roll-up calculation of your unit-test suite's overall success in the last run. It's calculated based on the following formula: Tests – (Failures + Errors) / Tests

When you're looking over your unit-test metrics, keep in mind that test failures and errors are as important as success density. If you see any value other than zero (0) for failures or errors, then it's time to start worrying and take immediate action. Non-zeros for either metric indicate either potential bugs in your code (worst-case scenario) or outdated unit tests (best-case scenario).

Skipped tests aren't as critical as failed tests, but any positive number means you need to do some investigation. Why are there tests that aren't executed? Are they useless? Are they failed tests disguised as skipped tests? Did someone notice they were failing (maybe due to the latest code modification) and, instead of address the problem, flag the tests to be ignored?

Unfortunately, developers sometimes skip tests instead of trying to fix them or clean them out. The tests that get skipped are usually the ones that are hard to maintain, provide no real testing, or, worst-case, are failing and nobody knows why. Unless you have serious reasons for skipping tests, you should examine each one carefully and decide what has to be done so you can stop ignoring them.

The last two metrics are the number of tests in your source code and the time needed to run them during SonarQube's analysis. Knowing how many unit tests you have in your project is of little use: 10 tests means one thing for 100 lines of code and something completely different for 10,000 lines of code. Also, how many assertions does each test include? Ideally, you have a single assertion per test, but it's possible that one single test asserts more conditions than 10 optimally written tests. Based on our experience, this metric is rarely used for quality conclusions, but it can still provide some valuable insight if combined with other metrics.

On the other hand, the time elapsed to complete all unit tests could be useful. Although you should write your unit tests in such a way that they fail if they take longer than you expect, the total time spent for unit-test execution is helpful to know in a Continuous Integration environment because it can help you recognize when it's time to refactor your tests.

Martin Fowler's definition of Continuous Integration

"Continuous Integration (CI) is a software development practice where members of a team integrate their work frequently. Usually, each person integrates at least daily—leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible" (Martin Fowler, http://martinfowler.com/articles/continuousIntegration.html).

One of the key concepts of CI is that the development team gets instant feedback (within a couple of minutes) on the build results so that they can take immediate actions to fix a broken build. If tests take too long to execute, then the feedback is delayed as well, and the team loses the immediate feedback advantage of the CI practice.

Continuous inspection using SonarQube (a step beyond CI) is covered in depth in chapter 9. You'll learn how to integrate your favorite CI tool with SonarQube and automate your quality-inspection process.

For the topic of this chapter, keep in mind that one of the key points of a successful CI environment is to send feedback to every team member as soon as possible after each commit. If unit tests, which are executed by every automated build, take too long to complete, then feedback is delayed, and your CI process is less effective. So if SonarQube reports a high value for unit-test execution time, you may need to optimize your tests.

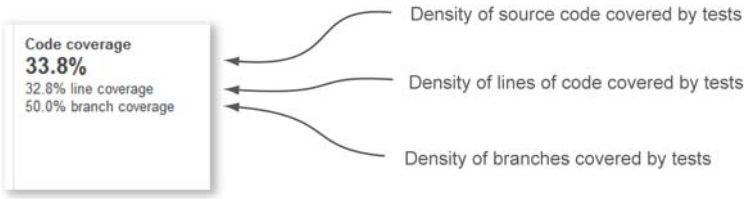


Figure 3.3 Unit-test coverage metrics

3.1.2 Getting reports on unit-test coverage metrics

Let's go back now to the dashboard widget and look at the code-coverage numbers on the left. As figure 3.3 shows, there are three different coverage metrics: code coverage, line coverage, and branch coverage.

We prefer to group the coverage metrics in a category called *unit-test coverage metrics*. All of them are based on testing execution and provide an overview of how well your source code is covered by unit tests. Once again, remember that numbers are everything, and at the same time, they don't mean anything if you try to interpret them in isolation. Table 3.2 explains how these metrics are calculated and what they mean.

Table 3.2 Unit-test coverage metrics explanations

Metric	Description
Line coverage	The density of lines that are executed at least once during unit-test execution. It's calculated by the following formula: *Lines to Cover – Lines with No Unit Tests / Lines to Cover* where *Lines to Cover* is considered to be all lines that need to be covered by unit testing.
Branch coverage	The density of possible paths (branches) in flow-control structures that have been covered by unit tests during SonarQube analysis. It's calculated by the following formula: *((2 * Number of Paths) – Covered Paths by Unit Testing) / (2 * Number of Paths)*
Code coverage	A combination of line and branch coverage. It helps assign a single value for test coverage. It's calculated by the following formula: *((2 * Number of Paths) – Covered Paths) + Lines of Code – Covered Lines / (2 * Number of Paths) + Lines of Code*

A *branch* is a source code block that is conditionally executed at runtime—that is, only when a condition at the branching point is satisfied. In our experience, branch coverage is always the metric with the lowest value. This means most projects are at risk, because only a fraction of the logical paths through the code are tested. Imagine that during an online checkout process the customer has to select a payment method (credit card, PayPal, or bank transfer) and then enter the account details (card number or bank account, and so on). For simplicity, we'll say that the selected payment method and its details are handled in a single source code file. So at runtime, there

are three different paths to follow (one for each payment method), depending on user interaction.

If there are only unit tests for credit-card processing, then only a third of the actual code is covered, and the branch coverage value for the file is 33.33%. Unfortunately, developers tend to write unit tests only for the branch that's most likely to be executed at runtime: the *happy path*. We believe this happens for two reasons. First, they usually don't understand branching from a unit-testing perspective. Second, even if they do, they feel that writing tests for all possible paths is a waste of time. But in our experience, leaving branches uncovered by unit tests means a lot of bugs not caught. Perhaps it won't be an issue immediately, but certainly it can be down the road as the code evolves. For clarity, here's an example of two branches:

```
public boolean isAdult( int age ){
  if (age < 18){
    System.out.println ( "You are still too young for that.");
    return false;
  }
  else{
    System.out.println ( "Well done. Are you ready to change the world?");
    return true;
  }
}
```

Branch 1 is executed only if age is less than 18

Branch 2 is executed only if age is 18 or over

Each branch displays a different message based on the result of the branch point (`age < 18`). If you write only one unit test that checks this code, let's say by passing in a value of 10 for the `age` parameter, then you've covered only branch 1. Branch 2 is still untested, and you can't be 100% sure that your code is working as expected. The following listing shows a JUnit test class that gives you 100% coverage for the `isAdult` method.

Listing 3.1 JUnit Test class that gives 100% coverage on the `isAdult` method

```
public class AgeValidationTest {

  @Test
  public void validateAdult() {
    int age = 18;
    Employee instance = new Employee();
    assertTrue(instance.isAdult(age));
  }
  @Test
  public void validateNotAdult() {
    int age = 17;
    Employee instance = new Employee();
    assertFalse(instance.isAdult(age));
  }
}
```

There's something else important here that SonarQube can't tell you. You could use the extreme ages of 0 and 500 in your test cases and get 100% branch or line coverage and still miss a defect if the code checks for 180 instead of 18. That's why you write two

tests that check the maximum age (17) at which people are still considered non-adults and the first age (18) at which people are considered adults.

Remember that you should carefully select the inputs of your unit tests. As we said in the chapter introduction, a couple of great books explain in detail all unit test best practices.

Let's go back for a moment to why it's not a good idea to focus on only one metric for test coverage. A couple of years ago, we analyzed an open source project with SonarQube and noticed that whereas line coverage was over 90%, branch coverage was below 30%. If that were your project (or one you were using), would you think there was enough coverage? We didn't; 30% is too low for branch coverage.

Let's take a closer look at those numbers. It looks like unit tests execute 9 out of every 10 lines of code at least once. And that's great. On the other hand, only a third of the paths in flow-control structures are exercised by unit tests. The conclusion? This code base needs more unit tests for these branches.

You might be wondering by now which number is acceptable for test coverage (line, branch, and code). Well, there is no "right" answer to this question. Ideally you'd expect to achieve 100% for all three metrics. But this isn't feasible, especially in large projects. Moreover, 100% coverage of a class doesn't verify that the code is doing what's expected. For instance, if you have 70% coverage, the only safe information is that the other 30% of your code isn't executed by unit tests. The "covered" 70% requires manual verification to ensure that everything is really and properly tested.

Take a look at the next listing. It again gives you 100% coverage for the `isAdult` method, but there is no real testing.

Listing 3.2 JUnit test class with no real testing

```
public class AgeValidationTestDummy {

  @Test
  public void validateIsAdult() {
    int age = 18;
    Employee instance = new Employee();
    instance.isAdult(age);
    age = 17;
    instance.isAdult(age);
  }
}
```

In the listing, both method branches are covered at least once during test execution. But do you see any assertions of the returned values? Although you have a 100% value on code coverage, you haven't tested the method's expected behavior. Furthermore, you can configure SonarQube to notify you about a test without an assertion. To do so, you need to enable the appropriate rules under the repository PMD Unit tests. You'll find more on administering rules and quality profiles in chapter 13.

We advise you to use test metrics solely to identify poorly tested code blocks. If you get high values on code coverage, feel free to celebrate, but only if you're sure that all developers correctly apply the unit-testing practice.

Let's go back to the numbers. If you *had* to pick just one testing metric to track over time, we'd recommend code coverage, the first number in SonarQube's widget. But all the coverage numbers deserve attention. With a little scrutiny, they reveal valuable information, such as the fact that you have untested paths or critical program features with low coverage.

Now that you've had a taste of testing metrics, it's time for real action. The discussion in the next few sections will be about technical details at a low level; so if you're a manager (technical or not), you might be tempted to skip ahead to section 3.4, which covers integration testing. But even if you're not a hands-on coder or tester, we think an understanding of the technical details can be helpful. So take a deep breath and dive in to your source code again. This time it's all about testing.

3.2 Explaining metrics on a file level

Remember that dashboard widgets provide information on a project level, but tracking unit-testing metrics and coverage is nearly meaningless if you don't drill down to each source file and get reports for individual lines of code. After all, if you're a developer, you're responsible for testing your code on the unit level to ensure that it's working as expected. And if you're a tester, you need to know which parts of the code (which program features) are poorly unit-tested (have low code coverage) so you can spend more time testing them at a higher level.

Next, we'll look at how to spot which lines of code in a file aren't covered and which branches aren't followed during test executions. When we're done, you should be able to get reports at a file level on the values of the three density metrics described in table 3.2 (line coverage, branch coverage, and code coverage). You'll also learn how to read the Coverage tab in the file detail view, and the Tests tab, which appears in the file detail view for the unit tests themselves and offers details on the tests executed for a single file. More important, when we're finished, you'll know how to find where to start fixing or testing what's not already covered.

3.2.1 Hunting source code lines with low coverage

Starting again from SonarQube's default dashboard, click-through on the first code-coverage metric, and you'll be redirected to the code-coverage drilldown view. You'd get basically the same screen if you clicked Line Coverage or Branch Coverage; we'll cover the minor differences in a minute.

At the top of the page, you see the drilldown view's normal presentation of modules, packages, and files (if there is only one module in the project, then the first pane is omitted). Click a module to filter the package and file sections to only those in the selected module. The same applies when you click a package name. Figure 3.4 is a screenshot taken from SonarQube's public instance, http://nemo.sonarsource.org. It shows coverage of SonarQube's own modules; because we clicked the CPD plugin module, lists in the second and third panes are filtered accordingly.

Next to each module, package, and file is its corresponding code-coverage metric. What's cool is that the drilldown view is always sorted to show the worst first, so com-

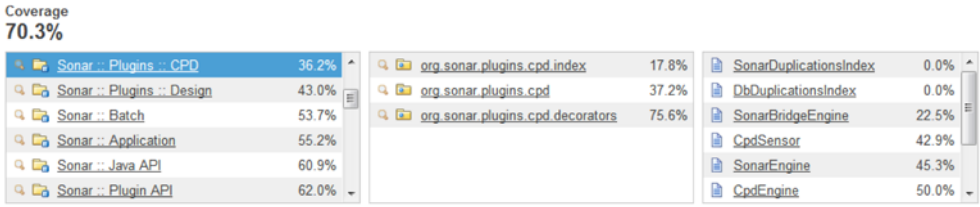


Figure 3.4 Drilldown view of code coverage

ponents with low coverage are displayed at the top. When you're looking at your own code coverage, keep in mind that components with perfect scores are omitted from the drilldown (after all, the point is to help you find what you need to work on). In other words, if a file or package has a code-coverage score of 100%, then it's not displayed in these lists. If you don't see some of your classes or packages, don't worry. It's a feature, not a bug, and it's telling you that you've done your job right!

Before we move to the file's Coverage tab, let's see how the drilldown view looks when you click Line Coverage or Branch Coverage from the dashboard. There are two major differences between figure 3.5 and the list on the left in figure 3.4.

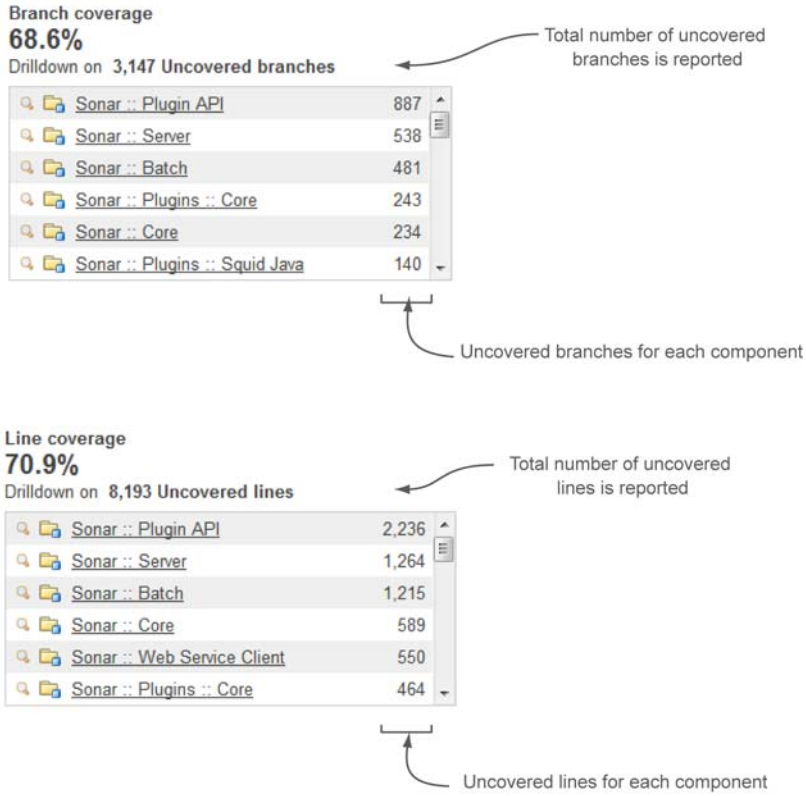


Figure 3.5 Drilldown view of line and branch coverage

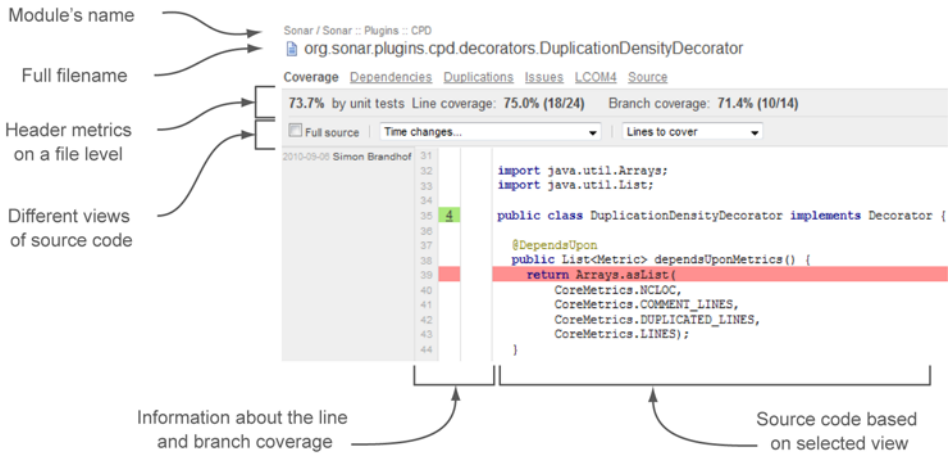


Figure 3.6 **The Coverage tab of a source code file**

As you'll probably notice, additional header information reports the absolute number of uncovered lines/branches on a project level. By the term *uncovered lines/branches*, we mean those source code lines or branches that aren't tested at all. The other big difference in the drilldown is that the numbers to the right of each component show how many lines/branches need testing, rather than the density of coverage. Bigger numbers are sorted upward, but the organization is still worst first, so less-tested components are again shown at the top of the list.

By now you can tell which modules, packages, and files are poorly covered by tests just by browsing to the drilldown view. We're sure the QA folks are already jumping up and down at the prospect of knowing which parts of the system aren't adequately unit-tested. Developers, on the other hand, have both a low-level and a high-level insight specifically into testing. What is missing is the file-level information that will allow you to locate the individual code lines with little or no unit-testing coverage.

Click a filename, and a brand-new world appears. It's the Coverage tab for your file, and as figure 3.6 shows, it contains everything you need to start improving your unit tests.

At the top is the fully qualified filename (including package) and the module it belongs to, if any. In the tab's header, you see all the code-coverage metrics we discussed in section 3.1: line coverage, branch coverage, and code coverage, this time on a file level.

Just below the header is the source code of the file you're looking at. By default, SonarQube hides useless lines such as package declarations, library imports, and header information. You can limit the number of lines shown by choosing a different view from the corresponding list (see figure 3.7), or you can browse the full source code by clicking the check box provided for that purpose. For now, ignore the first drop-down list box. We'll cover it in depth in chapter 9, which discusses continuous inspection in detail.

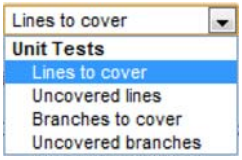


Figure 3.7 **Coverage tab view selection**

Table 3.3 Unit-test metric explanations

Selection	Explanation
Lines to Cover	The default selection. Displays all lines of code that need to be covered. Lines that don't need test coverage (header comments, the package declaration, and library imports) aren't displayed. Includes both uncovered and already-covered lines.
Uncovered Lines	Displays lines of code that aren't covered by unit tests.
Branches to Cover	Displays all possible branches that need to be covered, including the ones that are already covered.
Uncovered Branches	Displays uncovered or partially covered branches.

As figure 3.7 shows, SonarQube provides four options for identifying uncovered source code. Table 3.3 lists these options and discusses how they affect the file view.

READING THE SOURCE CODE VIEWER

Before you start playing with these options, you need to know one more thing: how to read the source code viewer. To the left of the code are three columns with numbers in them (ignore the leftmost column for now; we'll talk about it in chapter 10 when we cover SonarQube's review functionality). The first column with a number shows the line number within the file, the second reports on how many times the line was accessed during unit-test execution, and the third reports on the number of covered branches out of the total count of branches. As you can see in figure 3.8, because the second column is about branches, it's empty for lines that don't include flow-control structures.

Figure 3.8 Source code viewer with coverage indicators

When the first column is green and the line doesn't contain any branch points, you can check it off as "done" because it means the line is fully covered by tests. On the other hand, if you see a zero on a red background in this column, it means the line wasn't exercised at all during unit testing and you definitely need to write one or more tests for it. The third column is a bit more sophisticated. In addition to red and green, you'll also see yellow, which lets you know that some of the possible paths through the code aren't covered. When there's red or yellow in either of the last two columns, that color extends across the code so that even with a quick skim, your eye is pulled to the problem areas.

If you're feeling lost with all these numbers, don't worry. All you really need to do is pay attention to the lines where you see red or yellow. If a line is colored, it's not a good thing, and you should consider taking action to address it.

At this point, you understand how SonarQube "grades" your unit-test coverage, and you've seen how test-coverage metrics are presented at the project, module, package, and file levels. Next we'll look at the unit tests themselves, because SonarQube has something to say about them, too.

3.2.2 Finding problems in your unit tests

Until now, we've concentrated on the metrics for your program's source code. But SonarQube offers insight into your unit tests as well, which, although not usually included under the source code umbrella, are extremely important. This feedback ensures that a software system does what it's expected to do on a unit level. In this section, you'll learn how to identify problem areas in your unit-test files.

First, you'll have to go back to SonarQube's default dashboard. Until now we've looked at the left side of the code-coverage widget. Now it's time to see what's hiding behind the Unit Test Success metrics on the right. Start by clicking any number under Unit Test Success (shown in figure 3.9), and you'll land at the drilldown view.

We need to mention a couple of things here. Once you click-through, the packages and files you see aren't source code components, but testing components. For Maven folks, all files under src/test are displayed here. If you're an Ant-oriented software house, then you'll see all files placed in your project's test directory here.

If you click from the dashboard on a metric that has the best possible value (for instance, 100% success, or 0 failures), then the drilldown view will be empty because

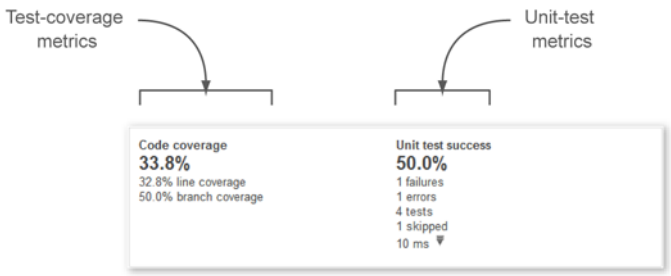


Figure 3.9 SonarQube's testing widget on the default dashboard

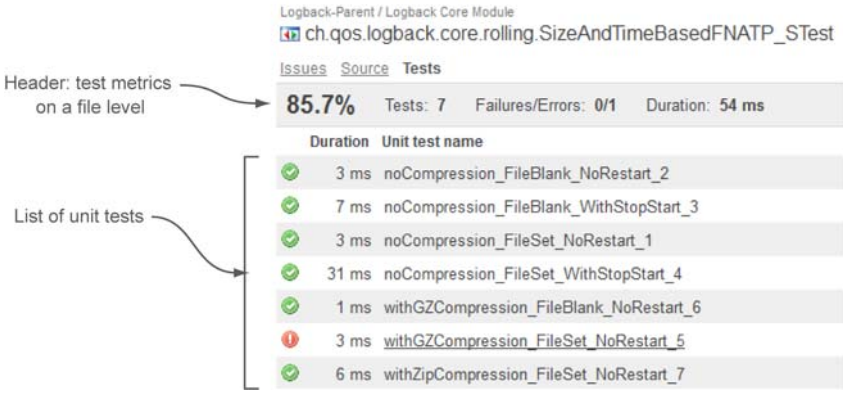


Figure 3.10 List of test methods in a test file

there is nothing you need to improve. If so, keep up the good work! And keep in mind that the beauty of the drilldown view is that it displays only components that need your attention.

We hope you don't have any failing or skipped tests in your projects, and the drilldown view is empty for every metric except test count and execution time (ms). If you'd like to see what skipped or failing tests look like in SonarQube, you can analyze the source code of this book, which was intentionally written to demonstrate all sorts of possible issues.

To start exploring the quality of your unit tests, click-through from the dashboard on the number of tests (you should have some), and navigate to any file you want in the drilldown. You'll find yourself on the Tests tab, which looks something like figure 3.10.

The unit-test presentation may remind you of the Coverage tab we discussed in section 3.2.1. The truth is, they have a lot in common. First, a similar header shows all the unit-test metrics for the current test file. Just below the header is a list of unit-test names, with the most recent execution duration and status of each. For those familiar with JUnit 4.x, this list shows all the methods in the test class that are annotated with `@Test`.

As you've probably guessed, the icon to the left of the duration represents the status of each unit test. It will match one of the images shown in table 3.4.

Table 3.4 Unit-test statuses

Status Image	Description
	Successful unit test
	Skipped unit test
	Errored unit test
	Failed unit test

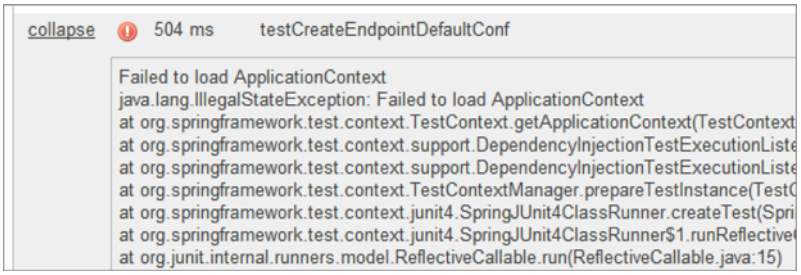


Figure 3.11 The Expand link reveals the root cause of a test error.

For tests in the list that failed or errored-out, you'll see a link in the far-left Expand column. Click this, and you'll get details about what went wrong. If the test failed, then you'll see the assertion that caused the failure. If there was an error, then you'll probably see an exception and the full stack trace, as shown in figure 3.11.

One last thing about this list: execution time is always displayed, no matter what the unit-test status is. For skipped tests or tests with errors—in other words, tests that didn't complete—the value of this metric is always 0 ms. For completed tests (successful or failed), you'll see the actual execution time of the test and its assertion(s).

At this point, you've seen all of SonarQube's core unit-test and coverage features. Without any extra configuration, it takes only minutes to obtain detailed reports and metrics on the testing health of your source code. The rest of this chapter focuses on more advanced topics, such as selecting and configuring your favorite code-coverage tool and displaying integration test metrics. At the end of the chapter, we'll present a couple of testing-related plugins that we think you might find useful.

Unit tests, rules, and issues

There's one last thing to know about discovering issues in unit tests. We told you earlier in this chapter that you can use rules and experience issues in unit-test files just as you do with your source code files. In fact, these rules support only JUnit tests. But if you're an experienced user, you can create your own rules by modifying existing ones to cover other frameworks.

In the previous chapter, we discussed issues and where they come from. In chapter 13, we'll cover rule-set administration in detail. Meanwhile, keep this in mind regarding unit-test rules: by default, all rules aren't enabled, and they fall under the rules repository named PMD Unit Tests. Activate those that make sense to you, and launch a new project analysis. Issues (if any) will appear in the Source tab. You won't have any trouble spotting the issues, because they look like those found in source code files.

3.3 Configuring your favorite code-coverage tool

SonarQube's default code-coverage tool for Java projects is JaCoCo; but Cobertura is also embedded (which means the two most popular Java coverage tools are available by default), and there are plugins to support EMMA and Clover as well. For other languages, coverage tools are provided in the corresponding plugin.

Table 3.5 summarizes the coverage tools supported by SonarQube. For more details about installing and updating SonarQube plugins, please refer to chapter 14. To check for the most up-to-date version of the tools, consult the update center in your installation.

Table 3.5 Code-coverage tools supported by SonarQube

JaCoCo	Embedded/Default	www.eclemma.org/jacoco
Cobertura	Embedded	http://cobertura.sourceforge.net
Clover (Commercial)	Plugin	www.atlassian.com/software/clover/overview
EMMA	Plugin	http://emma.sourceforge.net

As we've told you, SonarQube isn't only for Java gurus. It supports more than 20 languages that include their own code-coverage mechanism or integrate with popular external tools.

For instance, when analyzing a JavaScript project, the JSTest unit-testing framework is used by default to provide coverage reports. The PHP plugin uses PHPUnit, the Python plugin uses the Coverage.py toolkit, and so on. For a complete (and updated) list of the coverage tools included in a language's plugin, we advise you to visit SonarQube's online documentation.

In the following subsections, you'll learn how to choose your favorite coverage tool (sorry, non-Java folks, but this feature isn't yet available for other languages) and how you can adjust its settings to fit your needs.

3.3.1 Changing the default selection

You can change the default code-coverage tool on a project or global basis, but we strongly advise you to keep the default selection. If, and only if, you have very good reasons to change it, then do it for all your projects. There are some differences that aren't terribly important among code-coverage tools regarding how coverage is calculated. See "Code Coverage Tools (JaCoCo, Cobertura, Emma) Comparison in SonarQube"[1] for a comparison of the supported code-coverage engines supported by SonarQube. Using different tools for different projects would not only be disorienting as you browsed from project to project, but would also make comparisons of project health across your portfolio difficult, if not meaningless.

[1] Papapetrou, Patroklos, "Code Coverage Tools (JaCoCo, Cobertura, Emma) Comparison in Sonar," *Only software matters,* December 19, 2012, http://mng.bz/hjgg.

To make a global change, navigate to the global configuration page and select the Java category from the General Settings options. Then set the `sonar.java.coveragePlugin` property to one of the accepted inputs: `jacoco` (default), `cobertura`, `clover`, or `emma`. Note that although the tool names themselves are proper, the recognized values for this setting are in all lowercase. Once you've entered your choice, click Save Java Settings, and run a new analysis for each project. That's it! You've made the switch, and the coverage metrics you now see are coming from your new tool of choice.

Code-coverage tools for other languages

The code-coverage tool property is under the Java category settings. That means changing its value affects only Java projects. Other languages supported by SonarQube may provide similar configuration for code-coverage engines, so you're advised to look at the corresponding category under the global or project settings.

The same attribute is also available under project configuration settings, so you won't find it hard to change the code-coverage tool for a single project—even though it's not a good idea to do so.

Each supported code-coverage tool, whether it's embedded in SonarQube or not, provides some advanced settings. The majority of these properties are available only at the project level, but some can be applied globally, and a few can be set on a per-analysis basis (see appendix B). For a complete list of available settings and their meanings, you can browse SonarQube's online documentation about each code-coverage plugin.

3.4 Integration testing

Integration testing (IT) is today what unit testing was a decade ago. Unit testing is not enough for complex systems, and no software is considered well-tested if there aren't adequate integration tests to cover program features.

IT focuses on testing source code not in isolation (like unit testing), but in a production-like environment with all the external resources that implies. For instance, a JavaEE application runs on an application server, stores and retrieves data in a database, and takes advantage of services provided by frameworks such as EJB and JSF. Unit testing must fake these interactions by using mocking tools such as Mockito, jMock, EasyMock, and others. But IT validates the system directly against all these resources.

Originally this was a manual process, typically involving (for a web app) a web server, a browser, and a human, with the same process over and over: compile the code, build the artifact, and deploy the project. Then access it with a browser, and start manually testing each feature and interaction. At the beginning of a project, this process was quick and easy because the functionality was still limited.

After weeks of coding and adding new features, the process would become more and more time-consuming—and boring. How do you stay engaged when you have to check and recheck the same functionality over and over again, especially when it

hasn't changed? After a few weeks or months, this practice can drop in importance from "necessary" to "necessary evil," and it's often abandoned sooner or later.

Fortunately, this has changed in the last couple of years, and integration testing—*automated* integration testing—is considered as valuable today as unit testing. No matter what programming language is used, plenty of tools allow the automation of integration tests. Once you've got your tests set up, SonarQube lets you track integration test metrics separately from unit tests, so you can retain clarity on each.

3.4.1 Displaying integration testing coverage on the dashboard

SonarQube provides a separate widget for integration-test metrics. It's not on the dashboard by default, but it's easy to add; you'll find it in the dashboard configuration in the Tests group. We'll look at it in a minute, but before we do, we need to point out some things:

- At the time this book was printed, integration-test metrics were supported in Java, C#, and Python projects. It's likely that this has changed, so you're advised to browse SonarQube's online documentation for the most updated information (http://docs.codehaus.org/x/opS7DQ). Both authors have a Java background, so the material taught in this section is Java-centric; but similar concepts apply to other languages, too.
- You should use JaCoCo to create the IT report. It can be used to calculate IT metrics no matter which tool or application was used to run these tests (Maven Surefire Plugin, Maven Failsafe Plugin, Ant scripts, Selenium, Arquillian, HtmlUnit, and so on).
- Instruct SonarQube to reuse JaCoCo reports by setting the property `sonar.jacoco.itReportpath` to the report file produced by JaCoCo.
- You can still use your favorite coverage engine to compute unit-testing metrics.

Figure 3.12 shows the integration test coverage widget, which is similar to the one for unit-test metrics. But unlike the unit-test widget, the integration-test widget doesn't give any information about the tests themselves; it only shows how well they cover your code. Furthermore, the Overall coverage section describes the combined coverage of both unit and integration tests. We find this information useful, because there are cases where unit tests are meaningless but we still want to aggregate both metrics in one result.

The metrics for IT coverage are the same ones we looked at for unit tests in section 3.1.2. Getting reports on unit-test coverage metrics. We gave you the formulas then, but it's been a while, so here's a brief reminder:

- Line coverage % is the density of lines that are executed at least once during integration-test execution.
- Branch coverage % is the density of branches that are covered at least once during integration-test execution.
- Test coverage % is a combination of line and branch coverage.

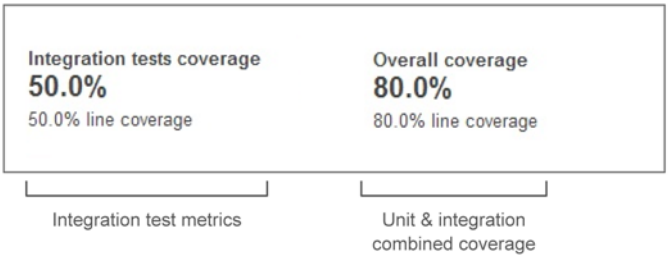


Figure 3.12 Integration-test widget

At the file level, the integration-test data is … well, integrated with the unit-test coverage information, so getting to it is the same as what we looked at earlier. Click-through on a test-coverage metric, and choose a file in the drilldown. Once you're there, you'll see a familiar presentation enriched with new metrics and features.

3.4.2 Getting IT information in the source code Coverage tab

Enable reporting on integration test coverage, and after the next analysis, you'll see a couple of changes in the file detail Coverage tab. First, the header now has an additional row for integration-test metrics. They're calculated with the same formulas we looked at for unit testing, but they're named slightly differently to reflect their meanings, as shown in figure 3.13.

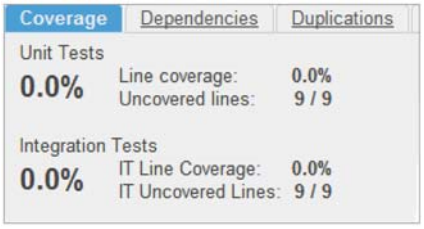


Figure 3.13 Source code Coverage tab with IT metrics

Now look at the dropdown that lets you pick your source code view. Again, you see changes, as shown in figure 3.14 and explained in table 3.6.

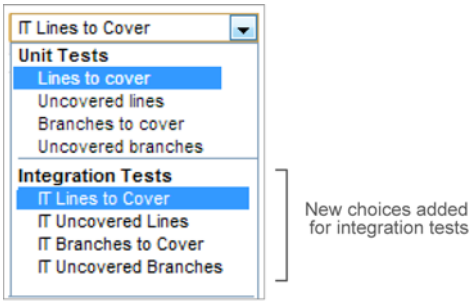


Figure 3.14 Coverage tab view selection—enriched with IT choices

Table 3.6 Integration-test metrics

Selection	Description
IT Lines to Cover	Displays all lines of code that need to be covered by integration tests. Header information, package declaration, and library imports aren't displayed. This number includes both uncovered lines and lines already covered by integration tests.
IT Uncovered Lines	Displays lines of code that aren't covered by integration tests.
IT Branches to Cover	Displays all possible branches that need to be covered by integration tests. This number includes both uncovered branches and already-covered branches.
IT Uncovered Branches	Displays branches that are uncovered or only partially covered by integration tests.

3.5 Related plugins

We've mentioned the EMMA and Clover plugins, which offer integration with those two coverage engines. But a complete testing strategy should probably include more than just unit testing, or even unit and integration testing combined. Ideally, it would also include functional and/or acceptance testing.

In this section, we'll discuss a testing-related plugin that adds functional/acceptance testing metrics by integrating SonarQube with Thucydides' open source library. We think it's worth your attention, and we suggest you give the plugin a try because it's particularly useful if you already use Thucydides or plan to do so.

Inactive plugins

There are a couple of plugins we decided not to cover in this book (JMeter and GreenPepper) because they've been inactive for more than two years. If you're interested in them, visit SonarQube's online documentation to find installation and configuration instructions.

Thucydides is one of the most promising acceptance-testing frameworks. With Thucydides, you can model your requirements and define acceptance criteria in a simple way by using Java and JUnit or a business-driven development (BDD) tool such as easyb (http://code.google.com/p/easyb).

BDD

Business-driven development (BDD) is a software development practice that focuses on modeling business needs and user requirements so that all team members (developers, analysts, customers, and so on) acquire the same knowledge about the system under development. The modeled specifications are transformed into an IT solution. The most important thing about BDD is that requirements are reevaluated constantly to improve the business process during the development process.

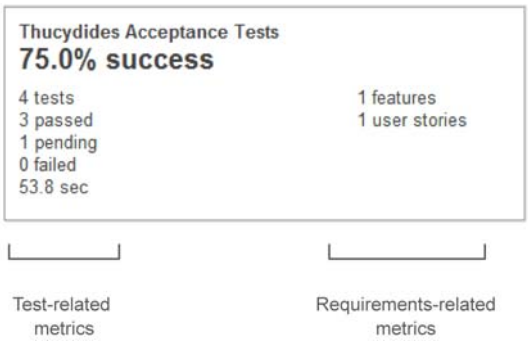


Figure 3.15 Thucydides plugin widget

The Thucydides plugin feeds SonarQube with metrics gathered during acceptance-test execution and presents them in a relevant widget. Before installing the plugin, we suggest you look at the online documentation (www.thucydides.info/documentation) or visit the official web site (www.wakaleo.com/resources/thucydides), where you can also find other interesting resources.

To add Thucydides metrics to your dashboard, click the Tests category filter and add the widget named Thucydides Acceptance Tests. Then run a new analysis of your project that contains Thucydides tests. What you'll get is shown in figure 3.15.

On the left is a list of metrics reminiscent of the default SonarQube test-coverage widget. Figure 3.16 illustrates the major similarities between the two widgets. First you see a report about the density of successful tests, and then the total number of executed tests. After that, you get information about how many tests have passed, how many are pending (not executed), and the number of failures. Finally, the widget displays the total execution time.

On the right side of the widget are two numbers indicating the total features and user stories tested by Thucydides. If you feel lost regarding these terms, keep in mind that one feature is composed of one or more user stories. For instance, imagine the feature Search Customers. A user story might be Search by Surname, and another one could be named Search by Social Security Number. For every user story, you can write as many tests as you need.

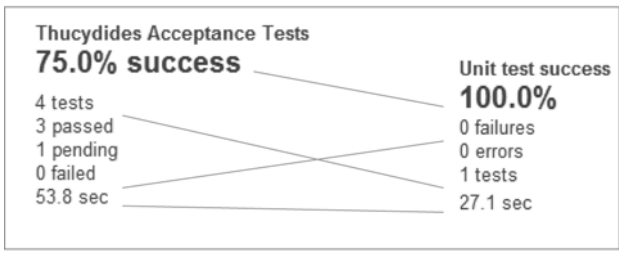


Figure 3.16 Thucydides acceptance-test metrics mapped to unit-test metrics

The current version of the plugin supports only Maven projects and doesn't execute the Thucydides tests. But it expects to find them in the predefined location of a Maven project structure, so you don't have to worry about additional configuration.

If you're in the mood to evaluate SonarQube testing plugins, be sure to try both Thucydides and its related plugin. Acceptance testing is an evolving field in the IT industry and is currently taking its first baby steps. The fact that more and more companies are in the process of including acceptance tests as part of their development lifecycle makes us believe that there is more to come on this topic.

3.6 Summary

Do you use unit testing today? Are your numbers where they should be? In this chapter we discussed the importance of unit testing, how it helps you know that your methods are doing the right things, and why it's important to have full coverage (both line and branch). We looked at SonarQube tools that help you find the chinks in your testing armor, and we delved into how to use SonarQube for integration testing.

We gave you a detailed overview of the features in SonarQube, including these important points:

- Testing metrics are organized in two categories: code coverage and tests.
- Click the value of any code-coverage metric on the dashboard, and you can get down to the file level to quickly see which lines aren't covered by tests or are only partially covered.
- Click the value of any test metric, and you can obtain a list of tests in a given file with status indicators for the last run of each.
- Integration-testing code coverage is easier than ever thanks to SonarQube's embedded JaCoCo plugin.
- Open source plugins provide integration with third-party tools and calculate metrics for load and performance testing, and acceptance and functional testing.

We hope we've so thoroughly inspired you that you're ready to start analyzing every one of your company's projects and spread the word to your co-workers and your manager. At this point, you're beginning to feel the power of SonarQube, but you have much more to learn. Get ready for more metrics and more quality. The code-duplication quality axis comes next, and it reveals the darkest aspects of your code base.

4 Working with duplicate code

This chapter covers

- The hidden cost of duplicate code
- Identifying duplications
- Realizing the impact of code duplications
- Finding duplications across multiple projects
- Cleaning up your duplications

When you start a new project, you have a clean code base with no duplications, unless you copied another project to start with. Every single line of code is written from scratch, and you have absolute confidence that neither you nor your teammates will introduce any duplications.

As the project progresses, you and your teammates communicate regularly and write good, clean code. You probably think you've done everything you need to, to avoid duplications. But over time they still creep in, because you can't totally avoid code duplications no matter how hard you try.

Because duplications always creep in, you need an efficient way to track and eventually remove them. You've already seen how SonarQube helps you measure test coverage and identify issues. Following a similar path, we'll show you how easy SonarQube makes it to spot repeated code blocks in a project and across multiple projects. We'll start from the comments and duplications widget on the dashboard, and progress to the Duplications drilldown and the Duplications tab in the file detail view.

Unfortunately, code duplications exist in every project. Even if this is the least controversial deadly sin, everybody agrees that duplication is bad. The more complex a software system, the more code blocks are likely to be repeated. This chapter will show you how duplications affect your code, what metrics are available for tracking them, and how to get started eliminating those duplications by applying basic refactoring patterns.

4.1 The hidden cost of duplicate code

Friday morning. Your team is on track to roll out a major relaunch of your online coin store on Monday. Suddenly, a guy from the QA team runs in. "The new version doesn't include the 10% discount we're offering for the next three weeks. We can't release it!"

After some frantic debugging, you pin down the problem and commit the changes. Once the QA team verifies the fix, you head off to lunch, incredibly relieved that you didn't have to spend the whole weekend working on the issue.

Monday noon. Customer Service says they've gotten several emails from customers complaining that they didn't get the discount you're advertising. And because you claimed to fix the problem on Friday (and then took a long lunch), your boss is apoplectic. You dive back into the code and quickly come to an alarming conclusion: there are two different methods for calculating the discount. You fixed only half the problem on Friday.

If only you'd known what SonarQube had to say about your duplications. If only you'd had access to what's shown in figure 4.1.

SonarQube reports duplication by line, block, and file. If you've lived through this type of situation, you already know that duplicate code is one of the highest risk factors for bug propagation. But why does code get repeated again and again?

Survey the developers you work with, and ask them if copying and pasting code is acceptable. Not only will most of them say code duplications are evil, but they'll probably also say they've never copy/pasted code themselves. Now analyze your code base with SonarQube. You'll probably see that nearly every developer has duplicated code. Can you explain this paradox? Can they?

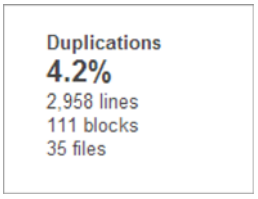


Figure 4.1 SonarQube detects duplications in files and projects and also across projects.

Table 4.1 lists some of the most common reasons for duplicated code, with the most likely ones first. Which ones apply to your organization?

Table 4.1 Common causes of code duplication

Cause	Description
Laziness	Reusing code that you know works is always tempting, even if it's written by someone else. Besides, copying and pasting several lines of code is faster and shows productivity with less effort.
Risk of regressions	Source code isn't covered by unit tests and integration tests, so to prevent any regression, this code is duplicated (and that may not be the worst option if there's no safety net).
Absence of refactoring	It's common to start developing functionality by copying an existing piece of software. You then improve it until it does what you want. Finally, you refactor to remove the duplication—but this step is often forgotten
Strict deadlines; never enough time	Some developers see copying existing, tested code as more efficient than refactoring, especially when deadlines loom. We've been there hundreds of times, and even after many years of coding, we're still tempted to copy and paste.
Poor team communication	Lack of communication can lead to duplications. It's more likely to happen on large development teams, where communication is more difficult. We saw one project where three folks wrote the same utility method, rather than asking if it already existed. Things get even worse in large organizations with multiple development teams.
Misunderstandings	Green developers have a tendency to slap in existing code that seems to do what they want, rather than working to understand the real problem or underlying business logic. Aside from leading to dirty code, this practice means they don't develop real knowledge of the problem's domain.
Merging projects	Merging projects isn't a common task, but when it happens, it's likely that the newly created/merged project will contain duplicated code. One reason is that coders tend to need similar utilities in every project, so of course they copy/paste already-proven code from previous projects. If the merged projects had similar requirements, then it's even more likely that similar code may have migrated from one to the other, producing duplications in the merged project.

Now that you've seen that code duplications exist in software systems and that there are many reasons for them, let's move on and discuss what SonarQube has to tell you about finding segments of repeated (copied and pasted) code.

4.2 Identifying duplications

We've said several times that SonarQube uses existing, best-of-breed technologies and aggregates the results for your convenience, but it doesn't do that across the board. When it makes sense, the creators of SonarQube roll up their sleeves and start from scratch to build code-quality engines that are superior to what's already available. That's the case with duplications. SonarQube's detection engine finds duplications at three different levels:

- Duplications in the same file
- Duplications in different files in a project
- Duplications across multiple projects (off by default)

Duplication detection is one of the core functionalities of the SonarQube platform, so it's available at all three levels natively, and with a uniform presentation, for every language you can analyze under SonarQube.

4.2.1 Finding your first duplication

To get your feet wet with duplication, look at figure 4.2, which shows the default project dashboard's comments and duplications widget. SonarQube provides one widget for both, with duplications on the right in the widget.

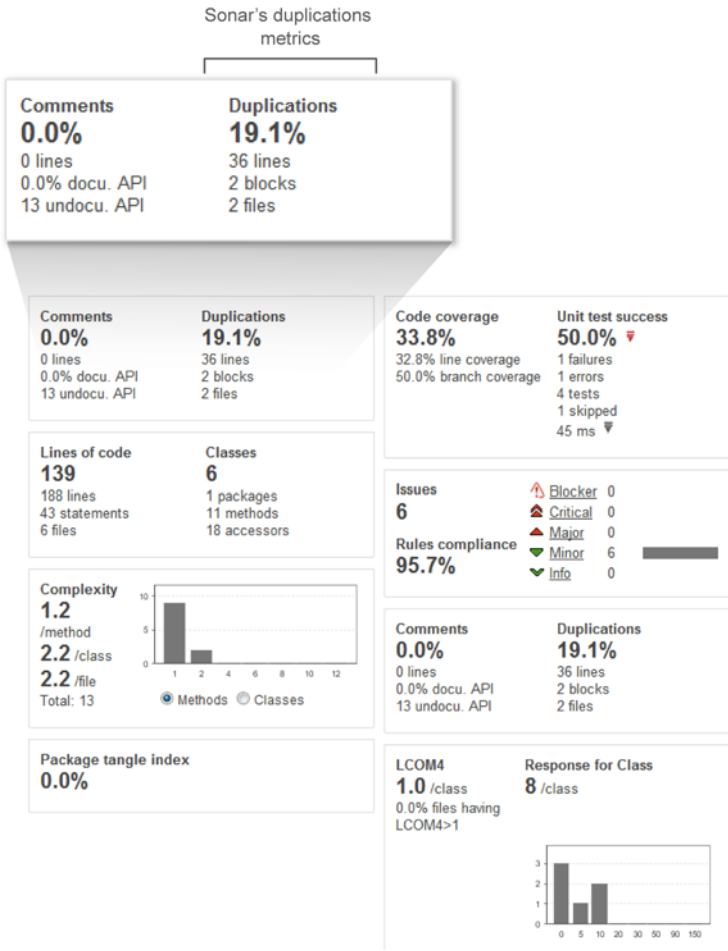


Figure 4.2 Duplications widget information

Figure 4.2 says there are 36 duplicate lines in two files. Before we proceed to the Duplications drilldown, let's take a quick look at those files, which come from the online coin store we looked at earlier. The problem in the example was duplications in different files in a project. The business requirement was to implement a default discount policy for all purchases made in the next three weeks. Unfortunately, it was only half done when the site relaunched on Monday, because of the necessary changes we made to only one of two files.

The files involved are related to the requirement that the site charge a sales tax that varies based on the customer's country. To achieve this, there are two different classes: `Order` and `InternationalOrder`. Listing 4.1 shows the relevant part of the `Order` class that is responsible for handling orders from U.S. residents.

Listing 4.1 `Order` class

```
public class Order {                                    <-- Order for U.S. residents
  private Customer customer;
  //Setters and Getters omitted
  private List<OrderLine> orderlines = new ArrayList<OrderLine>();
  //Add/remove order line code omitted

  public BigDecimal getTotal() {
    BigDecimal total = BigDecimal.valueOf(0);
    for (OrderLine orderLine : orderlines) {
      total = total.add(orderLine.getOrderLineTotal());
    }
    BigDecimal discount = total.multiply(getDiscount());
    total = total.subtract(discount);
    BigDecimal tax = total.multiply(getVat());
    total = total.add(tax);
    return total;
  }
  private BigDecimal getTax() {                         <-- Default tax for U.S. residents
    return BigDecimal.valueOf(0.05);
  }
  private BigDecimal getDiscount() {
    return BigDecimal.valueOf(0.10);
  }
}
```

1 Calculates order's total value (identical in listings 4.1 and 4.2)

2 Default discount for U.S. residents (identical in both listings)

The `InternationalOrder` class, which is excerpted in listing 4.2, handles orders placed by customers living outside the United States.

Listing 4.2 `InternationalOrder` class

```
public class InternationalOrder {          <-- Order for international customers
  private InternationalCustomer customer;
  //Setters -and Getters omitted
  private List<OrderLine> orderlines = new ArrayList<OrderLine>();
  //Add/remove order line code omitted

  public BigDecimal getTotal() {
    BigDecimal total = BigDecimal.valueOf(0);
```

1 Calculates order's total value (identical in listings 4.1 and 4.2)

```
    for (OrderLine orderLine : orderlines) {
      total = total.add(orderLine.getOrderLineTotal());
    }
    BigDecimal discount = total.multiply(getDiscount());
    total = total.subtract(discount);
    BigDecimal tax = total.multiply(getVat());
    total = total.add(tax);
    return total;
  }
  private BigDecimal getTax() {
    return (BigDecimal.valueOf(customer.getCountry().getVat()));
  }
  private BigDecimal getDiscount() {
    return BigDecimal.valueOf(0.10);
  }
}
```

1 Calculates order's total value (identical in listings 4.1 and 4.2)

Tax based on customer's country

2 Default discount for U.S. residents (identical in both listings)

You can easily spot the duplications. At a glance, you can see that the `getTotal()` method 1 is exactly the same for both classes. Moreover, the method that returns the discount percent 2 is also identical for both classes. This code may look simplistic, but we've seen real code much like it. Whether the examples are simplistic or sophisticated, the general idea remains the same: duplicate code means duplicate work down the road.

4.2.2 Finding duplications on a larger scale

In the previous code listings, you can find the duplications fairly easily. It's not hard to identify code repetitions in a few classes and a couple dozen lines. But what about in a hundred lines? A thousand? Or more? You'd need days to manually scan your source code just to find duplications in the same file, let alone across multiple files or projects.

> **NOTE** We strongly believe that any duplication density over 0% is a problem, one you need to address as soon as possible, no matter how small the number.

Clearly, manual detection is impractical at best. Instead, the best practice is to run a SonarQube analysis on a regular basis. This can be after each commit or at predefined intervals, depending on your continuous inspection strategy. Chapter 9 goes into detail on continuous inspection, but for now let's assume that you run an analysis every time the code base is modified.

4.2.3 SonarQube's duplication metrics

We've said that duplications are bad, but also that you can't totally avoid them. You're probably wondering what the duplication numbers we've shown mean and when you should start worrying about them. We'll answer those questions beginning with a brief definition of each duplication-related metric in the comments and duplications widget (see table 4.2).

Table 4.2 SonarQube duplications-related metrics

Metric name	Metric description
Duplicated Lines	Absolute number of physical lines (not just lines of code) of source code involved in at least one duplication. *Physical lines* means all carriage returns in a file.
Duplicated Blocks	Absolute number of duplicated source code blocks.
Duplicated Files	Absolute number of source files that contain duplicated lines.
Density of Duplicated Lines	Percentage shown at the top of the Duplications section, calculated by dividing Duplicated Lines by the total physical lines in the project multiplied by 100.

The first three metrics in table 4.2 are pretty clear. The only complicated one is Density of Duplicated Lines. Imagine that you have a project with a total of 1,000 physical lines of code, and the SonarQube analysis finds 1 one block of 5 lines repeated 10 times, for 50 duplicated lines. That means Density of Duplicated Lines is 50 / 1000 * 100 = 5%.

Note that all metrics are calculated not only across each project, but also on a submodule, package, and file basis. This makes it even easier to find the sections of your software with the most duplications.

4.2.4 Drilling in: from the duplications widget to the Duplications tab

To start chasing your duplications, click any metric in the comments and duplications widget, and you'll land at the common Drilldown metrics view (see figure 4.3). At upper left is the package panel: a list of the project's packages with duplications. At upper right is the file panel showing the files with duplications.

We've mentioned that SonarQube detects duplications on three levels: in a file, throughout a project, and across multiple projects of the same language. We'll talk

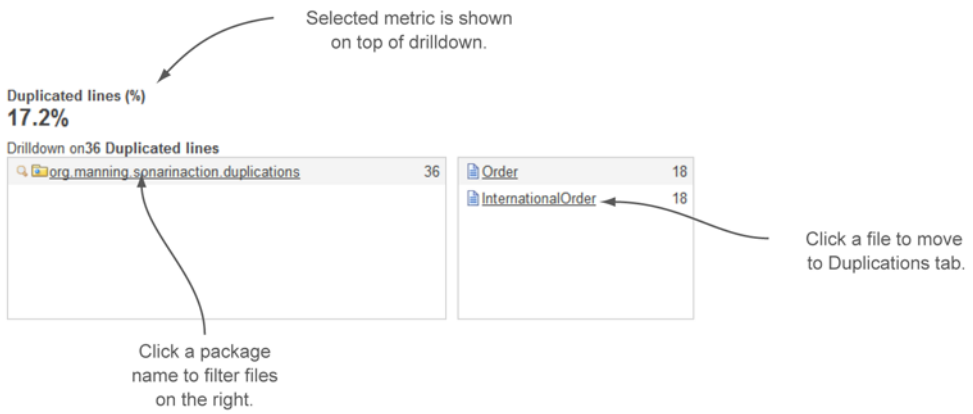


Figure 4.3 Drilldown view example: density of duplicated lines

here about the first two levels, and come back to how you can spot code copied from one project to another.

In the package and file panels, the number to the right of each item is the value of the metric you clicked in the comments and duplications widget in the dashboard for that package or file—with one exception. If you click Duplicated Lines %, then the number to the right of each item in the package and file panels is the number of lines, rather than their density in the project. Either way, descriptive messages in the drilldown view header make it clear which metrics are being displayed.

From the drilldown header in figure 4.3, you see the following:

- 17.2% of the code is flagged as duplicated.
- 36 lines participate in duplications.
- Two files (`Order` and `InternationalOrder` classes), both found under the `org.manning.sonarinaction.duplications` package, have 18 duplicate lines each.

This information is useful and important, but it's just the tip of the iceberg. Click a filename, and the file detail view for that file is added to the screen, with the Duplications tab activated. This is where you can see exactly which lines of the file are duplicated and exactly what it is that they duplicate. Figure 4.4 shows the file detail view for the `Order` class in the example project.

There's a lot going on in figure 4.4, so we'll break it into three parts, starting with the top section (the file header), which shows a metric summary for the current file/class (see figure 4.5). Note that whereas the header section of the Duplications drilldown varies based on which metric you chose on the dashboard, the Duplications tab is always the same.

The percentage displayed at left in the header is the density of duplications in the current file. Next is Lines, which is the number of physical lines in the file. Again, don't confuse physical lines and lines of code, which are computed differently.

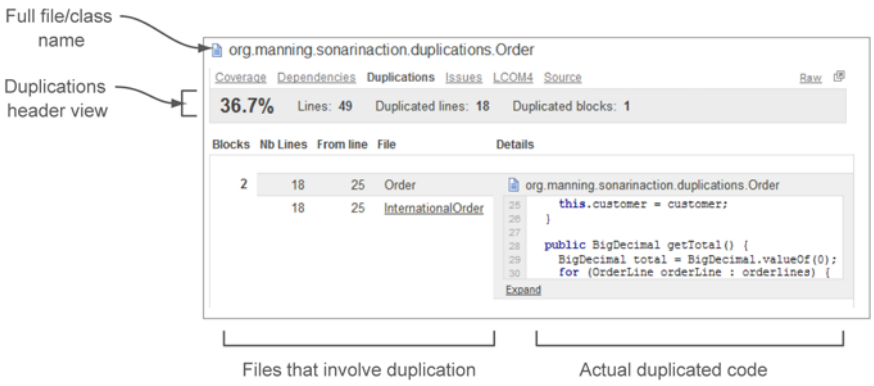


Figure 4.4 The Duplications tab view of a file/class

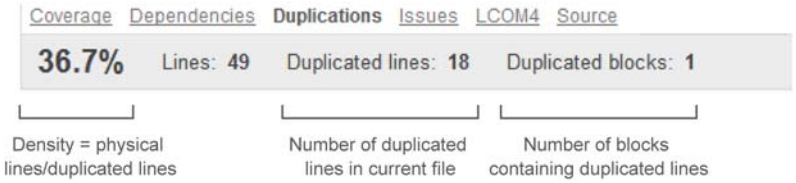


Figure 4.5 The Duplications tab's file header

> **NOTE** The lines of code (LOC) calculation varies slightly from language to language, but for Java it's as follows: LOC = physical lines – blank lines – comment lines – header file comments.

The Duplicated Lines number is the absolute number of repeated lines; and the Duplicated Blocks number, at right in the header, is the number of code blocks those duplicated lines are spread across. To sum up, the Duplications tab header includes a file-level view of three out of the four duplication metrics calculated by SonarQube, and it's easy to see why the fourth, Duplicated Files, isn't included here.

Below the header, SonarQube shows you the duplicated blocks in your file. They're presented in a two-column grid, with one row for each duplicated block. At left is high-level information about the block, including the exact position of the duplicated code in the current file/class, as shown in figure 4.6.

The left side of each row is broken into a subgrid, with one row for each instance of the duplicated block (one row for each time it was duplicated). It starts at the far left in figure 4.6 with a simple count of the number of times the code block under examination has been duplicated. You'll see a subrow in this column for each one of those blocks.

Each subrow ends with the name of a file containing a copy of this block. For first-level duplications (copies in the same class), the filenames are identical. In the example, which is a second-level duplication (from one class to another), the Nb Lines value shows that 18 lines of code are the same in the classes `Order` and `InternationalOrder`. Note that the number of lines listed from file to file isn't always the same. SonarQube's duplication-detection mechanism is sophisticated enough to pick up a duplicated block even when there are whitespace differences from one copy to the next.

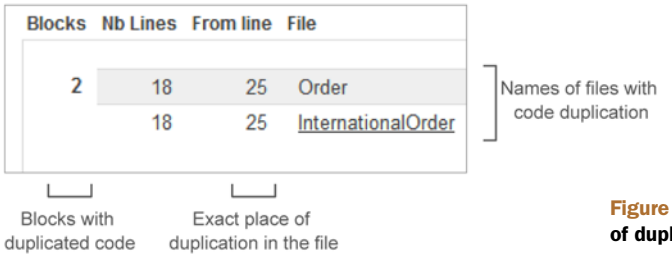


Figure 4.6 Locate the exact position of duplicated code in two classes.

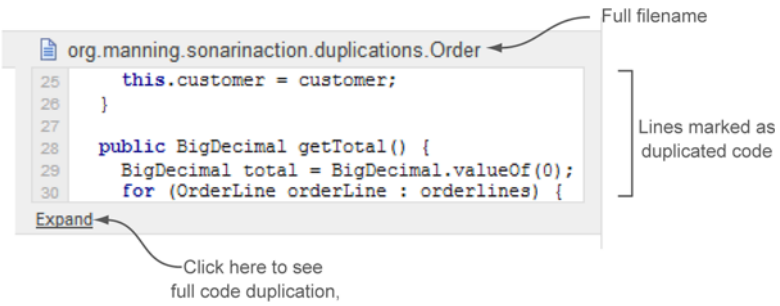


Figure 4.7 Viewing all lines containing detected code repetition

Next to NB Lines in each subrow is the From Line metric: the exact line number in each file where the duplicated block starts. Click the name of a file, and the column on the right shows a few lines of the duplicated block. By default, the code view is collapsed to hide very large blocks of code. But you can expand it to see the full duplication by clicking Expand, as shown in figure 4.7. Once a duplicated block has been expanded to its full length, the Expand link is swapped for a Collapse link, which does what you would think.

Now that you know how to find duplications, you're probably asking yourself one of two questions: either "Why do I care?" or "How do I fix them?" They're both good questions, and we'll answer them next, starting with why duplications are important and how they affect your project's overall quality.

4.3 Realizing the impact of code duplication

At this point you may be wondering, "What's wrong with having duplicate code? I have unit tests that cover all the lines and branches, and I know the code is bug-free, so why fix something that doesn't appear to be broken?" Our experience, though, has shown that most of the time, when there are lots of duplicated lines, there are few unit tests (see "Risk of regression" in table 4.1).

At first glance, that seems fair enough; but as our coin store example demonstrated, even bug-free, fully unit-tested duplicate code can cause problems. One more argument is that, as we'll discuss later in this section, the cost of maintaining software is commonly said to be directly proportional to the number of lines in the software. Next we'll delve more deeply into why duplication is a bad idea.

4.3.1 The DRY principle: minimizing and eliminating duplications

Don't Repeat Yourself (DRY) is a software engineering principle that should be applied to every aspect of a software project. It focuses on minimizing or eliminating duplications among the resources of a system, especially in code.

One of the basic concepts of DRY is that it's more efficient and productive to keep a single copy of each resource than to keep several copies. This may sound familiar to those acquainted with database normalization, but it applies beyond just your data.

That's because having multiple copies of anything, whether data or algorithms, not only means more work when there are changes, but also could mean that you end up with some outdated copies, which is the most dangerous side effect of duplication.

Think back for a minute to the coin store example, with its `Order` and `InternationalOrder` classes, each with identical `getTotal()` and `getDiscount()` methods. Forgetting to update the `InternationalOrder` class with the promotional discount had a huge impact on the system, the customers, and the business in general. How many of the customers who didn't get that discount will come back to place another order? Duplicated code is responsible for a lot of bugs, and it can have far-reaching impacts, especially in systems that are continuously evolving to reflect market needs.

4.3.2 Duplications vs. size and complexity

Having outdated copies of duplicated code is the most obvious problem caused by duplications, but there are others. Your project's size and complexity are pointlessly increased by each code repetition. You may think that's acceptable for small projects, but consider a project with 100,000 lines of code and only 20% duplications. That 20% means you have an extra 20,000 obsolete lines of code to maintain.

> **NOTE** By the terms *obsolete code* and *useless code*, we don't mean the code isn't working or used by the application; but you should refactor this part of your system in order to eliminate as many duplicated lines as is feasible. In section 4.5, we'll give you some tips on how to clean up this obsolete code.

Furthermore, code duplication is responsible for large blocks of code with only minor differences, sometimes only a couple of lines or, even worse, only a few characters apart.

The comprehension of this code is a time-consuming task that makes further modifications or enhancements extremely difficult. That means the maintainability of your software decreases dramatically. And last but not least, repeated code, especially between methods with different method signatures, can hide the real purpose of each method, making it harder to decide which one to use.

4.4 Finding duplications across multiple projects

Now that you know how to track and identify duplications in the same project, it's time to explore a noteworthy feature of SonarQube: duplication detection across multiple projects. This functionality is unique; you can't find it in any other relevant tool.

Because the duplication-detection engine is core functionality of SonarQube, it's available for all the languages you can analyze with SonarQube. Cross-project duplication detection isn't on by default, so this section will show you how to activate it and how to recognize cross-project duplications in the file detail view's Duplications tab.

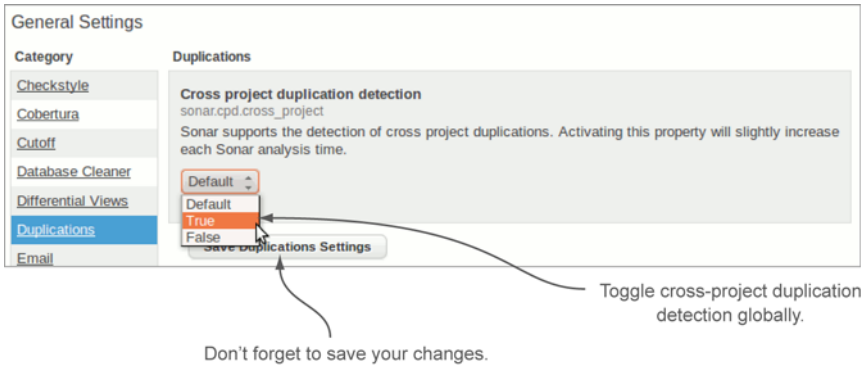


Figure 4.8 **Enabling cross-project duplication detection**

4.4.1 *Turning on cross-project duplication detection*

By default SonarQube is installed with cross-project duplication detection disabled, perhaps because its use incurs a performance penalty during analysis. But it's a powerful tool, and we urge you to turn it on.

This may increase the time needed for completing an analysis, especially when you have a lot of projects of the same language in SonarQube, but the ROI of cross-project duplication detection is worth the hit to analysis speed.

You can toggle its use at a global level or on a project-by-project basis. To set it globally, a logged-in administrator can choose Configuration under the Settings menu at upper right in the interface. A submenu is added next to the left rail; choose Duplications there, as shown in figure 4.8.

In the Duplications settings, choose True for Cross Project Duplication Detection, and save your changes. From now on, all SonarQube analyses will perform cross-project duplication detections. Additionally, you can turn this feature on or off for a particular project by modifying the same attribute under the project settings configuration.

> **NOTE** Remember that if cross-project duplication detection is turned off for a project, SonarQube won't look for cross-project duplications when the code is analyzed. But the project's source code will still be available for cross-project duplication detection to other projects with this option enabled.

4.4.2 *Cross-project duplications in source code tab*

After activating cross-project duplication detection, you'll need to run a new analysis to begin seeing duplications. In this case, to show how duplications across multiple projects are displayed in the file detail view, we reanalyzed the sample project with a different project key.

Once you turn on cross-project detection and start running new analyses, the first thing you notice is that your duplication metrics jump dramatically. This is expected and perfectly normal. But if you expected the duplication density to go to 100%, you

might wonder why it's sitting below that. It's because the duplication-detection engine only tallies up duplicated lines of code. When counting duplicate lines, it ignores `import` statements as well as any blank lines or comments that may be embedded in a duplicated block. As we mentioned earlier, the *density* of duplicated code is based on *physical* file lines (including those lines the duplications engine ignored), so you should expect to see a number below 100%.

Now let's examine the Duplications tab in the file detail view. If you're looking at a file with cross-project duplications, you'll probably see something like figure 4.9, which is similar to what we showed you earlier.

In the figure, notice that `InternationalOrder` is shown twice. As you can see, there is a special icon next to one of the class names, which indicates that the class belongs to another project. Click the icon or the class name to see part of the duplicated block. When you do, the class's project name (linked to its dashboard) is displayed to the left of the full class name (linked to its file detail view), making it even easier to identify the exact position of code repetitions, even between projects. Expanding and collapsing code blocks works the same way across projects as within a project.

Because our example of cross-project duplications analyzed the same project under two different project keys, figure 4.9 shows the same class name displayed twice: once for each project it appears in. Unfortunately, this wholesale duplication of classes isn't as outlandish an example as it should be. We've seen utility classes copied in their entirety from one project to another (class name intact!), not just in private enterprises, but also among well-known open source projects (but not SonarQube).

NOTE As you'll learn, you can analyze multiple branches of the same project by using the `sonar.branch` property. You may think that in this case, if cross-project duplication detection is turned on, all code would be marked as duplicated. To avoid this situation, SonarQube turns off this feature whenever you analyze a project with the `sonar.branch` property.

Now that you've seen SonarQube's suite of duplication-related functionality, it's time to look at how to clean up the mess. If you're not a programmer, you may want to skip to section 4.6, "Related plugins."

Figure 4.9 Cross-project duplication in source code tab

4.5 Cleaning up your duplications

The purpose of this book isn't just to teach you how to use SonarQube, but also to give you adequate examples to improve your code quality and help you evolve as a software engineer. Getting a list of code repetitions is a great start, but just looking at them won't fix the problem, unless you have some of Uri Geller's talents.

This section includes practical examples of applying software-engineering best practices to code-duplication issues. First we'll look at the extract pattern, then we'll explain the delegate pattern, and finally we'll present a case in which you need to create a new library to eliminate duplications across multiple projects.

4.5.1 Introduction to refactoring patterns

Teaching you refactoring patterns in general is out of the scope of this book. For that, you should consult *Refactoring to Patterns* by Joshua Kerievsky (Addison-Wesley, 2004) or *Refactoring: Improving the Design of Existing Code* by Martin Fowler et al. (Addison-Wesley, 1999). Those two books are about refactoring code using best practices that are typically referred to as *patterns*. The following examples use patterns that move code within the same file or to another file in the same project. Before you see them in action, here's a brief explanation of each pattern:

- *Extract method*—Used for duplicate code in the same class. All you have to do is create a new method, place the repeated block of code in this method, and replace the duplicated blocks with invocations of the new method.
- *Pull up field*—Usually used in conjunction with the extract method. The basic idea is to move a field used in two or more subclasses up to a superclass.
- *Extract superclass*—Applied in the case of code repetitions in different classes. When classes have the same or even similar features, consider moving those features to a superclass.
- *Extract class*—A combination of the extract method and extract superclass, applied mainly to unrelated classes. Move code that's repeated in unrelated classes to a new utility class, and then invoke its methods from the source classes.

4.5.2 Applying patterns to remove code duplication

Going back to the `Order` and `InternationalOrder` classes, let's look at how to eliminate the duplications between them. Thanks to SonarQube, you know exactly where the problem code is, so we can easily start refactoring our classes in order to get rid of it.

From a design perspective, the classes have a lot in common. Each is responsible for holding information about an order placed by a customer (domestic or international). The code that returns the total amount is exactly the same from class to class, but the classes differ in how they compute the discount and sales tax. Because the sales tax is based on a customer's country, you don't need to make any changes in the `getTax()` method. But you do need to do something about the other two methods. Because you have similar classes with duplicate functionality, the extract superclass method is the best pattern to use.

Refactoring

Refactoring is a practice that developers should learn and be able to apply in their everyday activities. But it requires experience in software engineering, discipline, and broad knowledge of many related topics. Among its main purposes are optimizing code, removing duplications, and increasing maintainability.

There is one thing to consider every time you want to refactor your code: ensure that the affected classes are covered by the correct unit tests and that after you complete the refactoring, the same unit tests aren't failing. Only then can you be confident that you haven't created any bugs or other side effects.

Now that you know which pattern to use, you're ready to begin refactoring. Start by creating a new class, `AbstractOrder`, and move the `getTotal()` and `getDiscount()` methods into it, as well as any shared members and their getters and setters. Because `getTotal()` calls `getTax()`, you also need to specify an abstract `getTax()` method to be implemented by your concrete subclasses. (That `getTax()` call is why we made this an abstract class to start with; another option would have been to leave the parent class concrete and implement an overridable, default version of `getTax()`.) The following listing details the `AbstractOrder` class after refactoring.

Listing 4.3 `AbstractOrder` class

```
public class AbstractOrder {
  private List<OrderLine> orderlines = new ArrayList<OrderLine>();
  //Add/remove order line code omitted

  public BigDecimal getTotal() {
    BigDecimal total = BigDecimal.valueOf(0);
    for (OrderLine orderLine : orderlines) {
      total = total.add(orderLine.getOrderLineTotal());
    }
    BigDecimal discount = total.multiply(getDiscount());
    total = total.subtract(discount);
    BigDecimal tax = total.multiply(getTax());
    total = total.add(tax);
    return total;
  }
  public final BigDecimal getDiscount(){
    return BigDecimal.valueOf(0.10);
  }
  # Abstract method to be implemented by concrete classes
  protected abstract BigDecimal getTax();
}
```

With the superclass in place, you can refactor the `Order` and `InternationalOrder` classes as shown in listings 4.4 and 4.5.

Listing 4.4 Refactored `Order` class

```
public class Order extends AbstractOrder{          <-- Order for U.S. residents
  private Customer customer;
  //Setters and Getters omitted

  private BigDecimal getTax() {          <-- Default sales tax for U.S. residents
    return BigDecimal.valueOf(0.20);
  }
}
```

Both refactored classes now extend from the `AbstractOrder` class and implement the abstract method `getTax()`. The code contained in that method is the only difference in those classes. The rest of their behavior, which is the same, is inherited by the `AbstractOrder` class.

Listing 4.5 Refactored `InternationalOrder` class

```
# Order for international customers
public class InternationalOrder extends AbstractOrder{
  private InternationalCustomer customer;
  //Setters and Getters omitted

  private BigDecimal getTax() {
    return (BigDecimal.valueOf(customer.getCountry().getTax()));
  }
}
```

Tax based on customer's country

With these modifications in place, a new SonarQube analysis will show that you've not only removed any duplications between classes, but also decreased the complexity of your code. It looks much cleaner than before and is much more maintainable. Next, we'll finish our examples by considering when you might need to create a new commons library.

4.5.3 Time for a new commons library?

In the previous section, we looked at evolving code to eliminate duplications in a single project. Assume that once you finish that project, you start a new one. After a few weeks, you find yourself in a similar situation, where you again need to use an `Order` class.

Remembering your work on the previous project, you copy the classes you need into your new project, feeling a little smug that you didn't have to rewrite them. What you've forgotten is that SonarQube's detection mechanism can identify code duplications across multiple projects. Because both projects now have exactly the same `Order` class implementation, SonarQube registers a new duplication for each file.

How you should handle this kind of duplication? It's time to start thinking about creating a new library for the functionality that's common to both projects. The process of creating such a library is similar to the process you used to create the `AbstractOrder` class.

Apache

Apache is the father of commons libraries. The purpose of such libraries is to develop and maintain reusable Java components that can be used by other applications or systems. Currently, several commons libraries cover various fields of interest, such as string manipulation, file I/O, logging, and many more. For more information, see Apache's website at http://commons.apache.org/.

After creating the new library project, follow these steps to refactor your code:

1. Move the `AbstractOrder` class and its concrete implementations to the new library.
2. Add the library as a dependency in your existing projects.
3. Modify your code to access the classes of the new library.
4. Run a SonarQube analysis, and pat yourself on the back when it shows zero duplications.

4.6 Related plugins

SonarQube has become a standard for code-duplication detection, and it's by far the most stable and mature tool in this category. But it can be extended by the development of new plugins (if you want to create one, check out chapter 16 of this book), with brilliant ideas for more functionalities and features. This section covers a plugin that's related to duplicated code and that extends SonarQube's default metrics and capabilities: the Useless Code Tracker plugin.

If you tried to give another definition for *duplicate code*, you'd probably end up with something like "lines that are useless and could be deleted from the code base." The Useless Code Tracker plugin reports on exactly this kind of metric by adding to a SonarQube analysis the meaningful numbers explained in table 4.3.

Table 4.3 Metrics of the Useless Code Tracker plugin

Metric name	Metric description
Lines in Duplications	Although it may seem identical to Duplicated Lines as described in table 4.2, this metric is a little more sophisticated. It reports actual lines that may be removed from your code, not just duplicated lines. For example, if your code has 100 duplicated lines in four blocks (that is, 25 lines repeated four times), then the duplicated lines to reduce should be 75.
Lines in Unused Private Methods	To activate this metric, you have to add one of two rules to your rule set: `PMD:UnusedPrivateMethod` or `SQUID:UnusedPrivateMethod`. See chapter 13 for more detailed descriptions of rule profiles.
Lines in Unused Protected Methods	To activate this metric, you have to add one of two rules to your rule set: `PMD:UnusedProtectedMethod` or `SQUID:UnusedProtectedMethod`. See chapter 13 for more detailed descriptions of rule profiles.

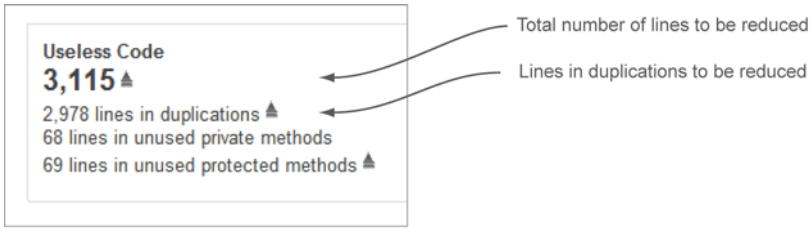


Figure 4.10 Useless Code Tracker plugin

You can add the plugin's widget to any dashboard, and because it isn't categorized, you'll find it if you select None as the widget's filter. Figure 4.10 shows how the metrics described in table 4.3 are displayed to the end user and what they represent.

The power of this plugin is that it doesn't execute any duplication detection (including cross project) itself, but rather relies on SonarQube's embedded mechanism and the results of duplication analysis. This means it can be used with any language with duplication detection. To see it in action, add the Useless Code Tracker plugin to your dashboard and run a new analysis. We're sure you'll find it useful.

4.7 Summary

At this point, you understand how to use SonarQube to detect duplicate code. Even better, you understand where duplications come from, why they're a basic software quality issue, and how they contribute to new bugs.

Face it! You can't avoid the creation of duplicate code. But SonarQube will help you find code duplications in the same file, in the same project, or even across multiple projects, so you can clean up your duplications.

There are four metrics related to code duplication. The most important is the density of duplications, which represents the number of duplicated lines relative to the total physical lines of the project. SonarQube shows you all the duplications metrics on a project, package, or class basis. Further, you can use the Duplications tab in the file detail view to see the location of a duplication in a class.

Just looking at your duplications isn't enough. Knowing that you have code duplications is the first step; eliminating them is the final goal. You can begin to accomplish that with the patterns you learned here, such as extract method and extract class.

5 Optimizing source code documentation

This chapter covers

- To document or not?
- The metrics of commenting
- Identifying undocumented code
- Simplifying your documentation strategy

You're probably wondering why documentation is a topic in a book about a software quality tool. First, let's be clear. This chapter isn't about technical documentation or user/administration guides. It's not even about design or requirements artifacts. It's all about understanding your source code. And because code understandability has a direct connection with quality, comments and documentation form one of SonarQube's seven Axes of Quality.

In this chapter, we'll look at what kinds of metrics are computed for comments and documentation. You'll see how they're reported on the dashboard, what you see at the file level, and how to identify undocumented source code. We'll talk about why documentation is important, what kind of comments to avoid, and how to create or enhance your documentation process.

If you don't think comments and documentation are an important part of your development process today, we believe this chapter will convince you to reconsider. Besides, the standard default Java API documentation is completely useless. Right?

5.1 To document or not?

Sam is a junior developer who's new on the team. She's green but talented, so she gets the task of integrating the in-house notification (email, SMS, and so on) library into the project. After discussing the requirements, she gives an estimate: "I'll be done in two days!"

You've warned Sam that the library's documentation is out of date, so she starts by looking at its code, but she realizes pretty quickly that it doesn't match what's in production. Even worse, the guy who wrote most of it doesn't work here any more. So, she turns to the production version's Application Programming Interface (API) to figure out how to use it.

Sam knows the library provides a single public class for sending event notifications with email or SMS. "It can't be that hard!" she thinks. Then, she sees the API.

"Which one should I use?" she wonders, staring at the method signatures:

```
public void sendMsg ( String varA, String varB )

public void sendMsg ( List list, String var ) throws Exception

public void sendNewMsg ( String varA, String varB ) throws Exception
```

She has little choice but to try all three methods and see what happens. After a couple days of testing, she settles on `sendNewMsg()` and finishes the integration in just under a week. She checks in her changes, but she still isn't sure she picked the right method, and she's embarrassed that her two-day estimate proved far too optimistic. Sam overshot her estimate by three days trying to figure out the API, but writing just a little documentation for these methods wouldn't have taken the original developer more than a few minutes.

Sam's example shows only one problem with not properly documenting your code. Without access to the correct source, she struggled to use a library that was written by someone else and left uncommented. Now imagine a developer trying to maintain a whole legacy system without comments or documentation. Even simple changes could take days longer than they should as she struggles to understand what each piece of code is actually doing.

Before we move on, we'd like to clarify some things. It's important that documentation should start by having clean code with self-explanatory names for entities (classes, methods, parameters, and so on). Then, when necessary, useful comments should be added. The code snippets we just examined have no clean code and no documentation, and that's why Sam struggled to determine the correct method.

Furthermore, we'd like to consider—and we advise you to do the same—that every public method or class is an API. And an API should be documented, because its purpose is to be used by others. You may wonder how to handle protected methods. Well,

there's no rule of thumb: it depends on how you're using the protected method. For instance, when you implement the Template Method pattern (www.oodesign.com/template-method-pattern.html), concrete classes may need to implement a protected method. That's a good case in which documenting this abstract protected method seems a good idea.

Finally, we believe you should start treating comments like source code: they will be updated throughout time. Therefore, you should only keep the useful ones, to make sure maintenance cost is kept to the bare minimum.

Most developers (including us) have left documentation to the end of an implementation and then "forgotten" to do it. Even worse, some coders actively deride documentation as a waste of time and refuse to do it because "Time is precious, and the task is meaningless." But the few minutes it takes to properly document code can save hours down the road, not just for green developers like Sam, but for every subsequent developer who will need to use or maintain a given method, class, or library. It's an investment—one you should make. If you haven't been held to a rigorous standard of documenting so far, you'll have some catching up to do, but SonarQube can help by providing comment and documentation metrics at the project, component, and file levels. We'll start by looking at those numbers and what they mean.

5.2 Even commenting has its own metrics

By now, you're probably familiar with SonarQube's default project dashboard. The widget that displays the comment and documentation metrics is shown in figure 5.1. It's the same one we looked at in chapter 4, in our discussion of duplications. (The widget shown in this section comes from SonarQube release 3.6. Since version 3.7, the widget has been split in two widgets that are identical to the ones described in section 5.5.1.) Duplications are on the right side of the widget, but in this chapter we'll focus on the left side.

5.2.1 How SonarQube calculates metrics

As you see in figure 5.2, there are four documentation-related metrics: a pair for comments and a pair for API documentation. Each pair consists of an absolute value and a density computation.

For the first couple of metrics, the ones counting comment lines, we can't tell you what the best number is (we'll come back to why that is in a moment). For the third number, % Docu. API, you want to see a high number; and the last value, Undocu.

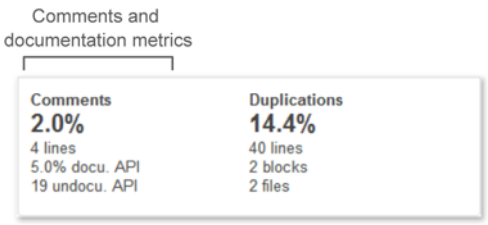


Figure 5.1 The comments and duplications widget appears in SonarQube's default dashboard.

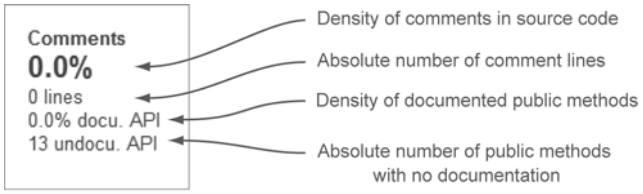


Figure 5.2 SonarQube metrics shown in the comments and documentation widget

API, should be as low as possible. Before we look at how to interpret these metrics, look at table 5.1 to see how SonarQube calculates their values.

Table 5.1 Commenting and documentation metrics definitions

Metric	Explanation
Comment Lines	Absolute number of comment lines. This metric is calculated differently for each programming language. For instance, in Java, all Javadocs (class, method, property) plus all single or multicomment lines and all commented-out code are counted as comment lines. Other comments, such as empty comment lines and header comments, aren't counted. Comments in C are simpler. Every non-blank, non-auto-generated comment line is counted. SonarQube's online documentation details how this metric is computed for other languages. See http://mng.bz/lOnd.
Comments (Density of Comment Lines)	This is the percentage the widget starts with. It's calculated for all languages based on the following formula: Comment Lines / (Lines of Code + Comment Lines) * 100 Chapter 1 details how lines of code are calculated.
Public API	This metric isn't shown directly in the widget; it factors in to other calculations and is reported at the file level in the Source tab's header. It's an absolute number, which is calculated differently for each programming language. For instance, in Java it's based on the following formula: Public Classes + Public Methods + Public Properties Notice that setter and getter methods, as well as final static properties, aren't counted. They're assumed to be self-documenting.
Public Undocumented API	This metric is the absolute number of Public API (as just described) without any documentation. For instance, in Java, a public method with no Javadoc comment is considered undocumented.
Density of Documented API	This metric is calculated based on the following formula for all programming languages: (Public API – Public Undocumented API) / Public API * 100

Before we move on, let's look at a simple code example and walk through calculating SonarQube's comment metrics. Listing 5.1 shows the `InternationalOrder` class you first saw in chapter 4. In order to give a more complete comment calculation example, we've added several different types of comments and removed any obsolete lines.

Listing 5.1 `InternationalOrder` class

```
public class InternationalOrder {
  private InternationalCustomer customer;
  /** Add - remove order line code omitted */
  public List<OrderLine> orderlines = new ArrayList<OrderLine>();
  /**
  * Calculates total amount of an order.
  * @return total amount as a BigDecimal number
  */
  public BigDecimal getTotal() {
    BigDecimal total = BigDecimal.valueOf(0);
    for (OrderLine orderLine : orderlines) {
      total = total.add(orderLine.getOrderLineTotal());
    }
    BigDecimal discount = total.multiply(getDiscount());
    total = total.subtract(discount);
    // Multiply with tax number
    BigDecimal tax = total.multiply(getVat());
    total = total.add(tax);     //  total = total.add(tax);
    return total;   }
  private BigDecimal getTax() {
    return  (BigDecimal.valueOf(customer.getCountry().getVat()));
}
  private BigDecimal getDiscount() {
    return BigDecimal.valueOf(0.10);
  }
}
```

Javadoc for class is missing

Javadoc for property

Javadoc for method

Comment in one-line format

Commented-out code

Default discount for non-US residents

First, let's count the lines with comments. There is one line above the `orderLines` property, two lines in the `getTotal()` method's Javadoc, and a single-line comment. The two blank comment lines aren't counted because empty comment lines are left out of SonarQube's calculations. The commented-out code line (just before the return in the `getTotal()` method) is still a comment, so it gets counted too. So the number of comment lines is 5, and there are 21 lines of non-blank, non-commented-out code. With the counts for Lines of Code (21) and Comment Lines (5), you can compute the density based on the formula in table 5.1: approximately 19.2%.

What about Public API? The public class has one public property and one public method, so the number of Public API is three (one public class + one public method + one public property). There are Javadoc comments only for the property and the method, so the number of undocumented API is one, because the class itself isn't documented. The calculation for the density of the documented API is easy: (3–1)/3*100 = 66.7%. Next, we'll dig deeper into the real meanings of these metrics.

5.2.2 What the numbers are telling you

Let's start with a closer look at the two density metrics. We strongly believe they're the most critical and give you the most valuable information. We'll start with Density of Comments, the first metric in the widget. Imagine you have three different projects,

each with 1,000 physical lines; and assume that you get the following numbers for their comment density: 0%, 50%, and 100%. The first one, 0%, means you have absolutely no comments in your project (that is, 1,000 lines of code and 0 comments). If you see a value of 50%, then you have as many lines of comments as you do of code (500 lines of code and 500 comments). Finally, if comment density is 100%, then your files consist completely of comments.

What's the story on this metric? Which of those scores is the best? Actually, we don't like any of them. A number north of 50% implies that you're over-commenting your source code, whereas a number near 0% means there are too few comments. There's no magic number, though we think somewhere between 20% and 30% is a good score for most projects. But a score outside of that range doesn't mean you should start adding or removing comments.

The significance of the Density of Documented API metric is more obvious and clear-cut. Higher numbers tell you your code is more thoroughly described. Normally you should shoot for a number near 100%. This is particularly important when the project is something like an in-house commons library (code packaged to be shared with other teams), and it's critical if the project is used by third-party systems for integration purposes. But even when you do see numbers near 100%, that doesn't necessarily mean you're in the clear. Unfortunately, SonarQube can only measure the quantity, not the quality, of your API comments. Many IDEs auto-generate documentation "shells," which could easily be used to game the system on these metrics, intentionally or not. (We've accidentally done it.) Regardless of whether they're filled in, SonarQube counts those shells as API documentation, so it would be possible to have high documentation scores without having documentation; a spot check of fully documented code might prove worthy of your time.

The importance of API documentation on code that's written for other teams or companies is obvious, but you still need to document non-commons projects, and you should even consider documenting private methods and/or choosing method and parameter names that properly describe their purpose. Remember that your first priority is to keep your code clean and understandable. After that, you can still document it. Even if your memory *is* perfect (will you *really* remember what every method does six months from now?), you're not the only one who will ever maintain or use your code.

If you're on a new project, your job is easy: document as you go. But if you've got some catch-up to do, you'll want to know which parts of your project need attention first. So, next we'll look at how to identify areas of a project with low levels of documentation.

5.3 Identifying undocumented code

So far you have a broad understanding of your project's comments and documentation. You can easily see from the numbers on the dashboard how much of your public

API is documented. And if your % API documentation is high, then you know that most of your code is probably documented.

Next, we'll drill in to see documentation at the file level. Unlike what you've seen for other metrics, there is no separate tab in in the file detail view that's dedicated to comments. Instead, you'll be directed to the file's Source tab, where you can see the file's comment metrics in the header and read the comments themselves in the context of the code.

5.3.1 Finding files to improve documentation

Starting from the dashboard, click-through on the Public Undocumented API metric. You'll land at the familiar drilldown view of measures, and you can click any module or package to narrow the file list to only those in the component you chose.

To the right of each component name is the value of the metric you clicked from the dashboard. Remember that SonarQube's drilldown views are always sorted "worst first," so you see the components that most need your attention at the top. So far, this should all be familiar, but there is one small variation to point out, and it's shown in figure 5.3.

As you see, even though we clicked-through on the Public Undocumented API *density* (a percentage), the drilldown view is based on the *absolute number* of public undocumented API.

The other thing to keep in mind is that components with perfect scores for the metric you're drilling in to are omitted from these lists. That means if you do decide to do a spot check of documentation quality (versus quantity), you'll have a little extra digging to do.

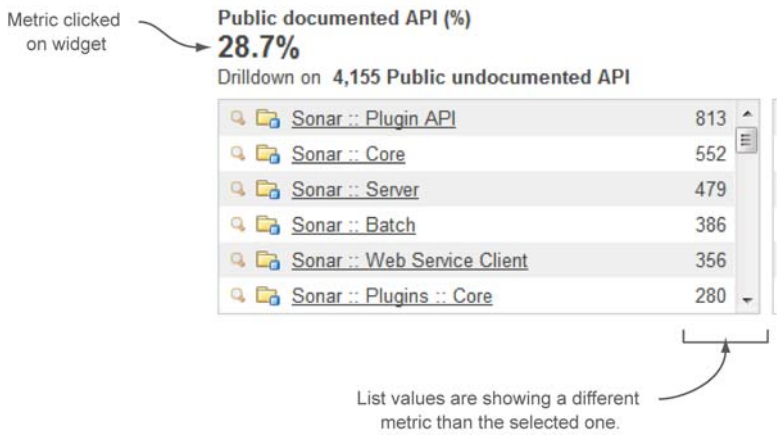


Figure 5.3 When the click-through metric is Public Undocumented API, the drilldown view is a variation on the norm. Instead of seeing files sorted by that density, they're shown with and sorted by the absolute number of undocumented API.

5.3.2 Viewing the generic tab in the source code viewer

Now click any of the files in the list on the right to see the file detail view. As we said earlier, there isn't a dedicated tab for documentation. Instead, you'll find yourself looking at the Source tab, which shows a metric summary for the file and its full source code. You might get fewer tabs at the top, but the selected tab is still the same. Figure 5.4 highlights the comments and documentation metrics in the header.

The header contains several metrics we've covered in previous chapters and a few we'll talk about in chapter 6 when we discuss complexity. For now, focus on the third and fourth columns. They contain the documentation and comment metrics. You'll see that all the comments and documentation metrics in the dashboard widget are shown in the header as well, but this time on a file level. Additionally, you get one more number here that you don't see on the dashboard. It's Public API, which we described in table 5.1 as the number of public *things* (classes, methods, members) that *ought* to have documentation.

Below the header, you should see the file's full source code so you can review the comments and documentation (or lack thereof). Beyond that, there's not much more to say, because there's no special presentation for comments.

At this point, we've fully covered the comments quality axis. You've seen the metrics, their meaning, and how SonarQube computes them. You've also seen how to find the parts of a system that are poorly documented or over-commented.

As you look over these metrics in your own projects, remember that numbers alone are useless if you haven't decided what your target values are for comments or how you're going to improve your scores if they're not where they should be. As you're setting these targets, keep in mind that commons libraries tend to need more documentation than code that's only used by a focused team.

We'll talk more about setting metric targets and strategies for achieving them in chapter 8. The rest of this chapter aims to give you some ideas for making your documentation process a smooth activity, and we'll wrap up by presenting a couple of related plugins we believe you'll find useful.

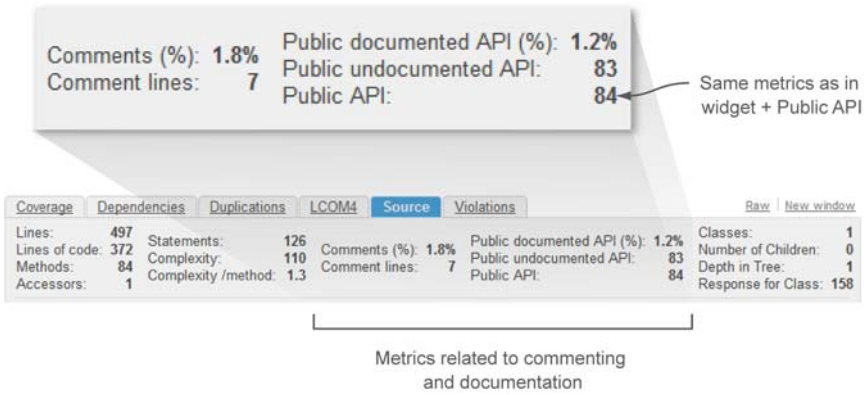


Figure 5.4 **Source code tab's header**

5.4 Simplifying your documentation strategy

At the beginning of the chapter, we discussed some of the reasons developers dislike documenting their code. The truth is, we're all eager to finish the *real* code (don't forget the unit tests) and see our systems running, and any activity that doesn't advance this goal is seen as unnecessary. But unless you're on a one-person project and you plan to remain a one-person shop, maintaining the same old code over and over *for the rest of your career*, you have some obligations to your fellow coders.

The code you write doesn't belong to you, and it's certain that during the project's lifecycle other developers will need to modify it or use it. Don't fall prey to the stereotypical developer egotism: "My code is straightforward. It's not my fault if you don't understand it." Instead, remember that at some point you'll be the new developer on a team again, and write the documentation you'd like to read in that situation.

If you're in the *extreme programming* camp, then your response is that "The code *is* the documentation." To some degree, we agree. No one can overemphasize the importance of well-named classes, methods, and variables. We have seen (much to our lasting dismay!) variables in "professionally" written code named things like `pleaseWork`. We'll never get back the time spent following *that* piece of spaghetti. So yes, the time and effort spent creating good names and writing clean code is invaluable.

Another tenet of extreme programming is that close team communication makes documentation unnecessary. While a system is being created, that's likely true. But the odds are that some day, the code will have to be maintained by someone who wasn't on the original team. Rather than making that person take time reading your clean code, spend a few minutes on documentation to sum up what your well-named method does.

5.4.1 Picking a documentation tool

Let's get back on track and look at how to improve your documentation process. First, every member of the team needs to commit to following the process. Once you have agreement among the team members, you need to pick the tool and the format you'll use for your comments.

The standard in Java is Javadocs. If you're among those who think default Javadocs looks ugly, keep in mind that you can create a custom XHTML doclet to provide good-looking and professional documentation. For other languages, there are similar tools, like NDoc for C# and DOC++ for C and C++. There's also Doxygen, which has nice SonarQube integration (we'll cover it in the "Related Plugins" section) and can be used for several languages, including Java, C, C++, Python, and PHP. You could even use a combination of tools for best results and customize them to fit your needs. In general, they'll all give you smart and effective ways to generate the documentation of your source code based on the provided comments.

5.4.2 Defining a straightforward process

Once you've picked a documentation tool, it's time to define an easy-to-follow process. To do that, you need to answer some simple questions:

- When should you write documentation?
- What parts of the source should you document?
- What information should the documentation include?
- How will the documentation be generated?

Your answers will guide you in creating the process that best fits your team's needs. Here are our suggestions to keep it simple and elegant.

When to document

First, document as you code. Before you start a new class, jot down a few comments explaining its purpose. The same applies to methods/functions. Briefly describe what they're expected to do.

After finishing the code, review the documentation and update it to reflect any changes you might have made (see figure 5.5). Follow the same steps whenever you need to modify a piece of code.

Which parts of the source to document

Every public class and each of its public methods and properties is a candidate for documentation. But there is no need to document setters and getters or other methods that are so simple that the signature itself is the explanation. The following snippet shows one of our favorite over-documentation anti-patterns:

```
private String name;

/**
* Returns the name
* @return name
**/
Public String getName()
```

In this case, the Javadoc doesn't provide anything the reader couldn't have gotten from the method signature itself—which means it's entirely redundant and, like other duplications, should be avoided. Redundant documentation, like redundant code, needs to be maintained but doesn't usually get attention. Instead, it only adds noise to your source code. Or worse, it stops reflecting the actual purpose of the code because it's not kept up to date, and it adds confusion instead of clarity.

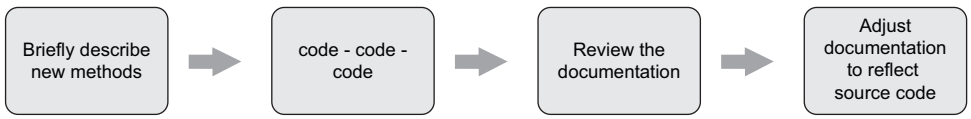


Figure 5.5 Simple documentation process

But you *do* need to document private methods, even though they won't change the numbers you see in SonarQube. Why bother? So that future developers maintaining the code (including you, six months from now) can quickly understand the purpose of each method by reading the clear and insightful summaries you'll write, instead of having to slog back through the logic to re-figure it out the hard way.

WHAT INFORMATION TO INCLUDE

In addition to describing the purpose of each class or method, it's a good idea to include descriptions of input and output parameters as well as any exceptions that might be thrown. When appropriate, you may also want to include things like the version of the API that introduced a particular feature. Include everything you think is important, and keep in mind that documentation should be complete enough that the reader won't need to ask for clarification. Popular IDEs like Eclipse and NetBeans offer many ways to facilitate this process, including the documentation shells we mentioned earlier. They fill in basic information for you and let you focus on the real work.

What you should avoid is as critical as what you document. Try not to comment-out lines of real code; delete them, instead. If you need it back, you can always revert to a previous version of the file from your source control system. So why add noise to your code?

Whether to avoid comments within the body of a method or function can be debated. Many people contend that if your code needs comments to be understandable, what it actually needs is refactoring. Regardless of whether you agree, make sure you agree (or disagree) as a team, so that everyone's on the same page.

HOW TO GENERATE

Once you've started writing documentation comments, you need to decide how to generate and publish the docs. We haven't talked about Continuous Integration (CI) yet, but in part 2 of the book we'll discuss how it will walk your dog and wash your car and generally make your life wonderful. Okay, that may be a bit of an oversell, but CI can make the development process go a lot more smoothly by offloading the tedious, repetitive tasks to computers (where they belong), leaving you and your teammates free to tackle the interesting stuff. There are manual ways to generate your code docs, but ideally you'll set it up as part of your CI process so that no one has to remember to do it and the docs are always up to date.

5.5 Related plugins

We've talked a lot about process. Let's get back to SonarQube and discuss the extensions that are available to help you track or publish documentation. First, we'll revisit the Widget Lab plugin we looked at in chapter 2. It offers extra widgets to give you a clearer and more complete view of your documentation metrics. Then we'll look at the Doxygen plugin mentioned earlier in this chapter. It integrates SonarQube with Doxygen, a popular documentation tool, and lets you view project documentation from within SonarQube. Let's get right to them.

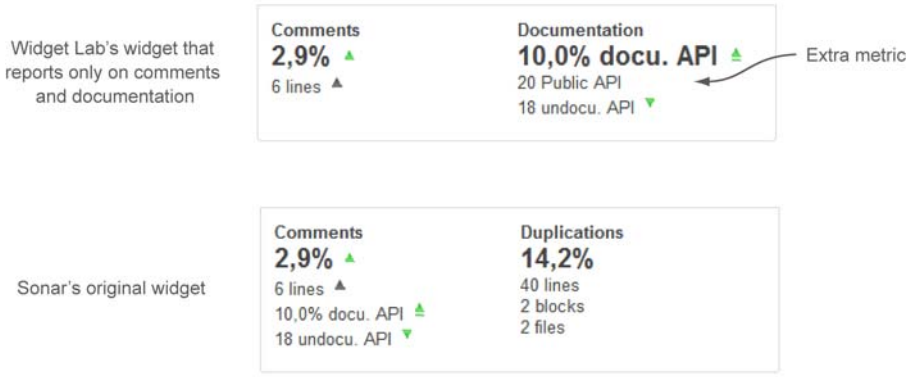


Figure 5.6 Visual comparison between SonarQube's original widget and the cloned widget provided by the Widget Lab plugin

5.5.1 Widget Lab

This plugin is a collection of widgets that offer modified versions of some of SonarQube's core widgets to provide extra functionality. SonarQube's default widget for comments and documentation mixes those metrics in with duplication numbers. So if you wanted to create a dashboard dedicated to documentation, you'd have to put up with irrelevant numbers.

Fortunately, Widget Lab offers a widget that reports only on comments and documentation. The widgets described in this section have been included in SonarQube's core since release 3.7. See figure 5.6 for a visual comparison between the original and the cloned version of the widget.

The Widget Lab version shows the same metrics you'll find in the standard widget, plus one addition: it also shows the total number of Public API. As with the standard widget, clicking any metric in the Widget Lab version sends you to the corresponding metric drilldown. When you're ready to add this widget to a dashboard, you'll find it under the Documentation category.

5.5.2 Doxygen

We said earlier that there's no special tab in the file detail view for documentation, and by default that's true. But you can change that if you like, with the Doxygen plugin.

Doxygen is a popular open source documentation tool with support for multiple languages. It generates docs from the comments in your source code and can output to a variety of formats. HTML is typical, but it also supports RTF (for Microsoft Word processing), PostScript, and hyperlinked PDF, as well as graphs.

What's really cool about this tool is that it can be used for several languages, including C++, C, Java, Objective-C, Python, and many others. For examples and the most up-to-date information on Doxygen, visit Doxygen's official website: http://doxygen.org.

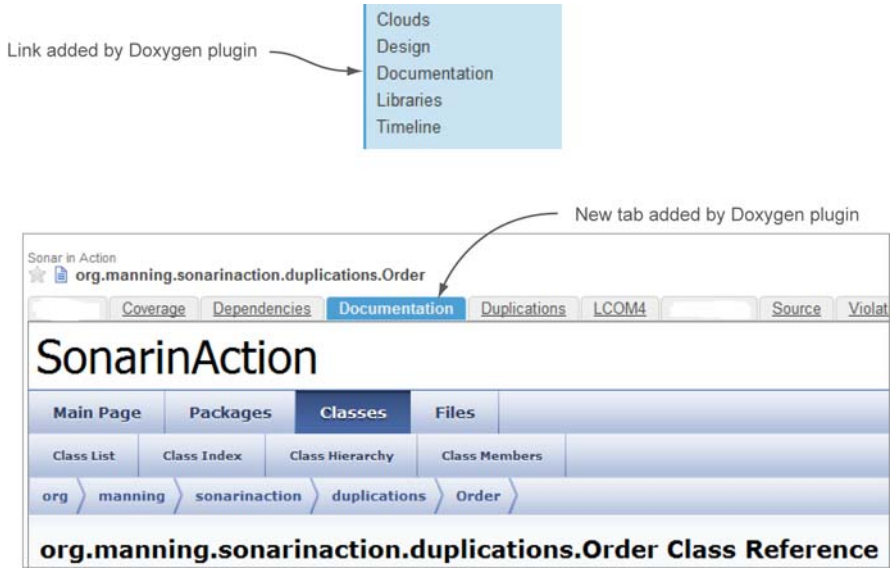


Figure 5.7 Doxygen plugin's new link and new Documentation tab in the source code viewer

To get Doxygen working in SonarQube, you'll need to install the plugin and Doxygen itself. If you want Doxygen to generate graphs, you'll need to install Graphviz as well.

After it's installed, you'll find that the Doxygen plugin is disabled by default, so you'll need to explicitly enable it for each project. At the same time, you can turn on generation of class graphs, caller graphs, and call graphs. Once it's on, run an analysis: you'll see a Documentation link added to the left rail and a new Documentation tab in the file detail view. The former navigates to the main page of documentation, and the latter shows the Doxygen documentation of the selected file (see figure 5.7).

5.6 Summary

In this chapter, we looked at documentation, which is an afterthought for many development teams. But if it's properly managed, spending a few minutes now on documentation can save you a lot of future work. At its heart, SonarQube is about technical debt: the things that were done poorly or left undone, and the things that will stand in the way of future productivity. And missing documentation falls squarely in that category.

All code should be documented, even when its use is limited to a single team. But documentation becomes critical when you're writing libraries for use by other teams or companies.

You've seen that SonarQube provides metrics to help you track and improve your documentation coverage. You've seen how the numbers are calculated, and their importance in the software development lifecycle.

At this point, you know how to spot components and files with poor documentation. You also know how to approach setting up an easy-to-use process to improve the documentation in your projects, and which practices to be wary of or avoid.

Finally, we discussed two open source plugins related to documentation, Doxygen and Widget Lab. Next, we'll look at software architecture and complexity in chapters 6 and 7. Let's move on!

6 Keeping your source code files elegant

This chapter covers

- Distributing complexity
- Lack of Cohesion of Methods: files that do too much
- RFC and couplings: classes with too many friends

An active project gets bigger day by day, week by week. As your software grows, it becomes more complex. New classes are added, and methods and attributes are created or improved. Each time you make a change, you're probably affecting the health of your system's design. Whenever logic is added, the complexity of the file, the package, and the module is increased.

Fortunately, SonarQube can alert you to these kinds of issues. For instance, it can tell you when the quality of your design goes down, when the increasing complexity of a file starts to make it hard to maintain, or when the time comes to break up this complexity *without* reducing the overall complexity.

In this chapter, we'll look at complexity—not from a system-level perspective, but at the level of individual files. How internally complex is the average file in your system? How complex are its interactions with the rest of your system? Both

questions are important because they help you gauge how difficult the average file is to work on and how broad an impact (good or bad) working on that file will have on the rest of your system. Unless you're familiar with a given system, you probably don't know offhand where its "worst" classes are, but SonarQube can help you find those trouble spots and prioritize them for refactoring.

We'll begin by looking at class-level complexity (known as *cyclomatic complexity*, or the McCabe metric). Then we'll move back a step to look at Lack of Cohesion of Methods (LCOM), which takes a more abstract view of each class to ask, "Is this class doing too much?"

After that, we'll move on to the complexity of interactions with Response for Class (RFC), which looks at how a file interacts in a system, and at afferent and efferent couplings, which also fall under the "complexity of interactions" heading. We'll show you how to identify poorly designed classes, discuss refactoring methods, and give you some tips on keeping your source code clean and simple.

6.1 Keeping complexity low

Sam, the junior developer on the team, is still smarting from giving an over-optimistic estimate on her last assignment. She's eager to redeem herself when she's told to change the validation algorithm for Visa credit cards. Visa recently changed the rules for what's valid, and the deadline to have the new validation in place is looming.

Unfortunately, validation for every credit card type the system accepts (American Express, Discover, Diners, MasterCard, Visa, and so on) is handled through the same huge class. Worse, most of it runs through one monster method packed with more spaghetti than an Italian restaurant. Sam knows it should be structured differently, but she doesn't have the time or (she's afraid) the experience to restructure the code. Instead, she spends several hours—with her restless boss asking constantly about her progress—just reading the code.

She finishes near the end of the day, just ahead of the deadline, but nobody is happy. What should have taken less than two hours took a whole day, and even her boss has to admit that it's not Sam's fault. The class was too complex, and she needed lots of time to read the code and become familiar with it before modifying it.

This section teaches you how to spot complex files by using SonarQube. It explains why it's important to keep the complexity as low as you can, and describes briefly how complexity is calculated. At the end, we give you some refactoring tips, and we show how you can minimize the complexity value without changing the output of your code.

6.1.1 Hunting those huge files

Modifying highly complex files like Sam had to do is an error-prone process. Some studies have shown that as complexity rises, so does bugginess. Sam went slowly, and rightly so, because a high level of complexity decreases a file's maintainability and increases the time it takes to understand the code. The harder it is to completely take

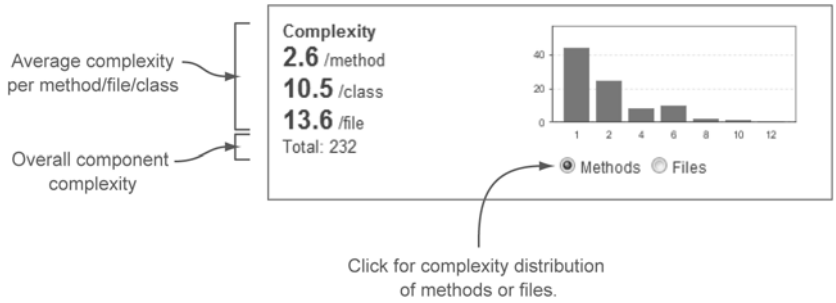


Figure 6.1 Dashboard complexity widget

in a method or file, the more likely you are to subtly (or not so subtly) screw it up when you work on it. Additionally, high complexity makes it difficult to properly unit-test a method; that's why most of the time, using a Test-Driven Development approach prevents developers from generating such complex methods.

How is complexity defined? Is it like the Supreme Court's take on pornography—you know it when you see it? Maybe, but believe it or not, it's also measurable. It's more complicated than this, but loosely, you can think of complexity as the count of pairs of curly braces (real or implied) in a class or method. That's the *McCabe metric*, commonly called *cyclomatic complexity*, which SonarQube reports in the complexity widget shown in figure 6.1.

The numbers in the widget are averages: average complexity per method, class, and file. The number at the bottom is the application's overall complexity, which is a sum of the parts. The graph on the right side of the widget shows the distribution of complexity across either methods or files, based on which radio button is selected.

Because you want to see that graph weighted to the left, the project reflected in figure 6.1 looks pretty good. But switch it to show the complexity distribution across files, as shown in figure 6.2, and the picture changes—literally.

Suddenly that smooth, pretty slope has a spike on the right side. What's this telling us? The first graph shows that most methods have low complexity, and the second one indicates that some files (five, to be exact) have high complexity. This likely means a handful of files have lots and lots of methods. Because getters and setters aren't counted in the complexity equations, you know the spike in the graph doesn't merely reflect classes with a lot of members. No, these are files with a lot going on.

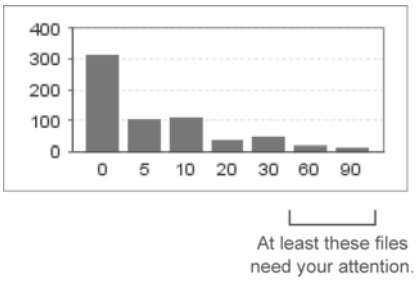


Figure 6.2 Files distribution/complexity bar chart

Because high complexity makes a file harder to maintain, you should consider refactoring any file with a complexity of 60 or more, whether that complexity comes from 2 methods with complexity scores of 30 each, or 10

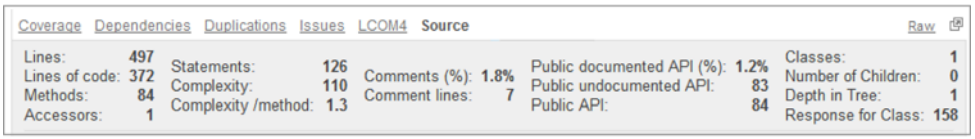


Figure 6.3 Complexity metrics shown in source code viewer default tab

methods each with a score of 6. In the first case, both methods are complicated; but in the second case, the class probably has too many methods. Tracking methods with too-high complexity is important, and in that case, you should activate some rules to get issues and track those methods that are candidates for refactoring (see Chapter 12 for more details).

Either way, these five files definitely need to be examined, but you'll have to look inside the file to decide your next steps. That's where the drilldown comes in. Click-through on the metric of your choice (probably complexity per file) to get there. Once you've landed at the drilldown, the normal worst-first sorting brings the worst offenders to the top. Choose a file, and you'll find yourself on the Source tab. There is no special tab in the file detail view for complexity, but the file's complexity metrics appear in the Source tab header, as shown in figure 6.3.

Okay, so now what? You know the class is a problem, but what do you do with it? To answer that, we'll look at what these numbers actually mean—how they're computed—and then we'll talk about refactoring strategies.

6.1.2 Complexity: what it looks like and how to fix it

We said earlier that cyclomatic complexity can be thought of as the count of pairs of curly braces. But it's more complicated than that. Early `returns` figure in, as do multiple conditions (`&&`s and `||`s) in control structures (`if`s and `while`s.) It's best explained with an example, like the following listing.

Listing 6.1 Example of code that is too complex

```
boolean myMethod(String value) {
if (value == null) {
    return true;
}

if (value.length() != 13) {
    return false;
}

for (int i = 0; i <= 11; i++) {
    cDigit = value.charAt(i);
    digit = Long.valueOf(String.valueOf(cDigit));

    if (((i + 1) % 2) == 0) {
        oddNumber = digit * 3;
         if (oddNumber > 9) {
             oddNumber = oddNumber - 9;
```

Each method starts with a count of 1

Add 1 for branch

Add 1 for early return

Add 1 for branch

Add 1 for early return

Add 1 for control structure

Add 1 for branch

Add 1 for branch

```
            }
            oddNumbersSum = oddNumbersSum + oddNumber;
        } else {                                              <-- Else: doesn't count
            evenNumbersSum = evenNumbersSum + digit;
        }
    }

    totalsum = oddNumbersSum + evenNumbersSum;
    checkdigit = totalsum % 10;
    lastdigit = Long.valueOf(String.valueOf(value.charAt(12)));

    if (lastdigit == checkdigit || lastdigit ==0) {   <-- Add 2 for branch and || (or)
        result = true;                                <-- Add 1 for early return
    }

    return result;                                    <-- Last return: doesn't count
}                                                     <-- Total complexity of 11
```

Cyclomatic complexity can be calculated for any fraction of source code (method, function, class, file, module, and so on). It counts the number of possible paths (branches) through the source code.

By default, every method has a complexity value of 1. For Java, the following keywords and statements are considered branches and add one point to the complexity: `case`, `catch`, `for`, `if`, `throw`, `while`, `&&`, `||`, the ternary operator `?` (which is just a fancy `if`), and `return`, except for the last one in a method. If you manually compute the complexity of the code in listing 6.1, you'll get a value of 11 points, which comes from the default value (+1), 5 ifs (+5), 3 early returns (+3), 1 for loop (+1), and 1 || (+1).

Listing 6.2 shows another example: a method that gets an array of integers and returns a count of array items that are divisible by four. If the array is empty, then an `IllegalArgumentException` is thrown.

Listing 6.2 Method that checks how many numbers are divisible by 4

```
public int divisibleBy4Count(int[] numbers) throws IllegalArgumentException
{                                                     <-- Method starts at 1

    int divisibleCount = 0;

    if (numbers.length > 0 ) {                        <-- +1 for branch

      for (int i:numbers){                            <-- +1 for loop
        if (i % 2 == 0) {                             <-- +1 for branch
          if ((i / 2) % 2 == 0) {
            divisibleCount++;
          }
        }
      }

      return divisibleCount;                          <-- +1 counts as early return

    }
    else {
      throw new IllegalArgumentException();           <-- +1 for throw
    }
  }                                                   <-- Total complexity of 7
```

The cyclomatic complexity of the `divisibleBy4Count()` method comes to 7. Note that even though there's only one return, it increases the complexity of the method because it doesn't come at the end of the method. The compiler might smooth out that little wrinkle, but it still makes the method harder to take in, and that's what you're trying to calculate.

The SonarSource folks suggest refactoring a method when its complexity is greater than 7. The method in listing 6.2 weighs in at 7, just under the threshold, but it's obvious that the method can be improved. You can remove the assertion of an empty array by introducing a new method for performing this check. The new version of the code looks like this.

Listing 6.3 Revised method that checks how many numbers are divisible by 4

```
public int divisibleBy4Count(int[] numbers) throws IllegalArgumentException {
  int divisibleCount = 0;
  assertArray(numbers);
  for (int i : numbers) {
    if (i % 4 == 0) {
      divisibleCount++;
    }
  }
  return divisibleCount;
}

private void assertArray(int[] numbers) throws IllegalArgumentException
{
  if (numbers.length == 0) {
    throw new IllegalArgumentException();
  }
}
```

Method starts at 1
+1 for loop
+1 for collapsed ifs
Return is last statement: doesn't count
Total complexity of 3
Method starts at 1
+1 for branch
+1 for throw
Total complexity of 3

Manually calculating the complexity for each method, you get the following results:

- `divisibleBy4Count()` = 3
- `assertArray()` = 3

The refactoring accomplished a couple of things. First, we reduced complexity by refactoring the basic algorithm. This was a trivial example, but we hope it makes the point that basic improvements like this are often possible. Second, we demonstrated reducing complexity and improving readability by distributing the complexity of one method across multiple methods.

Now that we've looked at cyclomatic complexity, we'll move on to a slightly more abstract metric, which looks at the complexity of what the class is trying to do.

6.2 Lack of Cohesion of Methods: files that do too much

Earlier, we talked about two theoretical classes with a cyclomatic complexity of 60 each: one with 10 methods that each scored 6, and one with 2 methods scoring 30 each. We've already looked at how to approach the class with two large methods.

The next metric we'll look at, Lack of Cohesion of Methods (LCOM), helps address the class with lots of methods. At the time of this writing, the metric is only available for Java projects, but sooner or later more languages' plugins will compute this metric. First we'll discuss the widget that reports on the metric and why it's not shown in the default dashboard. Then we'll show you the source code tab viewer that presents the LCOM of a class in a nice, clean way. After that, it's time for coding. Through a short example, you'll compute the LCOM metric of a simple class; then you'll try to refactor it to improve it.

6.2.1 Getting reports about the LCOM metric

At its root, LCOM is the count of the number of responsibilities a class has. There are several variations on the LCOM algorithm. SonarQube uses LCOM4, the fourth one to be published, because it's the most convenient for computations in real source code. The previous versions are used in scientific and research circles, but not in the field.

> **LCOM4 Variations**
>
> LCOM4 is also known as the Hitz and Montazeri version. If you'd like to know more about LCOM and other object-oriented metrics, look up the paper Hitz and Montazeri presented at the International Symposium on Applied Corporate Computing in Mexico, in October 1994. You'll find it here: http://mng.bz/G4k6.

Although complexity is one of SonarQube's Seven Axes of Quality, the related widgets aren't included in SonarQube's default dashboard because the metrics are considered "too hard." The thinking is that for the majority of SonarQube users, the widgets would only add noise to the dashboard. This won't apply to you, because after reading this chapter you'll have a thorough understanding of how these metrics are computed and what they tell you about your source code.

Before we get to the meat, you should add the LCOM4 widget to your dashboard. Chapter 14 gives you low-level details on how, but if you're logged in as an administrator, you'll find an intuitive interface behind the Configure Widgets link at upper right on any dashboard. While you're at it, add the Response for Class widget, too. You don't need it for this section, but you'll want it for the next one. The two target widgets are filed in the Design widget category, and you can see a preview of them in figure 6.4. Don't be surprised that they already have data. Even though the widgets weren't included on your dashboard, SonarQube has been calculating these values all along, so the normal "changes take effect after the next analysis" rule doesn't apply here.

The primary number in each of these widgets is the metric's average per class across the project, and the graphs represent a complexity distribution. In general, you want the numbers to be low and the graphs weighted to the left. For LCOM4, you also see the density of suspect files—those with LCOM more than 1. As you'll see, every file with LCOM >= 2 is a candidate for refactoring.

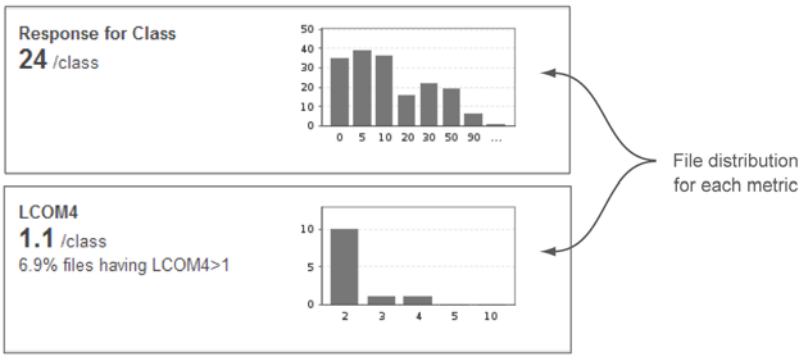


Figure 6.4 **LCOM4 and RFC metrics shown in dashboard widgets**

From the dashboard, click any of the LCOM4 numbers (or the LCOM4 graph) to get to the drilldown view. You'll get the standard drilldown behavior: only files or classes that need work, sorted worst first. Click a class, and SonarQube reports on the LCOM4 value for individual files in an LCOM4 tab with a graphic presentation of connected components, as shown in figure 6.5.

"But what does this mean?" you may be wondering. Let's find out.

6.2.2 *Counting responsibilities*

We could try to explain LCOM4, but the best way to give you a fundamental understanding of it is to show how you'd compute it by hand in a simple class. Listing 6.4 shows a simple `AmericanBreakfast` class with some local members and several methods accessing them. The purpose of the class is to prepare an American breakfast, update the availability of the materials, and print a message. A client would probably

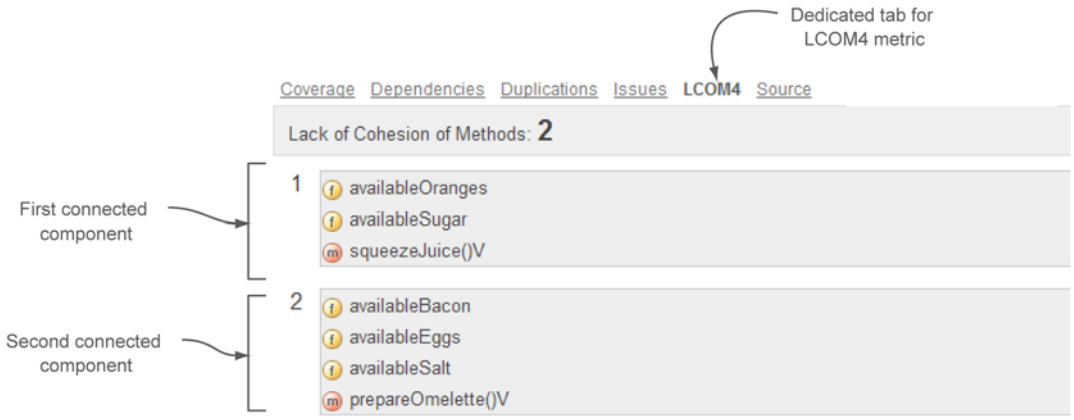


Figure 6.5 **SonarQube reports on LCOM4 with a separate tab in the file source code viewer. The beauty of the LCOM4 tab is that you get a visual representation of which methods (an *m* in a reddish circle) are connected to which fields (an *f* in a yellow circle), and how the relations are formed.**

use some or all of the public methods provided to prepare a partial or complete breakfast. This code is a good example of a class with LCOM4 > 1, because its first version has more than one responsibility.

Listing 6.4 `AmericanBreakfast` class with LCOM4 > 1

```
public class AmericanBreakfast {
  private int availableOranges = 10;
  private int availableEggs = 5;
  private int availableBacon = 4;
  private int availableSugar = 10;
  private int availableSalt = 10;

  public void prepareOmelet(){
    fryBacon();
    bakeEggs();
    availableSalt-=1;
    System.out.println ("Omelet is ready!");

  }
  public void brewCoffee(){
    availableSugar -=1;
    System.out.println ("Coffee is ready!");
  }
  private void bakeEggs(){
    availableEggs -=2;
  }
  private void fryBacon(){
     availableBacon -=1;
  }
  public void squeezeJuice(){
    availableOranges -=2;
    availableSugar -=1;
    System.out.println ("Juice is ready!");
  }
}
```

LCOM4 counts the class's responsibilities by looking at its methods and members and grouping together ones that are connected. (Kind of like the six degrees of Kevin Bacon, except that you don't care how many degrees there are.)

All class members used in a given method are seen as being connected to both the method itself and to each other. If another method uses those same fields (or some subset of them), then it's connected to those members and to the first method as well; it's added to the relation connection. If it uses totally different members, it forms a new, unrelated connection with those other members. Methods can be also considered connected if one calls another and vice versa.

To sum up, two methods (`m1` and `m2`) are grouped together if at least one of the following is true:

- Method `m1` invokes method `m2`, or method `m2` invokes method `m1`.
- Both methods (`m1` and `m2`) use at least one of the same class attributes.

The `AmericanBreakfast` class in listing 6.4 has five members and several methods. Let's look at how many concrete (unrelated) components exist in the class; in other words, let's calculate the LCOM4.

The `prepareOmelet()` method invokes `bakeEggs()` and `fryBacon()` and uses the attribute `availableSalt`, so all three methods and the `availableSalt` member belong to the same group. Further, `bakeEggs()` and `fryBacon()` access the `availableEggs` and `availableBacon` members. Based on the previous definition, you need to add the two methods and both members to the group just identified.

Now look at the remaining public methods: `brewCoffee()` and `squeezeJuice()`. They're connected because they use the `availableSugar` attribute. Notice that you also need to add the `availableOranges` property to this group because it's accessed by the `squeezeJuice()` method. You might argue that you don't add sugar to your orange juice; well, we don't either, but it's a good way to get the kids to drink up without complaint. Also, it works great in this example! It's obvious that these two methods have no relation with the first group, so they form a new component.

Figure 6.6 shows graphically what we've just walked through—which, by the way, yields an LCOM4 of 2 for the `AmericanBreakfast` class we're examining.

So, what's the best LCOM4 score? It's 1. This means all the methods and members in a class are well connected to each other and the class adheres to the Single Responsibility Principle, which as you might guess says that a class should have only one responsibility. A value of 0 implies a class with no methods, not even setters and getters, such as `package-info.java`. In other words, it's a file with no code in it. And any value higher than 1 indicates that the class is a candidate for refactoring. Notice that we used the word *candidate*. In a minute, we'll show you why an LCOM4 value greater than 1 doesn't always mean bad design.

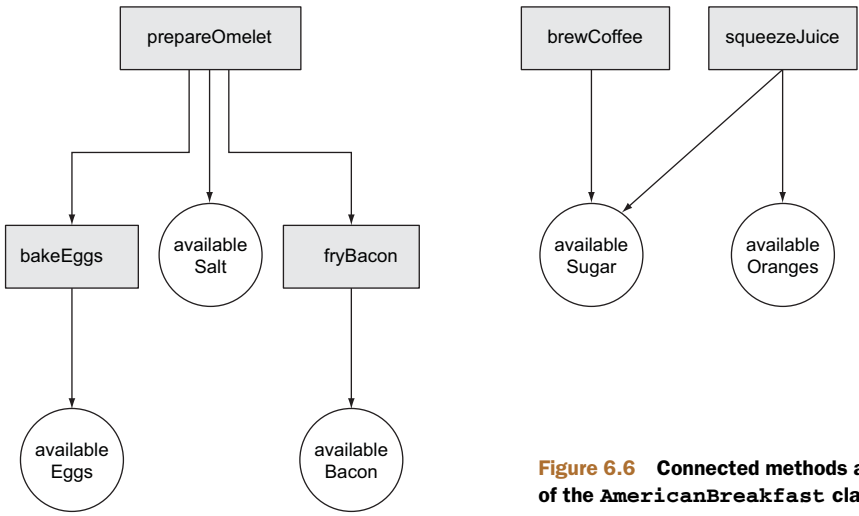


Figure 6.6 Connected methods and attributes of the `AmericanBreakfast` class

6.2.3 Refactoring for fewer responsibilities

With an LCOM4 score of 2, the `AmericanBreakfast` class is a candidate for refactoring. This means it breaks the Single Responsibility Principle because it's responsible for two tasks: preparing an omelet and fixing drinks. Let's refactor the class to minimize its LCOM4 score. There are two ways to go about this: split the class by responsibility, or add cohesion (which could be a hack, depending on the circumstances). If an American breakfast always contains orange juice, coffee, and an omelet, then you want to go the "add cohesion" route and prevent the client from calling classes that prepare only a portion of the breakfast. To do that, you make all the methods private, and add a new public method like so:

```
public void prepare(){
  prepareOmelet();
  brewCoffee();
  squeezeJuice();
```

Running a new SonarQube analysis gives an LCOM4 score of 1 for the class because the new `prepare()` method has tied together the formerly unconnected component groups, as figure 6.7 illustrates.

The second approach is more suitable if you want to allow clients to select which parts of the breakfast should be prepared. At least one of the two groups identified in figure 6.6 should be moved to a new class. Listings 6.5 and 6.6 show the code after the refactoring process. Here's the refactored version of the `AmericanBreakfast` class.

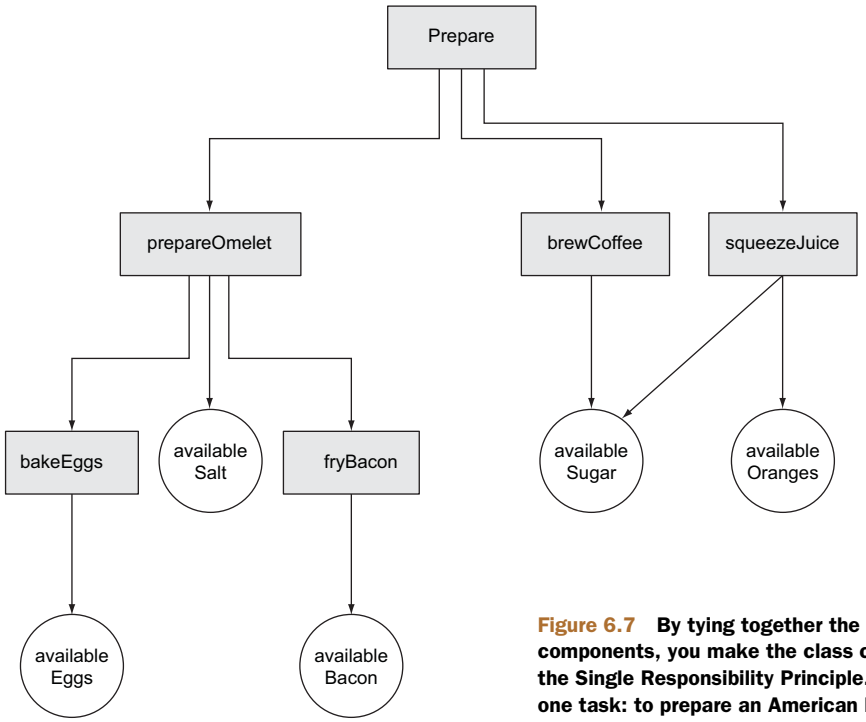


Figure 6.7 By tying together the separate components, you make the class comply with the Single Responsibility Principle. It has only one task: to prepare an American breakfast.

Listing 6.5 Refactored `AmericanBreakfast` class with LCOM4 = 1

```
public class AmericanBreakfast {
  private int availableOranges = 10;
  private int availableSugar = 10;
  private Omelet omelet = new
     Omelet();

  public void prepareOmelet(){
    omelet.prepare();
  }
  public void brewCoffee() {
    availableSugar -= 1;
    System.out.println("Coffee is ready!");
  }
  public void squeezeJuice() {
    availableOranges -= 2;
    availableSugar -= 1;
    System.out.println("Juice is ready!");
  }
}
```

New class reference to Omelet class, introduced after refactoring

Omelet preparation is delegated to Omelet class

Methods not modified during refactoring

Listing 6.6 shows a new class introduced after the refactoring (`Omelet`), which is responsible for preparing the omelet. Notice that part of the `AmericanBreakfast` class has been moved to the `Omelet` class.

Listing 6.6 `Omelet` class introduced after refactoring `AmericanBreakfast`

```
public class Omelet {
  private int availableEggs = 5;
  private int availableBacon = 4;
  private int availableSalt = 10;

  public void prepare() {
    fryBacon();
    bakeEggs();
    availableSalt -= 1;
    System.out.println("Omelet is ready!");
  }
  private void bakeEggs() {
    availableEggs -= 2;
  }

  private void fryBacon() {
    availableBacon -= 1;
  }
}
```

All class methods are taken from the initial class AmericanBreakfast

A new class, `Omelet`, has been introduced. It's responsible for preparing the omelet, and all the methods and attributes needed to do so have been moved into it. `AmericanBreakfast` now has a reference to the `Omelet` class and invokes its `prepare()` method when needed (in the `prepareOmelet()` method). The rest of the code hasn't been modified. Running a new SonarQube analysis after this refactoring yields LCOM4 scores of 1 for both classes. "But wait," you might be saying, "Why doesn't this

still get a 2? The `Omelet` in `AmericanBreakfast` isn't connected to `availableOranges` or `availableSugar`." And of course, you're right—because references to other objects (such as `Omelet`) don't count. So `AmericanBreakfast` now only has one responsibility, to prepare the drinks.

Splitting a class is one way to address an inflated LCOM4 score, but LCOM4 isn't the only reason you might need to refactor. Even if a class's LCOM4 is perfect, its Response for Class (RFC) and the incoming/outgoing dependencies may still lead you to refactor.

6.3 RFC and couplings: classes with too many friends

We've looked so far at two types of complexity in a class: cyclomatic complexity and LCOM4. Now we'll turn to the complexity of a class's interactions with RFC and couplings. Only one of these two, RFC, has a dashboard widget available, so we'll start with it.

RFC measures the size of a class's response set, or how many distinct methods and constructors can be invoked by the class, including the class's own methods and constructors (or the default constructor if there's no explicit constructor for the class). RFC is another way of looking at complexity, but this time it's about the complexity of a class's interactions.

6.3.1 Response for Class

For any complexity-related metric, you want to keep the score low. That's a given. But why keep this particular one low? Because having a class with a high RFC means that when you need to work on it, you have to understand a lot more than just the code in front of you to make sure your modifications are correct and appropriate. Conversely, a lower RFC means your maintenance job is easier. Having a high RFC also makes a class harder to understand when you're considering reuse, harder to test, and less portable.

SonarQube presents RFC in a dashboard widget much like the one for LCOM4. We hope you added the RFC widget to the dashboard at the same time you added the LCOM4 widget. In that case, your dashboard now includes something like figure 6.8.

Again, you want to see the graph weighted to the left and the average RFC/class number low. How low? Well, there's no best answer for that, except "as low as possible." Acceptable RFC values vary from project to project, but in general, classes with an RFC of 40 or more need your attention. That's because a high RFC means a method call to

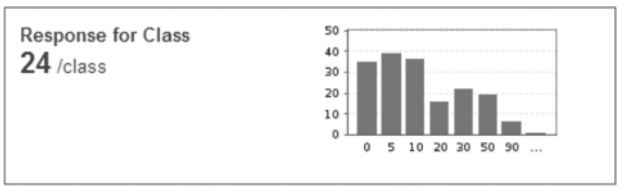


Figure 6.8 The RFC widget is one of the simplest, with only a distribution graph and an average score.

the class probably results in a large number of additional method calls either within the same class or to other classes (including superclass methods). In other words:

- It's hard to understand, debug, and maintain it.
- You need to read several extra classes to unit-test it.
- Simple unit-test cases will end up with several mocked objects.
- A lot of effort is needed for even simple modifications.

Because there's no perfect RFC score, you'll notice slightly different behavior in the drilldown for RFC. From the dashboard, click-through on the RFC metric or graph, and you land at a drilldown—standard behavior. But the drilldown itself isn't standard. Most drilldowns show only a subset of the files in a system: only the ones that need work. But the RFC drilldown shows them all, because there is no perfect score for RFC. Potentially, all the classes in a system need work. But aside from that minor point, the rest is pretty standard.

There's no special tab for this metric, so choose a file and you'll find yourself on the Source tab, as shown in figure 6.9. Once there, look at the numbers in the right column of the header, where you'll see several design metrics, including Response for Class.

There's not a lot else to see with regard to RFC. But before we move on to demonstrating the RFC calculation, we'd like to mention the other three metrics in this column: Classes, Number of Children, and Depth in Tree. They don't have a dashboard widget, but they're worth being aware of.

Classes reports on the total number of classes in the file. Typically it's one, but if there are nested classes, they're reflected here. Number of Children is how many classes inherit directly or indirectly from this class. Depth in Tree, sometimes called Depth of Inheritance Tree (DIT), is the converse: the level of inheritance from `java.lang.Object`. Because every class in Java inherits from `java.lang.Object`, 1—for something that extends `Object` directly—is the lowest number you'll ever see for this metric. These metrics aren't earth-shattering, but they're worth a look when you're considering the impacts of refactoring a class.

Now let's revisit the refactored `AmericanBreakfast` class and compute its RFC.

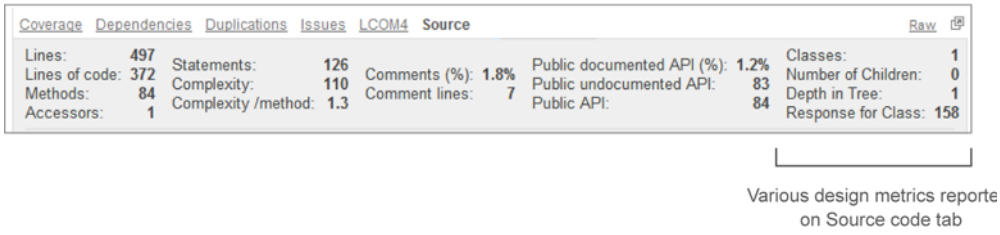


Figure 6.9 Various design metrics in the source code viewer tab

Listing 6.7 `AmericanBreakfast` class

```
public class AmericanBreakfast {                          <-- Hidden default constructor (+1)
  private Integer availableOranges = 10;
  private Integer availableSugar = 10;
  private Omelet omelet = new Omelet();                   <-- Constructor of other class (+1)

  public void prepareOmelet(){                            <-- Method of target class (+1)
    omelet.prepare();                                     <-- Method call of Omelet class (+1)
  }
  public void brewCoffee() {
    availableSugar -= 1;                                  <-- Method of target class (+1)
    System.out.println("Coffee is ready!");               <-- Method call of system.out (+1)
  }
  public void squeezeJuice() {                            <-- Method of target class (+1)
    availableOranges -= 2;
    availableSugar -= 1;
    System.out.println("Juice is ready!");                <-- println only counts the first time
  }
}
```

With three methods of its own, one member class, calls to the methods of two other classes, and the default constructor, `AmericanOmelet` has an RFC of 7. Notice that there are two instances in the code of `System.out.println()`, but the count is only incremented for it once, because each method is counted only once no matter how many times it's used. So the second instance of `System.out.println` in listing 6.4 doesn't count—literally!

What do you do if your RFC is too high? That depends on your situation and where the RFC hits come from. If it's high because your class has a lot of methods, you may also find that its LCOM4 is high, and refactoring to break out some of those responsibilities into other classes will naturally help your RFC. The same thing applies if it's high, because you're making lots of calls to many other classes. That's another case in which you may need to split the class into smaller, simpler classes with less going on. But if the RFC is high because you're making a lot of calls to just one or two other classes, then you may need to examine whether the things happening in this high-RFC class belong in those other classes.

You've seen so far that cyclomatic complexity, LCOM, and RFC metrics are key metrics for refactoring decisions, but you should also take a class's couplings into account. Next, we'll look at couplings in depth, what they are and how to calculate them.

6.3.2 Couplings

Incoming (*afferent*) and outgoing (*efferent*) couplings are the last two metrics to consider when looking at the cohesion and stability of a class. *Coupling* is the academic term for dependency in software engineering. It's used to describe the degree to which each component relies on or is relied on by other components.

These dependencies don't have their own widgets, so you can't get high-level information on them like you can with LCOM4 and RFC. As you'll see, they're only available at the file level via a dedicated tab in the file detail view, as shown in figure 6.10.

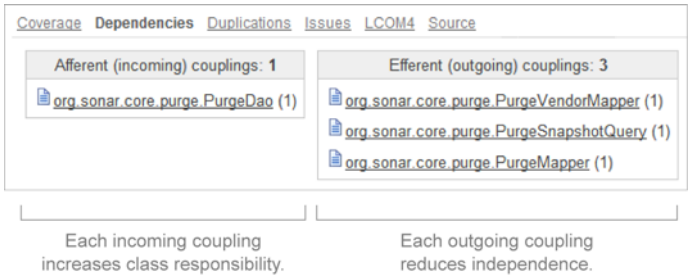


Figure 6.10 Afferent/efferent couplings of a class in the source code viewer

SonarQube shows couplings in two columns, with incoming couplings on the left and outgoing on the right. Notice that core Java classes (such as `java.io.File` and `java.math.BigDecimal`) don't count for these metrics.

The afferent coupling (incoming dependency) count represents the number of other classes that depend on the target class. High scores mean the target class is playing an important role in the module/system, and it's an indication of the importance of the tasks for which this class is responsible. High afferent coupling means a change to the target class will affect many other classes in the system (those that depend on it). So, changes to the class are riskier because there's a higher chance of introducing new bugs in the calling classes, or even breaking an integration.

The efferent coupling (outgoing dependency count) is the number of classes that *this* class depends on. High values mean this class uses a lot of other classes, which could mean it's brittle or unfocused, and definitely introduces the risk that every change to those other classes could affect this class's behavior, usually negatively.

Like RFC, couplings are a reflection of the connectedness of a system. Even though couplings are a file-level metric, you can look at them as a reflection of the complexity of the whole package or module. Because complexity in general isn't a good thing, you want to keep couplings low. The most popular way to reduce coupling is called *decomposition*. Extract smaller, more focused classes from the original, highly coupled classes so that the responsibilities of the initial class are spread among several simpler, easier-to-maintain classes with lower coupling.

Manually calculating the outgoing dependencies of a class is simple. Just count the class-level or method-level references to external classes.

Listing 6.8 `AmericanBreakfast` showing outgoing couplings

```
public class AmericanBreakfast {
private Omelet omelet = new Omelet();
private Coffee coffee = new Coffee();

  public void prepareOmelet(){
    omelet.prepare();
  }

  public void prepareDrinks() {
    coffee.brew();
    Juice orangeJuice = new Juice();
```

Dependency to Omelet class

Dependency to Coffee class

Dependency to Juice class

```
    juice.squeeze();
  }
}                          <-- Total number of efferent couplings: 3
```

In listing 6.8, the `AmericanBreakfast` class has been modified slightly to better demonstrate how to find efferent couplings. There are now class-level dependencies to the `Omelet` and `Coffee` classes, and one method-level reference to the `Juice` class, for a total of three outgoing dependencies.

Trying to compute the incoming dependencies is a lot harder, because you would need to search all your code to find references to a given class. In large systems, and even in smaller ones, it would be time-consuming and error-prone or almost impossible to get those numbers on your own.

This concludes our tour of design metrics. As you've seen, SonarQube provides quite a few, some more important than others, but all worth at least a glance when you're considering whether and how to refactor a class.

6.4 Summary

This chapter focused on file-level metrics related to design and complexity. Some of the concepts are considered hard to understand. But even if they're top-of-mind for you, they're still impractical to calculate by hand for an entire code base. Fortunately, SonarQube does the tedious bits for you and lays the numbers at your feet. In this chapter, you saw that

- High cyclomatic complexity makes your source code hard to understand and maintain. We've shown how SonarQube computes cyclomatic complexity and how to refactor for lower complexity.
- SonarQube's LCOM4 presentation helps you find classes that do too much and shows you, at a file level, how a class's responsibilities are grouped so you can make good refactoring decisions.
- RFC and the count of a class's couplings help you understand a file's interactions in your system. Social butterfly classes with a high RFC are invoking a lot of methods, either internally or externally, and may need to be reined in.

In this chapter, we looked at complexity at a file level. Before we move to part 2 of the book, we'll talk about complexity at a package and system level and show you what SonarQube has to offer with regard to the architectural design of your system.

7 Improving your application design

This chapter covers

- Layering your code
- Discovering dependencies and eliminating cycles
- Library management for Mavenites
- Defining your architectural rule set

The size of an active project increases commit by commit, day by day, week by week. As your software grows, it gets more complex. New classes are added, methods are improved, and new libraries are created. Gradually, the structure that was clearly delineated and pristinely designed at the start turns into a morass. The first sign of trouble you usually notice is the mess of external libraries needed in order to compile and run an application in a production-like environment, but that's just the tip of the iceberg.

In chapter 6, we looked at complexity at a file level. In this chapter, we'll zoom out to look at complexity at the package, module, and application levels, and discuss how to use SonarQube to keep your architecture in good *modular* shape. We'll look at package and module-level dependencies and show you how to read the

Dependency Structure Matrix (DSM) to find *suspicious dependencies* and *suspicious cycles* among software components. We'll talk about why they're bad and how you can remove them.

You want to keep your design clean and your system maintainable and extensible. SonarQube provides an extremely useful and powerful mechanism called the Dependency Structure Matrix. With the DSM, you can visualize the dependencies between software components (libraries, packages, or even source code files) and then find those cycles that make your system's design too complicated.

This chapter will also show you how painless SonarQube makes it to examine a Maven-centric project's external library dependencies, their versions, and whether there are conflicts that need your attention.

Finally, we'll show you how to create architectural rules to monitor access between certain files or packages. You won't be able to keep developers from accessing files or packages in ways that run counter to your overall design; but you can have SonarQube tell on them by creating issues based on the rules you define, allowing you to catch problems and correct them early.

Before we move on, we'd like to explain something important: the analysis done by SonarQube, although powerful, is performed without understanding what packages do. SonarQube computes the minimum number of links to break to remove the cycles, but this is a theoretical approach. What you need to do might be completely different.

7.1 Layering your code

Sam, after surviving a couple of rough assignments, now has a new challenge. It's time to improve an existing feature. She needs to implement recurring credit card billing for a new Coin of the Month subscription that the coin store plans to add. Basic credit card processing is already in place, but the current checkout process only supports one-time payment.

She gets to work, but after a couple of days Sam sees that it's harder than she realized, because the code that handles payment processing is too complicated. Several classes participate to complete the checkout, and there is no *orchestration class*—a class that coordinates other classes to perform a single task. Each class has several dependencies (which could be potentially reduced), and changing a single method affects many components. She worries that every little change could break the whole system.

Sam's having a difficult time, and it's made worse by the fact that navigating to several source code files just to see how the system works is time-consuming and potentially error-prone. A modular system with concrete responsibilities among packages would be less painful to understand and maintain.

7.1.1 Looking at dashboard widgets

The first step in improving the design of your application is to find (and fix) dependency cycles between your components. But first, why are cycles bad? To get a feel for

Figure 7.1 SonarQube dashboard widget pertaining to cycles and dependencies

the effect of cycles, imagine a small application that's split into three libraries. Over the course of time, each library has evolved to depend on the other two (thus creating cycles!), until finally you're at the point that you can't upgrade one without also touching the others. Because of dependency cycles, simple changes that should be easy take three times as long (at least) to complete.

Detecting this sort of unhealthy circular dependency at the package level is the focus of the package design widget, shown in figure 7.1. The widget is split into two parts. On the left side you see a metric named Package Tangle Index and the number of package cycles detected during analysis.

On the right, the widget reports on unwanted dependencies between packages and files. These are the dependencies you should try to cut in order to remove package cycles. For the project shown in figure 7.1, you would need to fix 35 package dependencies involving 53 files to clean up your package dependency cycles.

All the numbers in this widget are golf-style metrics, and 0 is always your target score. Before you start clicking widget numbers (yes, there is a drilldown view, although it's different from what you've seen before), let's clarify how SonarQube computes these metrics.

7.1.2 Understanding cycles and unwanted dependencies

In chapter 6, you saw that SonarQube inventories each file's afferent and efferent couplings (incoming references from and outgoing references to other files). At the end of an analysis, it has a full list of everything a given resource (file or package) uses and everything that uses it. Each time a file or package is on both sides of that list, it's considered a cycle, and the usage on one side of that list will be shown as a dependency to cut. Figure 7.2 illustrates a set of package dependencies.

Package `A` depends on package `B` and package `C`. Package `C` depends only on package `D`. So far, so good. It's a common dependency graph. But add in the fact that package `D` depends on package `A`, and you've got a dependency cycle: `A > C > D > A`.

Now imagine that packages `A`, `C`, and `D` (the ones in the cycle) contain only one class each, like the simple classes in listings 7.1, 7.2, and 7.3.

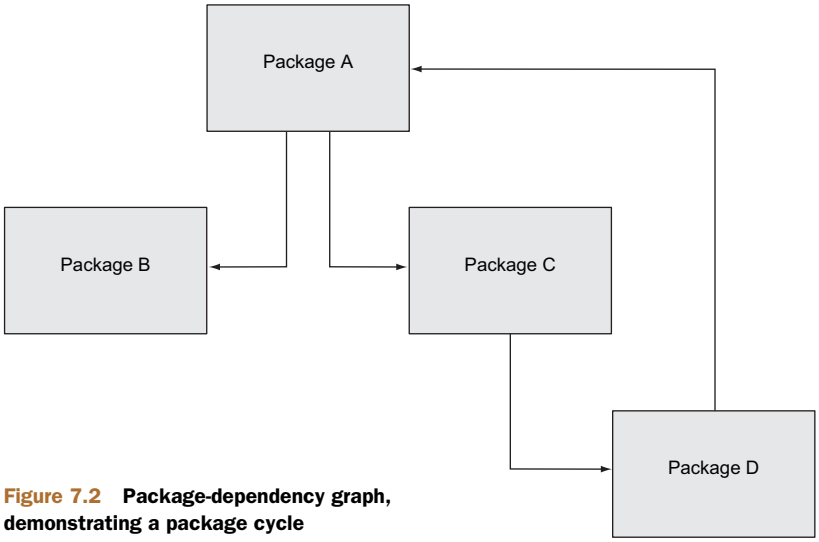


Figure 7.2 Package-dependency graph, demonstrating a package cycle

Listing 7.1 Simple `ClassA` of package `A`, referencing `ClassC` of package `C`

```
package org.manning.sonarinaction.packagea;
import org.manning.sonarinaction.packagec.ClassC;
public class ClassA
{
  private ClassC classC = new ClassC();
  public void doSomething(){
    System.out.println ( "doSomething");
  }
  public void doSomethingBasedOnClassB(){
    System.out.println (classB.toString());
  }
}
```

Dependency on ClassC of package C

Listing 7.2 shows the simple class `ClassC` that references `ClassD`.

Listing 7.2 Simple `ClassC` of package `C`, referencing `ClassD` of package `D`

```
package org.manning.sonarinaction.packagec;
import org.manning.sonarinaction.packaged.ClassD;
public class ClassC
{
  private ClassD classD = new ClassD();
  public void doSomethingBasedOneClassD(){
    System.out.println (classD.toString());
  }
  public String toString(){
    return "classC";
  }
}
```

Dependency on ClassD of package D

Listing 7.3 shows the class `ClassD` that references `ClassA` and produces the cycle.

Listing 7.3 Simple `ClassD` of package `D`, referencing `ClassA` of package `A`

```
package org.manning.sonarinaction.packaged;
import org.manning.sonarinaction.packagea.ClassA;
public class ClassD
{
  private ClassA classA = new ClassA();
  public void doSomethingBasedOneClassA(){
    System.out.println (classA.toString());
  }
  public String toString(){
    return "classD";
  }
}
```

Dependency on ClassA of package A

After running a SonarQube analysis on these three classes, you would get a Package Tangle Index value of 66.67%. The other three metrics in the widget would each be 1, meaning that

- Two out of three packages (66.67%) participate in dependency cycles.
- At least one file cycle is detected inside a package.
- One dependency needs to be cut on a file level.
- One dependency needs to be cut on a package level.

You've probably already figured out that `ClassD`'s dependency on `ClassA` is the one that closed the loop to create a package cycle (`A > C > D > A`), and that's the dependency we'd target to cut the cycle. Many times, the cycles you see will be *directly reciprocal*: that is, package `A` including package `B` and vice versa. We intentionally structured this example to involve three packages, to make the point that cycles aren't found only between pairs of packages, but can involve several packages at once.

Now that you know a little more about dependency cycles, let's move on to SonarQube's drilldown. As you'll see, it's different from the drilldowns you're used to.

7.1.3 Moving from project to package level

To access SonarQube's drilldown, click-through on any number in the package design widget. Figure 7.3 shows the result of clicking Package Tangle Index.

The top half of this drilldown is somewhat familiar. The metric you click is at upper left, followed by a list of modules sorted worst first—although that appear only if you're in a multimodule project. If you're in a single-module project, the list is eliminated and you only see the table shown at the bottom of figure 7.3. This presentation is like the others you've seen in that components with no design issues are omitted from the list.

Just below the library list in figure 7.3 is the Design tab with the DSM. The DSM is a powerful tool for exploring your project's package interactions. Regardless of which metric you click in the dashboard to get to the drilldown, the numbers in this matrix are always package dependency counts.

Figure 7.3 Drilldown view for design metrics

The first time you access this drilldown from the dashboard widget, the DSM displays all the packages in the project (see figure 7.4). Click any row in the module list at the top of the page, and the matrix is filtered to display only the packages in the selected module. You can even deselect a library by re-clicking its row to have the DSM return to showing all the packages in the project.

An alternate way to navigate to the DSM is to click the Design link on the menu at left at the project level. Regardless of whether you're in a multimodule project, you don't see the upper panel if you go this route (as illustrated in figure 7.4 with the XStream project's modules, and packages from one of those modules).

Now that you're thoroughly versed in how to get to the DSM, let's take a deeper look at what it shows.

7.2 Discovering dependencies and eliminating cycles

As we've said, the DSM is a flexible, colorful, and easy-to-use tool for browsing dependencies at every level of your projects. Although it's powerful, using it requires some knowledge; so next we'll look at how to decode what it's telling you. You'll see how to read the DSM to spot dependencies between components. But be aware that the DSM relies heavily on colors to convey its content. We'll do our best to point out where data is being washed out of our black-and-white screenshots, but your best bet is to follow along in your own SonarQube instance as we walk you through it.

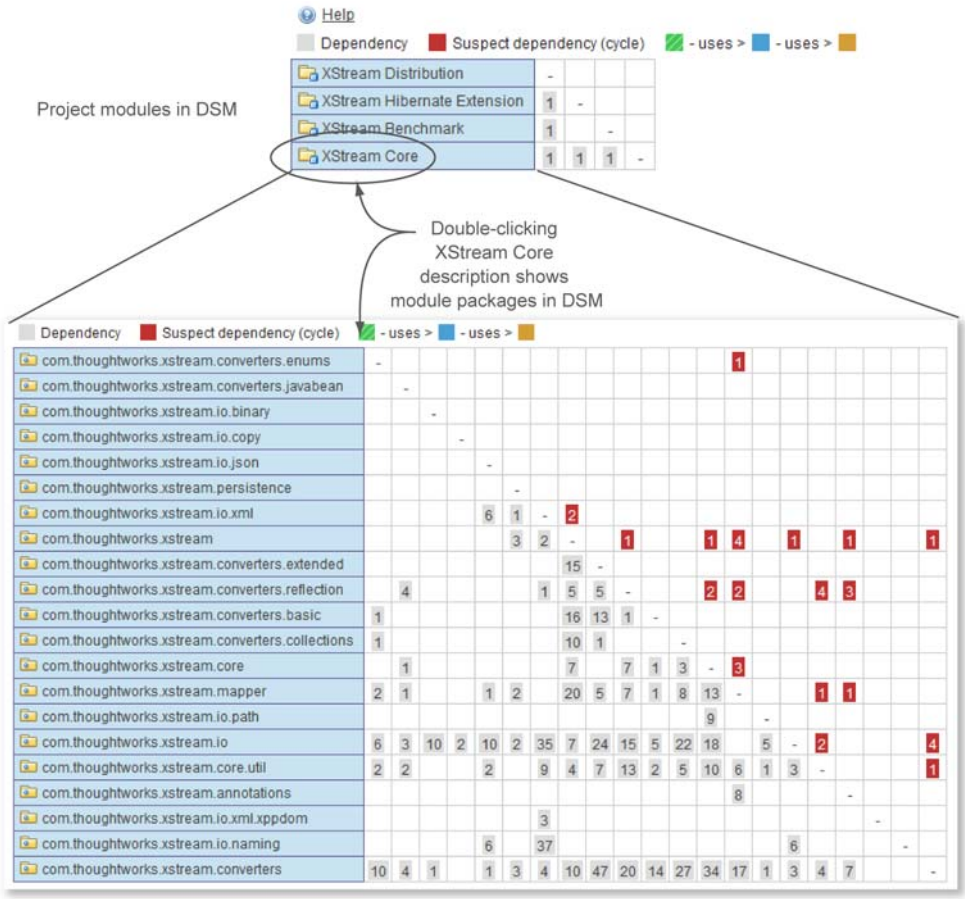


Figure 7.4 **DSM view for library and package dependencies**

NOTE Before we move on, we'd like to remind you that it's not always possible to fix cycles. For example, when the cycles involve an API, fixing them is likely to involve a retro-compatibility issue.

7.2.1 Navigating the Dependency Structure Matrix

When you look at the DSM for the first time, it's normal to find it difficult to read. Fortunately, it's one of those things that most people get used to pretty quickly. You've seen a couple of screenshots of the DSM, and you know it's considered a matrix.

You may be wondering why only one axis has labels. The column labels are omitted to save space and to simplify the interface, because they would be the same as the row labels: the same text in the same order, as shown in figure 7.5.

We've manually added the column headers in figure 7.5 to help you better visualize the information in the graph. Unfortunately, unless you turn your head to the side, it's hard to read the headers in the figure. Now imagine how much more difficult it would be in a very large program. We've seen DSMs that scrolled down multiple pages

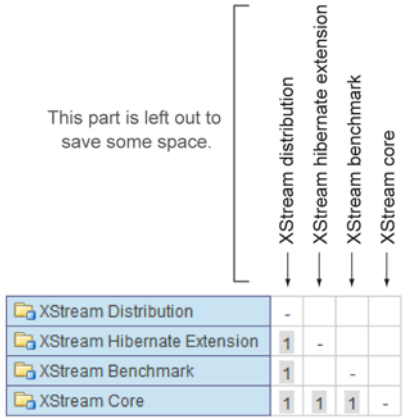


Figure 7.5 Column headers would make the DSM look busy.

(and, because the DSM is always square, they scrolled sideways just as far). Imagine how much noise duplicate header labels would create.

Before we get into the DSM's functionality, we'll explain what its colors tell you. We'll start with the colored cells that contain numbers, as shown in figure 7.6.

Some of the color cues that apply to the rest of the SonarQube interface work here as well: red is bad, and gray is neutral. A black number in a gray cell is the count of references between two components. White numbers on a red background alert you to a dependency cycle that could (should?) be removed. Unfortunately, because this book is printed in black and white, you're seeing *all* the cells as gray. The good news is that the colors of the numbers themselves will help you differentiate: the numbers in cells that report on the dependencies between the components are written in black for innocuous dependencies or white for cycle dependencies.

The numbers represent dependencies. When the DSM lists modules in the left row header, the numbers show package dependencies. When the left row header shows packages, the DSM numbers show file dependencies.

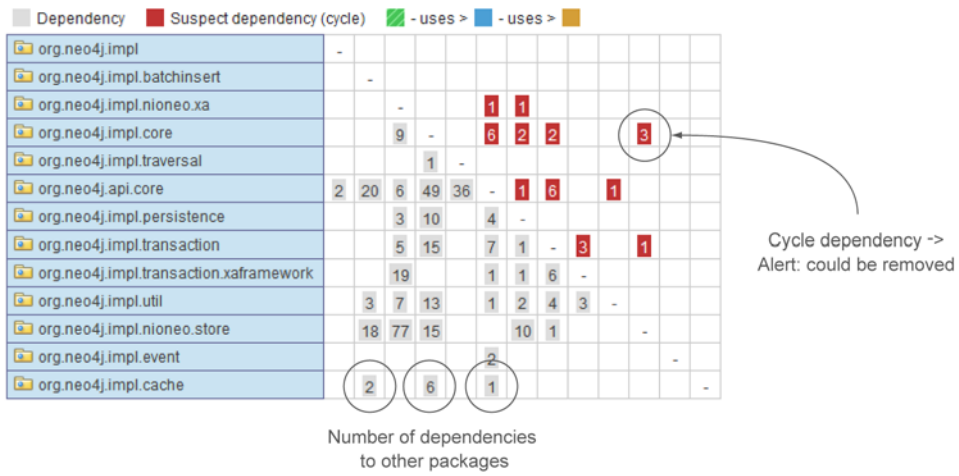


Figure 7.6 Colored cells in the DSM report on dependencies and cycles between components.

Most of the figures and examples that you'll see in the next few pages focus on package dependencies. Based on our experience, these dependencies are the most important when we talk about software design; potential package cycles should be carefully examined and removed.

Next we'll look at how to read the DSM and how to use it to identify cycles that may need to be removed to keep your application's modularity and maintainability in good shape.

7.2.2 How the DSM works

To examine a package's dependencies, click the package in the left column. The package's row and its corresponding column are highlighted in pale blue. The numbers in the package row are the package's afferent, or incoming, dependencies—the other packages that this one *includes*. Conversely, the numbers in the highlighted column are the package's efferent, or outgoing, dependencies—the packages that this one is *included by*. Figure 7.7 shows this in action; it's a screenshot of the DSM for the `Server` module from SonarQube.

As you see, the package `org.sonar.server.charts` was clicked, and its description is highlighted in blue. All cells for its corresponding row and column are also highlighted light blue.

TIP Remember RICO to know what the rows and columns in the DSM are telling you: Row-Incoming, Column-Outgoing.

Let's do some simple math with the DSM. Assume that you need to find the total incoming and outgoing dependencies of the selected package. All you have to do is

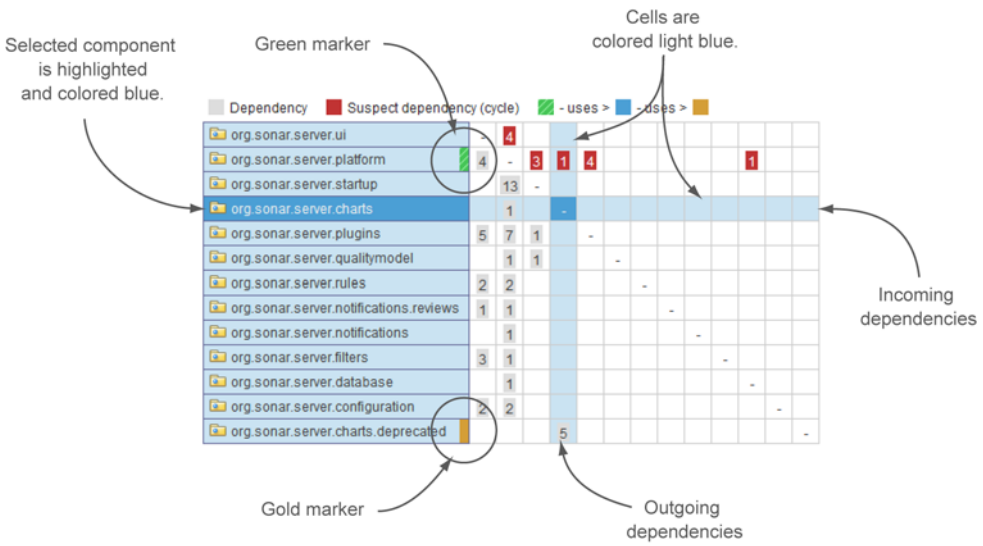


Figure 7.7 Viewing dependencies of selected component in the DSM

sum up the numbers shown in the highlighted row and column, and you're done. For the record, `org.sonar.server.charts` has only one incoming file dependency (the sum of the numbers shown in the highlighted row) and a total of six outgoing file dependencies (the sum of the numbers shown in the highlighted column).

You may have noticed that some rows (other than the selected one) have a colored marker (green or gold) at the end of the package description. These colors are your guide to finding out which package is dependent on which other one. They let you read the dependencies as a sentence. The legend at the top of the DSM explains the meaning of each color. Green uses (depends on) blue, and blue depends on gold.

You can read the dependencies of the highlighted package in figure 7.7 just as you're reading this book. The `org.sonar.server.platform` package (green marker) has one file dependency on `org.sonar.server.charts` (blue highlighted package), which has five file dependencies on the `org.sonar.server.charts.deprecated` package (gold marker).

As we mentioned, each number represents the total file dependencies (incoming or outgoing) from one package to another. The fact that there are no column headers in the DSM might make you think, especially if you're a SonarQube starter, that it'll be hard to find the dependencies between two specific components. That's where the beauty of the DSM comes in.

So far, you've seen that you can click any row label in the DSM. Now it's time to try clicking a cell. When you do, you'll notice a couple of changes in the DSM. Figure 7.8 shows the DSM after we clicked the fourth cell in the last row: the one containing 5. First, notice that more rows and columns are highlighted than before: *two* package names are highlighted in blue instead of just one.

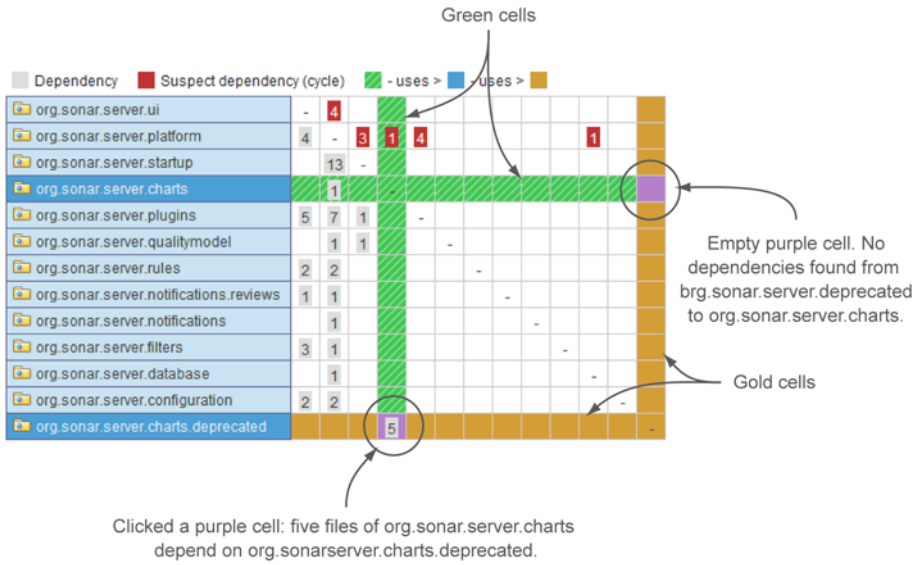


Figure 7.8 Clicking a DSM cell. Dependencies between selected components are highlighted and differently colored in both directions. The clicked cell also turns purple.

That's because when you click any single cell in the DSM, you're picking half of an interaction: where package `A` meets package `B` in the grid. To show the full relationship, SonarQube lights up the entire rectangle for you, also showing where package `B` meets package `A`.

In addition to highlighting the row you clicked, the DSM also highlights the package that belongs to the clicked column. For instance, in figure 7.8, we clicked the fourth cell (column) in the row, so the corresponding package (the fourth row) is also highlighted.

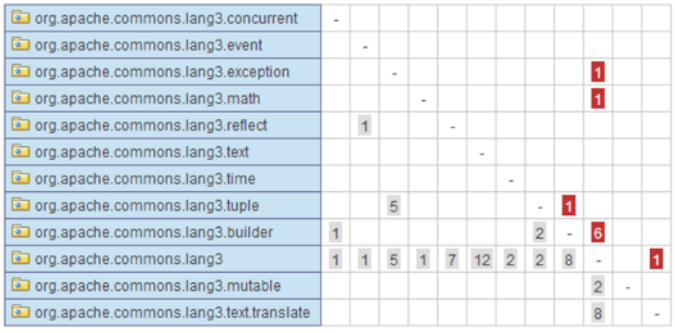
If you're following along on a live instance of SonarQube, you've also noticed that the highlight colors have changed. Cells in the grid aren't light blue anymore: they're either green or gold. The new colors indicate the direction of the dependency, similar to the package name markers we showed you in the previous figure. And the same rules apply: green depends on blue, and blue depends on gold.

You'll also have noticed (if you're following along in a browser) that the two corner cells where green and gold meet (lower-left and upper-right) are colored purple. The numbers in each corner reflect the dependencies in one direction of the relationship. In figure 7.8, the empty purple square at upper right is where `org.sonar.server.charts` is included by `org.sonar.server.charts.deprecated`. That square is empty, so you know there are zero dependencies. The corresponding purple square with a 5 at lower left shows the other half of that relationship, where `org.sonar.server.charts.deprecated` is included by `org.sonar.server.charts` five times.

The last thing to talk about before we move on to hunting package cycles is how the DSM sorts packages. High-level components are listed at the top of the DSM, and low-level components are listed at the bottom. To better understand what makes a package high-level or low-level, look at figure 7.9.

The `concurrent` and `event` packages are considered high-level packages because they have no incoming dependencies. Thus they appear at the top of the list, subsorted alphabetically. On the other hand, several packages depend on the `lang3` package, so

Concurrent and event packages are high-level components because they have no incoming dependencies from other packages.



Many components depend on the lang3 package so it's considered a low-level component. But it's not placed last on the list because it has outgoing dependencies to the mutable and text.translate packages.

Figure 7.9 DSM sorting. High-level components appear at the top of the list.

it's listed near the bottom—but it's not last. Why? The `lang3` package has outgoing dependencies on the `mutable` and `text.translate` packages, so those two packages are even lower-level and are listed after `lang3`.

Now that you've got a handle on the DSM's basic navigation, let's move on to hunting down package cycles to improve the design of your application.

7.2.3 Identifying cycles

You may already be suspicious about the design of the project we've been showing you because there are packages that depend on packages that depend in turn on the first packages. (Dizzy yet?) You're right to be suspicious. These are what we call *undesired dependencies*. You've already seen that some cells in the DSM are red: you can get a quick take on how bad your design is with a glance at the number of red cells, and an even better understanding by looking at the size of the numbers in them. Yes, big is bad in this case. In this section, we'll dive in to what the DSM has to tell you about these cycles.

First, remember that even though the column headers aren't shown in the DSM, you know what they are—they're the same as the row headers. If the first column in figure 7.10 were labeled, it would say `org.sonar.server.ui` to match the first row. The second column would say `org.sonar.server.platform` to match the second row, and so on.

The diagonal line across the grid shows where each package matches up horizontally and vertically. These cells never contain a number because SonarQube doesn't report a package's dependencies on itself at this level.

If you visually split the DSM into two triangles by drawing a line across all the dashes, as we've done in figure 7.10, then all dependencies that circle back to form cycles are shown in the upper-right triangle.

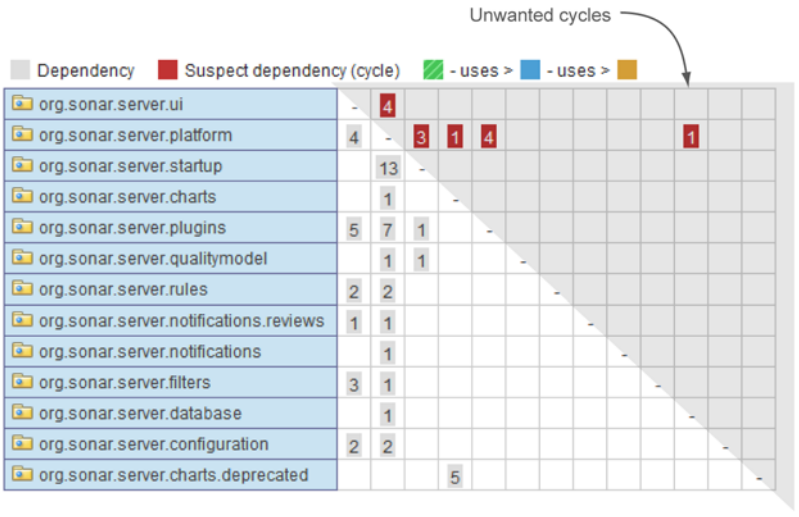


Figure 7.10 Unwanted cycles are displayed in the upper-right triangle.

But why do the numbers showing unwanted dependencies all end up in the upper-right triangle? Is it this way all the time? Well, yes. This is the norm because SonarQube lists software components so dependencies to cut are always shown in the upper-right triangle. You've seen that SonarQube sorts higher-level packages to the top of the DSM. In general, you'll see that SonarQube marks a higher-level package's *incoming* dependencies as the ones to be cut.

We've told you that the numbers on red backgrounds in that upper-right triangle in the DSM represent dependencies to cut in order to remove cycles. But knowing that you have a total of *N* cycles in your project or module is completely useless, because a single bad dependency can generate a lot of different cycles. The most important thing is to quickly know which dependencies you need to cut in order to fully remove unwanted cycles.

Fortunately, when you're looking at a package-level DSM, double-clicking a cell gives you a list of file dependencies. When you click a red cell, the list represents the file dependencies that should be removed to eliminate a package cycle, as shown in figure 7.11.

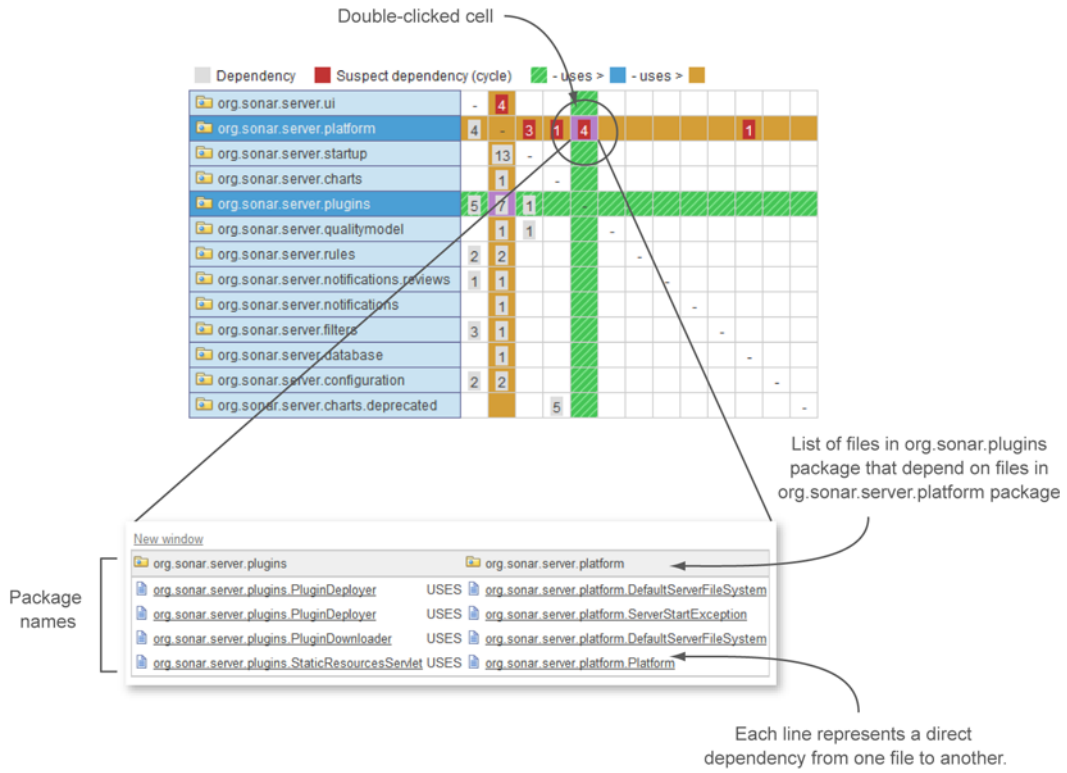


Figure 7.11 Double-click a numbered cell, and you see a list of outgoing file dependencies between the selected packages. Clicking a file name pops up a window with the file detail view you've seen before.

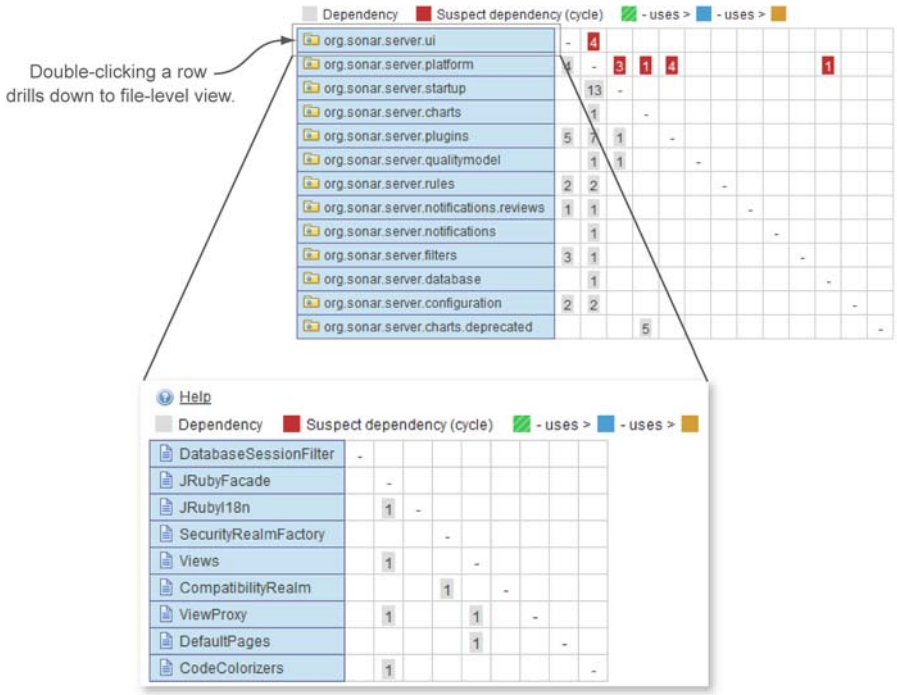


Figure 7.12 Double-clicking a row drills down the DSM to the next navigation level.

Before we move away from the DSM, there is one last thing to mention. You've seen that double-clicking a cell drills down to display a list of individual couplings. You can also double-click any row label to drill down within that package. Figure 7.12 shows the result of that kind of drilldown: the file dependencies of a specific file.

Figure 7.12 doesn't show any file-level dependency cycles, but they're possible, and they would appear here in a case where two files directly depended on each other. Most of the DSM behaviors we've discussed apply here as well, except for a slight difference in double-click behavior. Double-clicking a row pops up a new window showing the source code viewer instead of drilling down (because there's no down left to drill in to), and double-clicking a numbered cell shows the names of the connected files instead of showing a list of associated files.

Now that you've identified all these cycles between packages and/or files, you may be wondering how to get rid of them. The answer depends on the design of your system, but there is a general starting place. Most cycles appear because the packages or classes involved don't have clear and concrete responsibilities. This could be a matter of original design, of gradual design distortion, or lack of communication within the team. Whatever the cause, to resolve the issue, you need to find those mixed responsibilities and clean them up by doing one of the following:

- Moving responsibilities into higher-level classes
- Creating new classes

- Merging classes
- Reorganizing your packages

You've seen what SonarQube has to tell you about design and complexity among the packages and modules in your application. Next we're going to move back a step to look at the complexity of an application's external dependencies: its libraries.

7.2.4 Library management for Mavenites

The time when software systems were built from just a few source code files has long since passed. Today, applications make extended use of common libraries to avoid reinventing the wheel, improve productivity, and increase modularity and maintainability. Theoretically, whenever a new library version is available, the old one is replaced, and the system takes advantage of new features and resolved issues.

This section discusses features that are available only for Java projects built with Maven. We assume a familiarity with basic Maven concepts. (Sorry, non-Java and non-Maven folks!)

In general, Maven has dramatically decreased the time developers spend on library management. It lets you define which libraries your source code depends on, and it takes care of downloading those libraries and their dependencies from third-party repositories. The idea is that Maven takes care of the tedious stuff, and your life as a programmer gets a lot easier; and that's almost true.

When it works, it works beautifully. But sometimes your library dependencies have second levels that conflict with each other. We've seen several problems caused by jars that were silently downloaded by Maven and that were incompatible with other libraries in the dependency tree.

If you've ever been in the position of untangling one of these knots, you know it's a painful task. But the next time you find yourself in this situation, SonarQube can make your life easier. It lets you browse library dependencies at the project or module level and offers a dynamic search to spot libraries of interest. It also gives you a separate search function to find which libraries depend in turn on a specific library.

7.2.5 Browsing the library-dependency tree

To start browsing your library dependencies, click the Libraries link on the left menu at the project level. The link appears even if you're watching a non-Maven project, but clicking it won't display any results. As figure 7.13 shows, you'll get a library-dependency tree including both direct and indirect dependencies. This is the result of running the command `mvn dependency:tree`.

Each top-level tree node represents an outgoing, first-level dependency—the one that was explicitly stated in your pom.xml file. To the left of each library's name is an icon representing the status of the library in the current SonarQube installation. Table 7.1 lists the possible icons and their meanings.

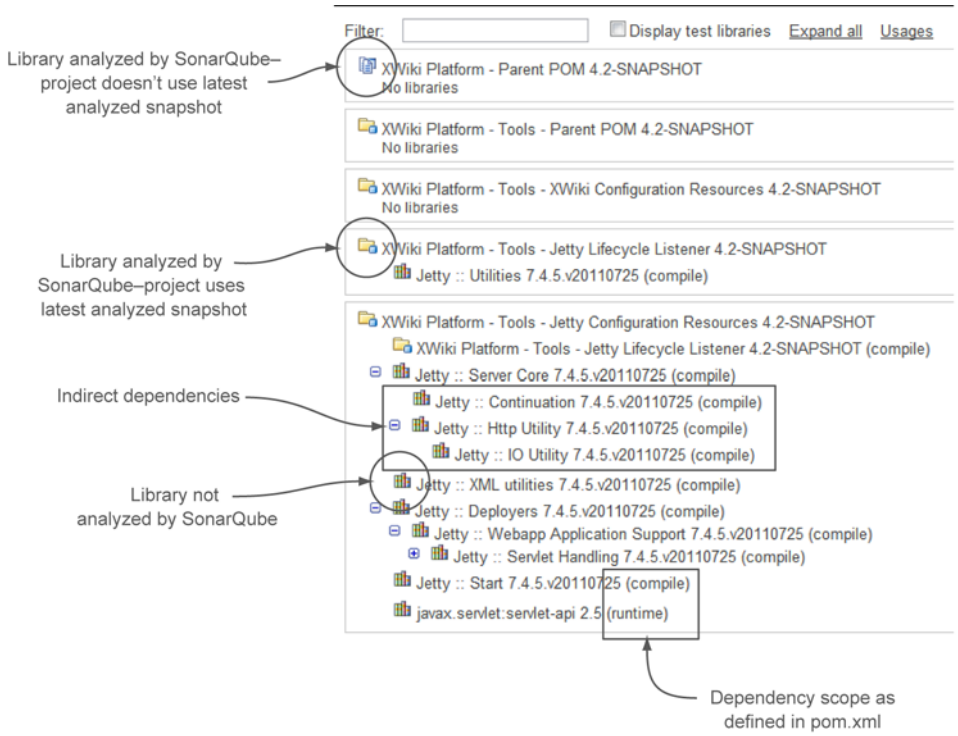


Figure 7.13 **Library-dependency tree for Java Maven projects**

Table 7.1 **Library statuses**

Icon	Explanation
	The library has been analyzed by the current SonarQube installation, and the project/library uses the latest analyzed snapshot.
	The library has been analyzed by the current SonarQube installation, but the project/library uses an older analyzed snapshot.
	The library hasn't been analyzed by the current SonarQube installation.

To the right of each library name is its dependency scope as defined in pom.xml (compile, provided, runtime, or test). By default, only development libraries are shown in the tree; but if you select the Display Test Libraries check box, the tree is re-rendered to include libraries used for testing.

Second-level tree nodes represent indirect dependencies. Those are libraries used by your project's first-level dependencies, the ones specified in the pom.xml files.

Including these second- and third-level (and so on) dependencies can sometimes result in a huge dependency tree. But huge or not, if everything works well, you probably don't care. Unfortunately, that's not always the case. Sometimes you end up with branches of your tree that conflict with each other, such as one branch that specifies version 1 of a library and another that wants version 2. This sad state is known as *Maven dependency hell*, and as we've mentioned it can cause problems (what else would you expect from Hell?). But unlike Dante's Hell, there is no "Abandon hope, all ye who enter here" sign. There is a way out, and SonarQube is your GPS.

To start, imagine that you want to find all project dependencies (direct or indirect) on Apache's `commons-lang` library. You can use the dynamic filter at the top of the dependency tree. Just type the name of the library you want to check, and the list is automatically filtered accordingly. An example is shown in figure 7.14.

With a few clicks, we found that three modules in the same project depend directly or indirectly on `commons-lang`. Unfortunately, they all depend on different versions, which could (and probably will!) cause stability and compatibility issues at runtime, because they'll load three different (most likely incompatible) versions of the same library into their shared JVM. Which one will be used by your libraries? How can you be sure your system won't use an older library version than the one it needs (the one that has the new API you call from your code)? Basically, you can't know. And you can't do much at runtime to control which version is used. You need to deal with the issue earlier in the development lifecycle.

Your goal is to use only the latest stable release of `commons-lang`—version 2.6. To get rid of the other versions, you need to make some pom.xml changes. First, change all direct dependencies to the desired versions by setting the right attribute in pom.xml (if you haven't already). If your target library isn't already a direct dependency, then make it one, being sure to specify the correct version. Then, exclude the indirect dependencies to avoid future references to the unwanted versions. That's it. The project now uses a single, correct version of the library.

Listing 7.4 shows the starting point. You include version 2.6 of `commons-lang` explicitly, along with version 1.6 of `commons-configuration`. Unfortunately, `commons-`

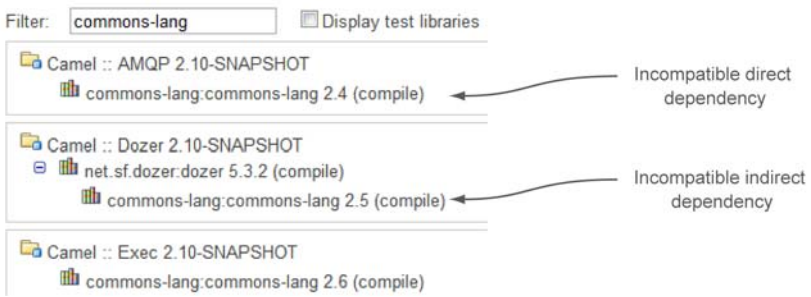


Figure 7.14 Dynamic library filtering

configuration depends in turn on an older version of commons-lang. Because both dependencies have a compile scope, both versions of the commons-lang library will end up in the final WAR, which could cause problems at runtime, as mentioned earlier.

Listing 7.4 Sample dependencies section of a pom.xml file (commons-lang)

```
<dependencies>
    <dependency>
        <artifactId>commons-lang</artifactId>
        <groupId>commons-lang</groupId>
        <type>jar</type>
        <scope>compile</scope>
        <version>2.6</version>
    </dependency>
    <dependency>
        <artifactId>commons-configuration</artifactId>
        <groupId>commons-configuration</groupId>
        <type>jar</type>
        <version>1.6</version>
        <scope>compile</scope>
    </dependency>
</dependencies>
```

Version you want

Depends on wrong commons-lang version

To overcome the issue, you need to add an exclusion to the commons-configuration dependency to tell Maven to leave out the extra copy of commons-lang. The next listing shows the changed pom.xml snippet.

Listing 7.5 Revised dependencies section of a pom.xml file

```
<dependencies>
    <dependency>
        <artifactId>commons-lang</artifactId>
        <groupId>commons-lang</groupId>
        <type>jar</type>
        <scope>compile</scope>
        <version>2.6</version>
    </dependency>
    <dependency>
      <artifactId>commons-configuration</artifactId>
      <groupId>commons-configuration</groupId>
      <type>jar</type>
      <version>1.6</version>
      <exclusions>
        <exclusion>
          <artifactId>commons-lang</artifactId>
          <groupId>commons-lang</groupId>
        </exclusion>
      </exclusions>
    </dependency>
```

Version you want

Excludes the version you don't want

Now that you know how to navigate out of dependency hell, we'll move on to SonarQube's library-search function.

7.2.6 Who uses this library

Libraries have been a boon to software development, but that doesn't mean they're trouble-free. Unfortunately, many things can go wrong. Consider the following example. Tom works for a Java software company. Most of the applications developed in the company use the Apache `commons-io` library for file handling. Once in a while, each project team checks for a new version of the library. They review the release notes and then decide whether to upgrade their project to the newest version.

Suppose that six months after the latest release of `commons-io`, a security issue is discovered. Because there's no standard process for library upgrades—when and if libraries are upgraded is up to each team—nobody has a global view of which projects and which libraries currently use the problem version.

A manual search would be tedious, time-consuming, and error-prone. Fortunately, SonarQube makes it easy with not one but *two* ways to search for library dependencies. It's worth mentioning that this functionality is unique: you can't find anything similar in other software quality tools. From the home page, you can click the Dependencies link in the left rail and then enter the name of the library in the text box. Or, from the project Libraries page, you can click the Usages link. Both entry points are shown in figure 7.15.

As figure 7.16 shows, the dependencies view is a comprehensive screen divided into three panels. First is a list of all libraries matching the search criteria. Click a row here, and the middle panel lists all the versions found as dependencies in SonarQube. Click a version, and the last panel is filtered to show the projects/modules that use the selected version. You're done. With a search and three clicks, you can find all the uses of a specific library version.

Next, we'll move away from library dependencies but stick to the package interactions theme by discussing the ability SonarQube gives you to add governance around the interactions within a system.

Figure 7.15 **You can reach the dependencies view via two entry points in SonarQube.**

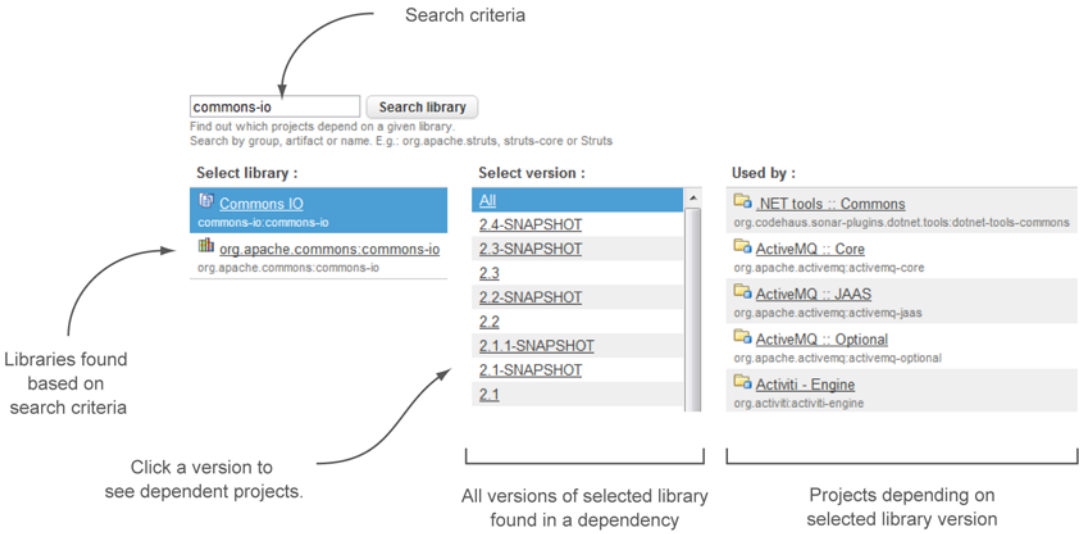


Figure 7.16 Finding library dependencies

7.3 Defining your architectural rule set

The majority of modern enterprise systems are based on *n*-tiered architectures. This allows developers to build software in logical layers. Each layer should only interact with the next lower layer via a predefined interface. Internal properties should also be hidden from other layers. These architectures borrow basic principles and ideas from the Open Systems Interconnection (OSI) model (http://en.wikipedia.org/wiki/OSI_model), which targets communication systems.

You Java folks who wrote code in Java EE, Spring, or Struts a decade ago are probably familiar with the similar Model-View-Controller (MVC) design pattern for building an application in distinct tiers.

The Model-View-Controller pattern

Model-View-Controller (MVC) is a software design pattern that lets you separate the user interface from the representation of data. The model consists of application data and business rules. The controller is the go-between: it's the part responsible for translating data and commands between the model and the view. The view shows (in various ways) the data the model knows.

Assume that an accident happened in your area. The model in this case is the data: where exactly it happened, how many cars were involved, any injuries, and so on. A local TV channel, the controller, covered the event, and a short segment will be shown on the evening news. Your TV is the view, and you click your remote control to pick your favorite channel.

What's the story with SonarQube and all this architectural layers stuff? As with every design pattern, MVC is a design principle. It gives you guidelines about structuring your applications, but it doesn't obligate you to follow any rules. Fortunately, SonarQube allows you to define architectural rules to police access among layers.

Before we explain how, figure 7.17 shows the layers in an MVC-structured application. Similar layers and technologies apply to .NET and other languages. It's a common practice to name all packages in each layer with a prefix to suggest their location in the MVC structure. For example, packages in the presentation layer are often named `com.mycompany.ui.*` for *u*ser *i*nterface, or `com.mycompany.web.`. In the business logic layer, it would be something like `com.mycompany.ejb.*`; and in the persistence layer, `com.mycompany.model.*` or even `com.mycompany.hibernate.*`.

According to the MVC pattern, there should be no access from the data layer (`com.mycompany.model.*`) directly to the view layer (`com.mycompany.ui.*`). Although it would be great to prevent such access, you can't. But you're not entirely powerless. You can't remove the possibility of creating those interactions, but you *can* catch them and correct them quickly with SonarQube's architectural rules. This feature is only available for Java projects with activated byte-code analysis (which is on by default in SonarQube), so be aware that if you enable the `sonar.skipDesign` property, you won't be able to use it (see appendix B for more details).

Chapter 2 discussed issues and where they come from, and chapter 13 will give you full details on administering rule sets. Here we'll just provide a few guidelines on how to activate and customize SonarQube's architectural rules.

First, there's only one rule you care about at this point: it's named *architectural constraint*, and it's deactivated. This is the rule you'll use to set up your constraints. SonarQube offers a few rules that act as templates, and the architectural constraint rule is one of them. That means you can modify it directly to impose a single architectural constraint, or you can make multiple copies of it and set up as many constraints as you'd like. Clearly, the second route is best.

You'll have the opportunity to name your copies. It's a good practice to give each one a name that reminds you of its purpose; and although the name doesn't need to be unique, it's a good idea.

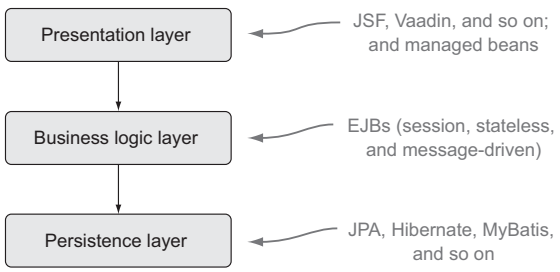


Figure 7.17 Typical layered architecture of a Java EE application

You also need to supply `fromClasses` and `toClasses` properties, and this is where the rubber meets the road. Once enabled, your new rule will flag as an issue every instance of something in the `fromClasses` category accessing something in `toClasses`. Both inputs accept class names or packages names, and using wildcards to broaden a rule's scope means you don't have to maintain a lot of granular rules.

> **Rule isolation**
>
> Defining an architectural rule set especially for large-scale layered systems is a powerful feature of SonarQube. But after a while you may find yourself writing overlapping rules or rules that seem to be hierarchically connected to each other. How does SonarQube deal with those rules? Well, each rule is handled in isolation; so if two or more rules check overlapping constraints, you'll get one issue for each rule.

Once you're done creating rules, be sure to run a new analysis to check your work. In fact, because no syntax checker is built in to rule administration, it may be useful to start with a test rule you know will be broken so you can verify your methodology. It will be easy enough to get rid of it if you don't need it.

7.4 Summary

Measuring design and architecture quality can be difficult to get your head around. Fortunately, SonarQube generates metrics for these facets of program quality and presents them in ways that are easy to understand. All you have to do is to look over metrics and take action, if needed.

We hope you've learned the following by reading this chapter:

- Dependency cycles can decrease the productivity, maintainability, and compatibility of your project.
- The Dependency Structure Matrix (DSM) is a valuable tool to recognize cycles in your projects and dependencies between software components.
- If you're in a Maven-oriented software house, SonarQube makes finding dependencies in your projects quick and easy.
- SonarQube lets you set custom architectural rules to monitor access between specific packages and help you maintain the cleanliness of your application layers.

At this point, we've covered all seven of SonarQube's Axes of Quality, and you have a thorough grounding in what SonarQube has to tell you. Unless you're on a brand new, greenfields project, you're likely feeling a little overwhelmed by what you've learned and the magnitude of the technical debt SonarQube has revealed. In the next few chapters, we'll look at how to deal with that technical debt—from high-level strategies to day-to-day operational tactics.

Unless you're very lucky, you've used SonarQube to kick over a rock. Now we'll show you how to deal with what's underneath.

Part 2

Settling in with SonarQube

In part 2 we'll help you integrate SonarQube into your regular work flow. We'll start in chapter 8 with a discussion of strategies you might want to adopt for addressing your technical debt. Then we'll move on to more hands-on parts of a successful approach. In chapter 9, we'll cover Continuous Inspection with SonarQube and how it can lead to continuous improvement. Then we'll move on to code reviews in chapter 10 and discuss how to make the most of SonarQube's review features. We'll also share some general techniques we've seen used to successfully structure and run code reviews. Finally, we'll take you to where the rubber meets the road, with a look at SonarQube's IDE integration in chapter 11.

8 Planning a strategy and expanding your insight

This chapter covers

- Planning your strategy
- Project history and metric trending
- Everything's a component

So far, you've installed SonarQube and begun analyzing your projects, and you've taken an in-depth look at each of SonarQube's Seven Axes of Quality, the seven ways SonarQube measures what's right or wrong with your code. If you're typical (and don't worry, you probably *are* typical) you're wondering how you'll ever get through all the things SonarQube points out that need work.

It may be intimidating, but it's probably also clear that what SonarQube offers is too valuable to walk away from. So now what? As with any big job, the best thing to do is break it down into small, manageable chunks, and that's what the next few chapters will help you do. In part 2, we'll help you see how to begin fitting SonarQube and what it has to say into your daily and weekly routines with Continuous Inspection, code reviews, and IDE integration.

We'll start in this chapter by giving you some background knowledge that's useful in dealing with SonarQube, such as a better understanding of the history of

your project, which SonarQube builds through data snapshots, and project events and how they relate to the database housekeeping algorithms. We'll also show you some of the other views of your data SonarQube gives you. Part 1 of this book focused mainly on the default dashboard and metric drilldowns, but if raw numbers don't speak to you, there's still hope. SonarQube offers several other presentations, and in this chapter you're likely to find one that gives you what you need.

Before we get to the other presentation formats, we'll begin with what's probably foremost in your mind right now: how to whittle down your technical debt. We've spent quite a while explaining what SonarQube's numbers mean and helping you understand why you should care. Now that you're sold, we'll help you figure out what to do about the problems SonarQube helps you identify.

8.1 Planning your strategy

In chapter 1, we asked you to imagine that your CEO's Aunt Betty was a customer, one with a tiny account and big opinions. When she was hit by bugs in a new release, the CEO went through the roof and demanded that you find a way to measure quality and show improvement. Now that you've got SonarQube up and running, you certainly have a way to measure quality—but it's not measuring very much of it. In your gut, you've known it all along, but now you're facing clear evidence that years of developer turnover and jumping to the hottest new framework every few years without removing the old one have left your project in a shambles. Worse, your CEO has seen the numbers too. Where he was raging before, now he's almost too quiet. You know you've got to come up with a plan. Fast.

First, don't panic. Unless you're lucky enough to be on a relatively new project, you're probably facing a technical debt that built up over years. It's perfectly reasonable to expect that it will take at least months, if not years, to clean up. (And once you show your CEO that you've got a plan, he'll probably calm down and agree.)

Second, remember that the best way to approach a problem of any size is to break it down into smaller pieces. Even though they're all important, and no one quality axis in isolation will give you the full picture, you can't address all seven axes at once. So tackle them one at a time. Now that you've understood and assessed the full weight of all seven, start with the one or maybe two that call out to you the loudest from the project dashboard. That could be any of them; but for most people, issues are the sore thumb, because they're about the things in your project that are demonstrably wrong. So that's what we'll focus on here.

The third step is getting the team on board. Diverting some of your time to quality remediation and away from new features will require buy-in and backing, not just across the development team (*including* testers, architects, and project managers) but also from management and your business partners. For the technical folks on your team, showing them what SonarQube has to say usually gets their attention and some degree of buy-in. For the nontechnical types, like your business partners and certain levels of management, you'll probably need to pitch SonarQube as an investment—

not necessarily in quality, because that may be a bit too esoteric. Instead, be candid about the fact that low-quality code is harder to maintain. It takes longer, and the process of making changes is more error-prone than it should be. Allowing time for remediation now will help you give them what they want (more features and fewer bugs, delivered faster) in the future.

The next step is to pick the exact metric or metrics you want to track and decide how you'll approach them; that's what we'll spend our time on in this section. We'll begin by looking at what to consider when picking a metric you want to track long term, and then discuss a few strategies for improving it.

Before we move on to that, we need to touch on the final step: keeping everyone focused. Choosing a metric to work on and getting everyone excited about it won't do much good if the effort trails off after a month. To keep the technical staff focused, weekly code reviews can work wonders. We'll go into detail on SonarQube's code-review functions in chapter 10. To keep your nontechnical teammates on board, you'll probably want to exploit the history and trending graphs we'll show you later in the chapter. Just seeing the graphs move in the right direction should help your teammates stay motivated, too.

8.1.1 Picking a metric

We've already said we'll focus here on issues, so we could say that the target metric for this discussion is "issues." But that's a bit vague. It will be easier to rally the troops and keep them rallied if the target metric is specific and everyone is on the same page about what it means. Narrowing the field to issues still leaves a number of specific metrics you could choose, and picking one isn't as straightforward as it may seem. To show you why (and lead you to our favorite issues metric), let's walk through the pros and cons of a few.

Rules Compliance Index

Focusing on the Rules Compliance Index (RCI) is tempting because it's a high-visibility number. It's shown in the default filter (the front page) and again on the default dashboard in the rules compliance widget shown in figure 8.1.

It's also tempting to use RCI because everyone loves a good percentage, right? The problem is, it's based partly on project size. So a developer could inadvertently cause a drop in the RCI by cleaning up duplications, which is behavior you want to encourage. That's because the RCI is calculated by dividing a Weighted Issues (WI) score by the project's number of lines of code (LOC). Eliminate lines of code, and you torpedo your RCI score, discouraging behaviors that benefit the project.

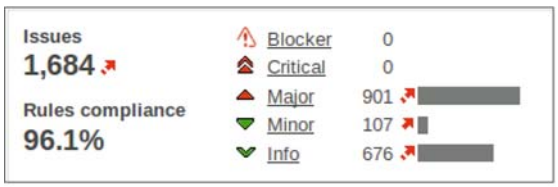


Figure 8.1 The rules compliance widget appears on the default dashboard, making the RCI a prominent gut-check metric. It also shows the count of issues at each level of severity.

Conversely, adding a few new domain classes with lines and lines of issue-free getters and setters will boost your RCI without fixing a single issue.

While we're talking about the RCI, note that the problem isn't limited to that metric. Any of SonarQube's percentages will suffer from similar side effects. These percentages are good gut-check metrics, but for tracking progress, they're probably not what you want.

COUNT OF BLOCKER AND CRITICAL-LEVEL ISSUES

By focusing on the specific counts of high-value issues, you're choosing metrics that aren't impacted by changes to the project's LOC. Additionally, meeting a goal for these numbers means high-value changes for your project. And if you're working with a single project or a group of projects that are similar metrics-wise, this may be exactly the right way to go.

But if you're setting goals across a set of diverse projects, you may have a hard time balancing where to set a goal based on these numbers between projects that are already in decent shape and those that need a lot of work.

Additionally, a developer who notices a few Major-level issues in another part of the file he's working on and cleans them up while he's in there has improved the project's quality, but he's had no impact on the target metrics. That tiny disincentive could be the difference between cleaning up those Major issues and letting them lie to move on to other work.

WEIGHTED ISSUES

The WI metric, which is part of the RCI, is also immune to swings in a project's LOC count. Unlike the Blocker and Critical counts, choosing WI as your focus metric has the additional advantage that nearly every issue eliminated has an impact on the goal you set against it, because the WI formula includes everything but Info-level issues. The exact formula is as follows:

```
(Blockers * 10) + (Criticals * 5) + (Majors * 3) + Minors
```

Thus although fixing 10 Minors counts toward a goal, developers are incented to go after the more important issues first.

Additionally, it may be easier with a WI-reduction goal to find a balance between the needs of healthy projects and those that need a bit more attention. Because a project's WI score encompasses nearly all its issues, setting a target reduction number in WI points lets teams with cleaner projects work through their Blockers and Critical issues and then move on to the Majors, and so on, while the teams with greater challenges focus on their highest-value issues.

WI ticks a lot of the boxes as a desirable metric to track, but it does have one minor downside: it's not visible by default. It's not shown in either the default filter or the default dashboard. But you can easily add it to your filters, and chapter 13 tells you how. There are also a number of ways you can add the metric to your dashboard, including a WI-focused version of the issues widget that's shown in figure 8.2 (available in the Widget Lab plugin). This is our favorite issues-related metric for long-term

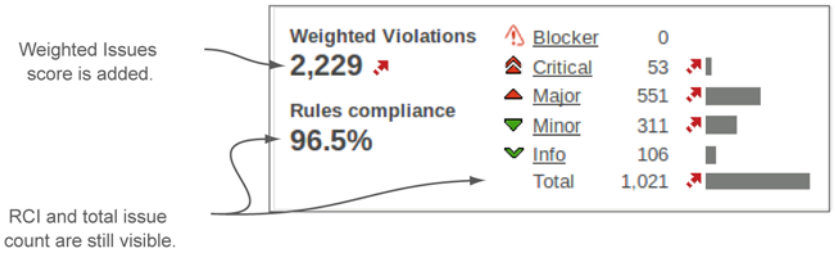


Figure 8.2 This version of the issues widget includes all the metrics in the standard rules compliance widget, and it also shows the project's WI score. (Issues used to be called violations.)

tracking, but you'll have to make an effort to see this number if it's what you choose for your focus.

If issues aren't your target, your candidate metrics will be different, but you should take similar factors into account—particularly when looking at percentages. Once you've picked a metric to focus on, you need to decide how you're going to approach it. What follows are a number of methods we've seen work well. Each method is effective on its own, but don't think you have to choose just one approach. Many of these methods work well together.

8.1.2 Holding your ground

The first approach we'll look at is just holding your target metric steady. It can be alternately stated as "Just don't let it get any worse." This may sound like sandbagging, but for a project under active, heavy development, this approach can be surprisingly challenging—particularly for a team whose members have varying degrees of experience.

In addition to teams with very active development, this may be a good approach for teams that are new to the world of SonarQube and a bit intimidated by the thought of having to improve quality while they do the "real work" of satisfying customer needs. In that case, a first goal of treading water for a while lets the team get comfortable with SonarQube before being asked to step up and start moving the numbers in the right direction.

If you do take this approach, you may be surprised to find that after an initial adjustment period, the numbers *do* begin moving in the right direction. In our experience, good developers are passionate about good code, and once they see (and accept) what's wrong in their projects, they'll want to fix it. Like straightening a crooked picture, they'll be compelled to put it right.

8.1.3 Moving the goal posts

The next method is a bit like weight training, where you start with a weight you can just manage and work at it until lifting it becomes easy. Then you increase the weight and begin again.

In this case, you start with a limited analysis (either by using only the most important rules or by analyzing only the most critical parts of your code), then you clean up the problems, and then you make it harder (by adding either more rules to your rule set or more code to your analysis). In other words, every time your coding team cleans up the project, you move the goal posts.

The main benefit of this approach is that you're never faced with an overwhelming task. And if your whole team understands at the outset that the goal posts will be moving, this could be a great way to handle your project's initial cleanup.

The downsides of this method are as follows:

- It messes up your trending. Each time your project approaches or attains perfection, you change the goal, and quality seems to plummet. If you like this approach, but the trending is important, you may be able to use the `sonar.projectDate` analysis property discussed in appendix B to retroactively apply the enhanced profile.
- If your goal is rule-based, this approach limits your ability to use the teaching aspect of SonarQube's rules. Seeing what code is flagged with issues helps developers learn to write better code. Hide some of those best practices, and even the best developers may inadvertently add Majors that you'll hold them accountable for later, while they're fixing the Blocker- and Critical-level issues you're showing them now.

8.1.4 Boy Scout approach: leave the class better than you found it

The Boy Scout approach builds on the "holding your ground" method. Rather than stopping at "don't add any new issues," it says that you clean up the existing issues in files you touch in the normal course of satisfying customer needs.

Because this approach limits your quality-centric changes to classes that need testing attention anyway, it should help limit the impact of the team's quality-remediation changes on the QA folks. The downside of this approach is that frequently modified classes quickly reach squeaky-clean status while your code backwaters continue to stagnate.

If you choose this method, you need to decide how rigorously to apply it. For instance, typically you'd like developers to fix all the issues in the classes they have to modify anyway. But if Susan needs to make one small change in a large, complex, issue-riddled class, should she be required to clean up the entire thing? Or would it be enough in this case to limit her changes to only cleaning up the method she needed to modify?

Stick with this approach, and eventually even your backwaters should be cleaned up, but it may take a while. If "eventually" isn't fast enough for you, you may want to combine this approach with one or both of the next two methods.

8.1.5 SonarQube time: worst first

This approach is a combination of two distinct mechanisms. The first is *SonarQube time*, or setting aside a specific amount of time each week (or each sprint) for quality remediation. Teams that use this approach refer to it as SonarQube time because it's when they're explicitly focused on what SonarQube has to say. The rest of the week, they're practicing the Boy Scout method or at the least holding their ground. But for a certain number of hours every week, they're working through what SonarQube reports and knocking out problems.

During those hours they use a *worst-first* approach, picking the file or class that scores the worst against the team's target metric and working through it, fixing problems. This can require a little coordination if multiple folks are taking their SonarQube time at once, but otherwise this is a great approach if you have the freedom to dedicate time for quality on a regular basis.

There's another whole dashboard that makes this approach an easy one to take. The Hotspots dashboard is composed mainly of hotspot widgets, each set to show a different metric. If your target metric isn't included, you can easily add another widget instance or edit one of the existing instances to show your target metric. You can also choose the number of files to show. Figure 8.3 shows the hotspot metrics widget set to show the five worst (highest-scoring) files for weighted issues.

8.1.6 Re-architect

If none of the previous methods seems right to you, it may be because a more drastic approach is in order. In *The Mythical Man-Month* (Addison-Wesley, 1975) Frederick Brooks advised that when you're working with new concepts or technologies, you should plan to "build a system to throw away" so that your second system, the one you'll actually use, can incorporate the lessons you learned the first time around.

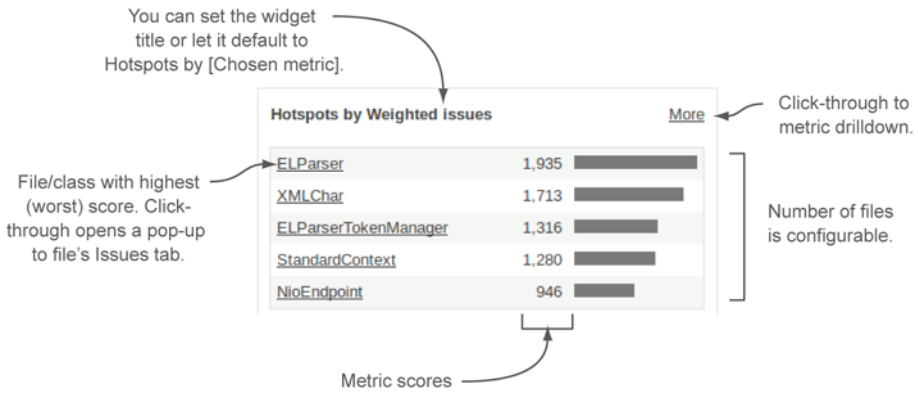


Figure 8.3 The hotspot metrics widget provides an easy reference for which files or classes have the highest score for any given metric. You can have as many instances of the widget as you like, each configured to a different metric.

Clearly, that's not always practical, and we're not necessarily advocating here that you toss your current project code. We *are* saying that sometimes you find yourself with a project that's such a conglomeration of disparate ideas and technologies that patching up issues, or adding tests, or eliminating duplications would be like putting a Band-Aid on a broken leg.

If you find yourself in that situation, your best bet is likely re-architecting. Many times that can be accomplished without huge, disruptive change. For instance, switching (or unifying) frameworks can often be done gradually, migrating one piece of the code at a time from the old framework to the new and giving everyone involved time to accommodate the changes.

Whether you do it slowly or suddenly, in the process of refactoring you're likely to remove large swathes of issue-riddled, duplicative, poorly structured code. Because you're more attuned to quality now, the code you replace it with will be cleaner, better structured, and more thoroughly tested. (Right?) Almost as a side effect of making your code base more maintainable, your SonarQube metrics will improve.

When it's needed, this is the best approach to take. The hardest part is facing—and then selling—that it's needed.

8.1.7 The end game

Now you've seen approaches to remediation that run the gamut in terms of aggressiveness from seemingly passive (the "holding your ground" approach) to rather bold (re-architecting). We hope we've presented at least one that will work for you. Just keep in mind that it's only a starting point.

Yes, we've advised you to pick one metric and focus on it, but you won't be finished when you've mastered it. This is a long-haul effort for a couple of reasons. The first is that addressing only one quality axis isn't enough. Eventually, you'll want them all under control. You'll begin by choosing one or two metrics, but as you whip each quality axis into shape, you'll want to add another one, and another, and so on, until you've got a handle on all seven.

As you add axes, give some critical thought to the order that makes the most sense for you. For instance, working on getting your API documented before tackling duplications may not be the best approach. Similarly, if you're faced with a combination of low test scores and high complexity, you may not want to spend a lot of time writing tests for code you know you need to refactor.

The second reason code quality is a long-haul undertaking is that once you've started seeing some quality wins, you won't want to backslide. Code reviews and Continuous Inspection, which we'll talk about in chapter 9, can help prevent that, but eventually you'll want to set quality gates as backstops. For instance, you may say that you can't release a new version to production if there are any Blocker or Critical issues (maybe limit this rule to new issues at first) or if test coverage has dropped. Assuming you have management backing to delay a production release, then these kind of criteria can help keep code quality top-of-mind.

You'll gain that management backing as you begin to show progress and get some quality wins under your belt. But you may wonder how you'll track changes and show that progress. That's next.

8.2 History and trending

It's all well and good to pick a metric and try to move it in the right direction (or hold it steady for a while), but what's the mechanism for keeping track? Are you supposed to write down the number for your metric's starting point and do the math every time? In a word, no.

One of the best things about SonarQube is that it doesn't just give you a look at the current state; it offers trending as well. We promised that in chapter 1, and it's finally time to deliver. We'll look at differentials in chapter 9; they give you a quick comparison of current state against one of three points in history, but there are other mechanisms for tracking trends in SonarQube. They're up next.

8.2.1 Time Machine

SonarQube offers an entire dashboard devoted exclusively to trending. It's called the Time Machine, and it also touches most of the Seven Axes of Quality. Additionally, it provides something else we promised in chapter 1 but haven't delivered on yet: graphs! Figure 8.4 gives an overview.

This dashboard is focused on showing not current state, but your project's progress over time. Sure, you've added 40 new features, but has that come at the cost of

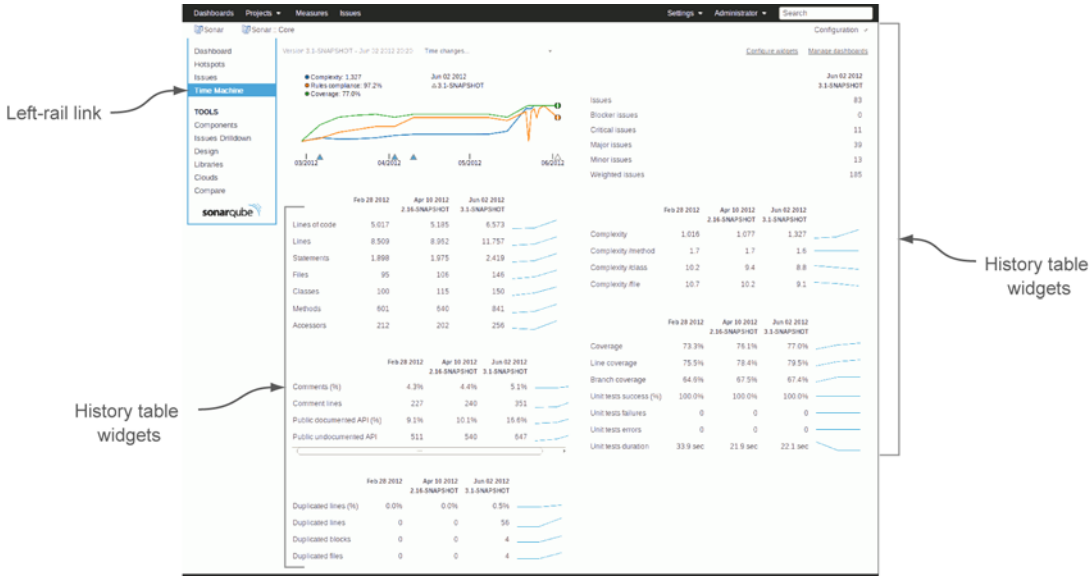


Figure 8.4 The Time Machine dashboard is composed of multiple instances of the history table widget, each configured for a different quality axis, and the timeline widget, at upper left, which gives a granular history of up to three metrics.

quality? And if so, in what areas? The Time Machine dashboard can help you answer those questions. It's composed of multiple instances of the history table widget and one copy of the timeline widget. As with each of the dashboards, you can edit the Time Machine dashboard to change the balance if you like, and each widget instance is itself configurable.

At upper left on the Time Machine dashboard, the timeline widget offers a colorful, granular view across each snapshot in your project's history. It covers up to three metrics and is one of SonarQube's sexiest out-of-the-box widgets. Figure 8.5 shows a close-up.

Although graphs are typically intuitive—that's the beauty of a graph, after all—you can't take the timeline at face value. Because there's no fixed y-axis, each line in the graph is relative only to itself. A line in this widget can jump the full height of the graph for a change of hundreds, and it can jump the full height of the graph for a change of less than one. Figure 8.6 illustrates the point.

Despite the mild caution required with the timeline widget, it's still an excellent tool to give at-a-glance trending of key project metrics over time. By default, it shows

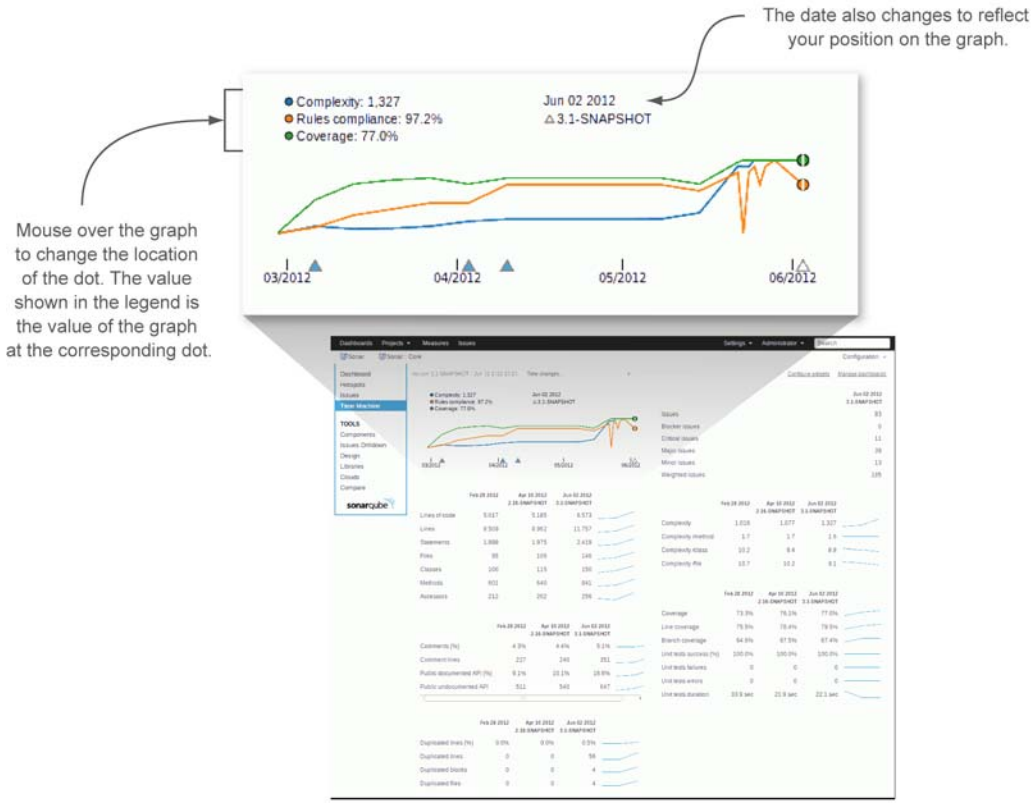


Figure 8.5 The timeline widget, at the dashboard's upper left, offers a colorful, granular graph of your project's history. There's no fixed y-axis, though, so mouse over the graph to have the relevant values shown in the legend.

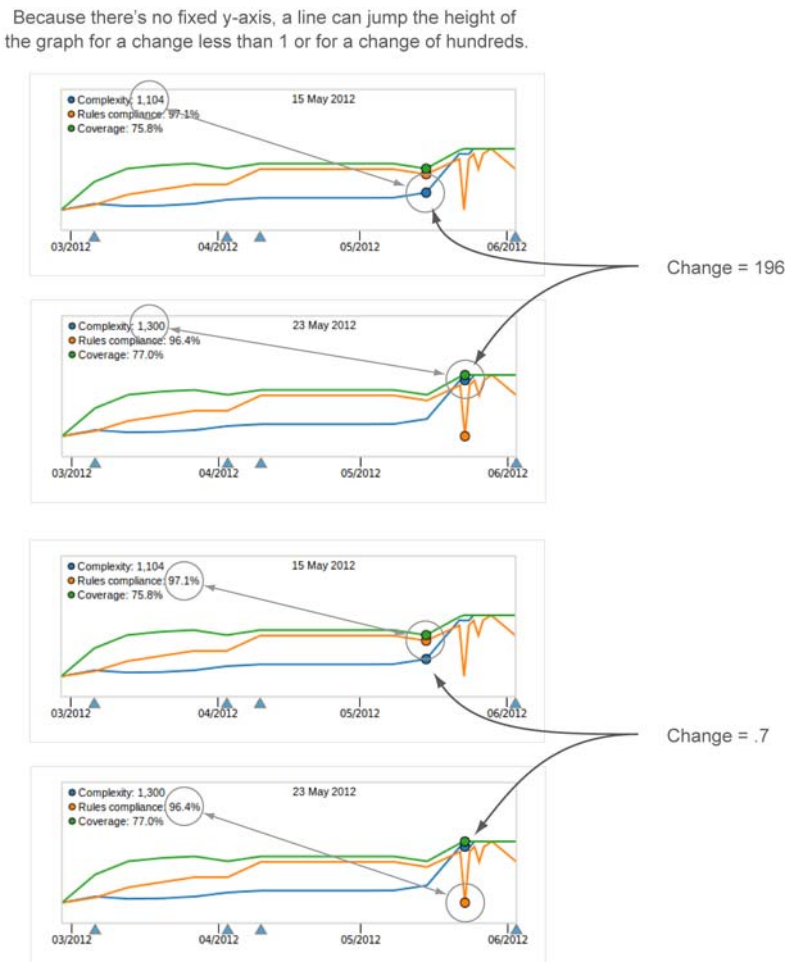


Figure 8.6 Don't let your intuition rule too strongly when you're reading the timeline widget's graphs. Because each line in the graph is essentially an independent sparkline, you need to mouse over the graph to see the numeric values whenever they show what looks like a large jump. As the bottom two figures show, a line can jump the full height of the graph for a change of less than 1: .7 in this case. The upper two figures show the opposite end of the spectrum, with a full-height jump for a change of 196.

complexity (which you want to see headed down), rules compliance (which should be headed up), and test coverage (which we hope is also headed up).

The other widget on the Time Machine dashboard is the history table widget. Each instance gives you precise values for up to 10 specific metrics and sparklines for those values at points in your project's history. Figure 8.7 gives you a closer view.

By default, those points in time are the first analysis on record, the last analysis on record, and the analysis for the most recent `version` event. We'll cover events in detail next; but briefly, when you change the value of the `sonar.projectVersion` analysis

Figure 8.7 The history table widget shows sparklines and historical values of up to 10 metrics. By default, it shows the first value on record, the last value on record, and the value from the last time the project's version number changed. The sparklines to the right reflect only those values shown in the widget, not the kind of detailed graphing that's available from the timeline widget shown in figure 8.6.

property, it triggers a `version` event, and that snapshot is marked for special treatment. One of the ways that snapshot is treated differently is that it's eligible to be shown in the history table widget.

Both the number of metric rows and the number of snapshot columns are adjustable. As you increase the number of snapshot columns, you see additional `version` event snapshots, working backward from the ones already showing. Because the sparklines this widget shows reflect only the values displayed in the table, rather than the values from the fully detailed project history, the more snapshot columns you include, the more detailed your sparklines.

Now let's take a closer look at exactly what those `version` events are.

8.2.2 Events and database cleanup

We've breezed over the topics of events and database cleanup previously. Now it's time to get into the details. Essentially, *events* are special flags on your project snapshots, and snapshots with event flags get special treatment. Specifically, they're exempt from database cleanup.

It's analogous to old-fashioned photo snapshots. Take one every day or multiple times every day, and you'll have a lot without long-term value. But snapshots from events (graduations, holidays, and vacations, for example) have long-term significance. Those you hold on to. SonarQube does the same thing.

There are four kinds of events:

- *Version*—The value of the `sonar.projectVersion` analysis property changes. These events can also be set retroactively by a project administrator.
- *Profile*—A change is made to the rule profile against which your project is analyzed. This is when rules are added, removed, or edited.
- *Alert*—An alert status changes. You can set alert thresholds on rule profiles. When your project crosses a threshold in either direction, an event is recorded.
- *Other*—The project administrator manually sets an event on a snapshot.

Why is it even a question? Because each snapshot holds a lot of data. If you run multiple analyses a day, that adds up to a lot of snapshots and a whole lot of data over time; so SonarQube does regular housekeeping. You may have noticed lines at the end of your analysis logs that start with "Keep one snapshot per …." That's the database cleanup we've been talking about. It's run for each project as part of the analysis. By default, SonarQube keeps the following snapshots:

- *One per day*—After the first 24 hours
- *One per week*—After the first month (4 weeks)
- *One per month*—After the first year (52 weeks)
- *None*—After 5 years (260 weeks)

Like many other settings, these thresholds are editable, both globally and at the project level; we'll show you how in chapter 14 for the global level and in chapter 15 for the project level. In case you ever need to know (you probably won't), we'll also show you how to manually clean out snapshots in chapter 15.

Now that you have a better understanding of events, we want to point out a feature of the timeline widget that we left out before: it shows events (see figure 8.8).

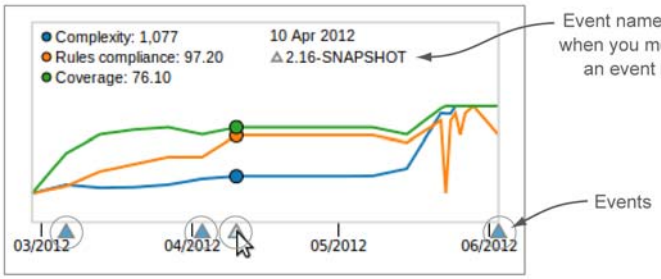


Figure 8.8 Each event in a project's history is marked by a blue triangle on the X axis. Mouse over the triangle to add its name to the graph.

8.3 Everything's a component

In this chapter we promised you multiple ways to look at your data, and the next one is based on a concept that we haven't talked much about yet. SonarQube treats each project, module, package, and file as a component. That means just as it calculates metrics at the project level, SonarQube also calculates them for each module, submodule, package, and file in those projects. You can easily get a test-coverage score not just for a project as a whole, but also for each package and class it contains.

You can see this concept at work starting from SonarQube's default front page, with the treemap widget at lower right, like the one shown in figure 8.9.

Assuming you've got multiple projects under analysis, you'll see the treemap broken up into multiple subrectangles, one for each project. The size and color of those blocks convey two key metrics. Block size reflects LOC; and color, which ranges from a cool, grassy green to an angry red, ties to rules compliance. In general, the closer a block is to an angry red, the more quickly you should pay attention to it.

The treemap gives you two different ways to drill in. Click the block label (the component name) to land at a dashboard for the component you've chosen, whether that's a project or something at a lower level, and click in the unlabeled portion of the block to drill in to a treemap for the chosen component's subcomponents. Drill down far enough, and you'll find yourself looking at filenames. Clicking the filename opens the file detail pop-up.

8.3.1 Project component view

Next, let's drill in to a project and choose the Components link in the left rail. The presentation you see looks a lot like a filter. But instead of showing the projects in your SonarQube instance, it shows the modules or packages in your project, with a few key

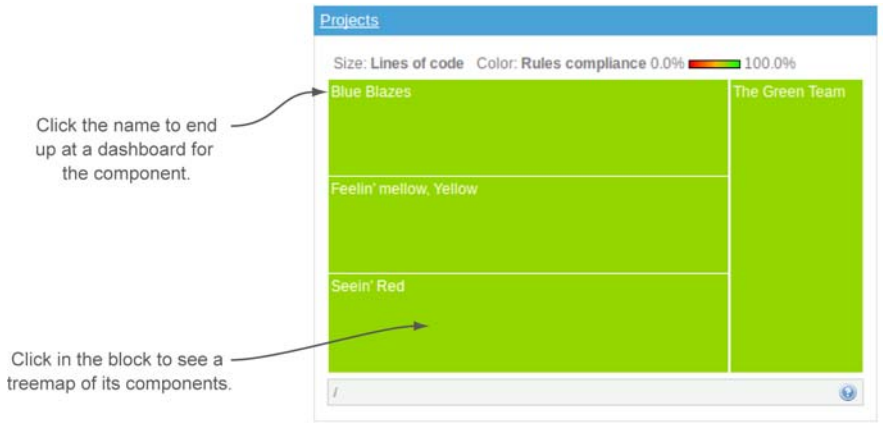


Figure 8.9 The treemap defaults to a subblock for each project, with relative block size indicating Lines of Code, and block color (cool, grassy green to angry red) indicating rules compliance. Click in the unlabeled part of the block to drill in to a treemap of that block's components. Click-through on the block's name to land at a dashboard for that component.

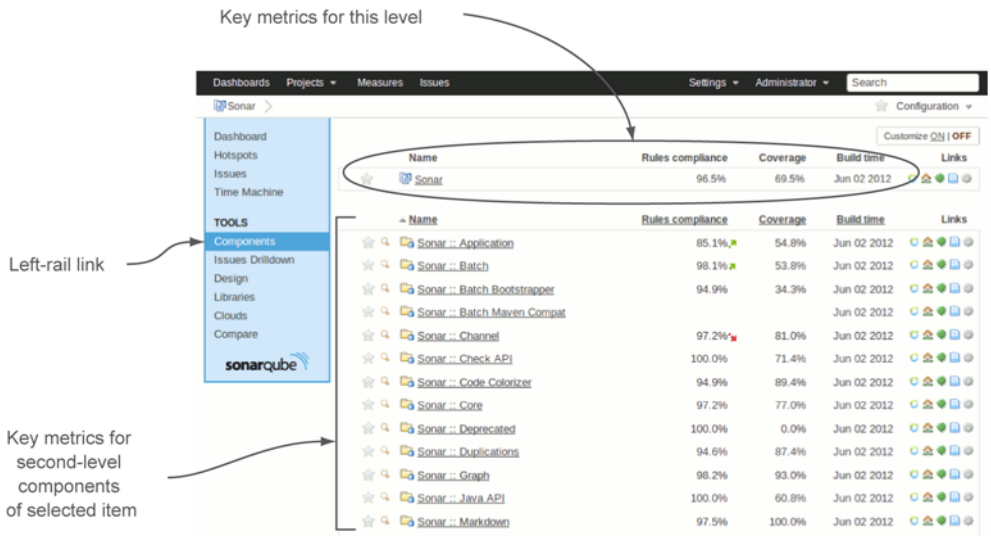


Figure 8.10 The Components view shows a filter-style listing of the modules or packages in your project.

metrics for each (configuring which metrics is like configuring a filter, which is covered in chapter 14). Figure 8.10 shows part of the Components view for SonarQube.

Click an item in the filter list, and you'll find yourself at the Components view for that item. Of course, the component names in the list change, but little else does. As proof of that, figure 8.11 shows a drill-in from the Components view shown in figure 8.10.

Continue clicking in the textual component list, and eventually you end up at a level that only shows filenames. Just as with the treemap, once you're at that level, your clickthrough (on the magnifying glass) pops open the file's detail view in a new window.

If you're not paying attention, it can be easy to get lost in the Components view drilldown. Fortunately, each click adds a link to the breadcrumb trail at upper left to

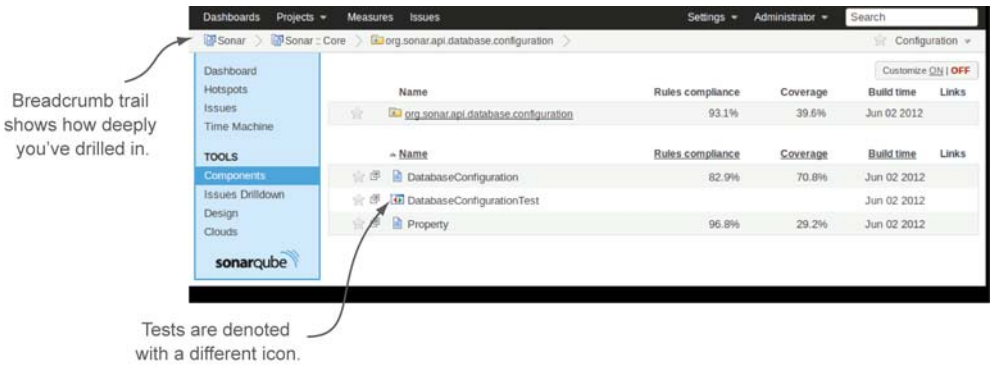


Figure 8.11 The breadcrumb trail that's added at upper left in the interface shows how deeply you've drilled in to the Components view.

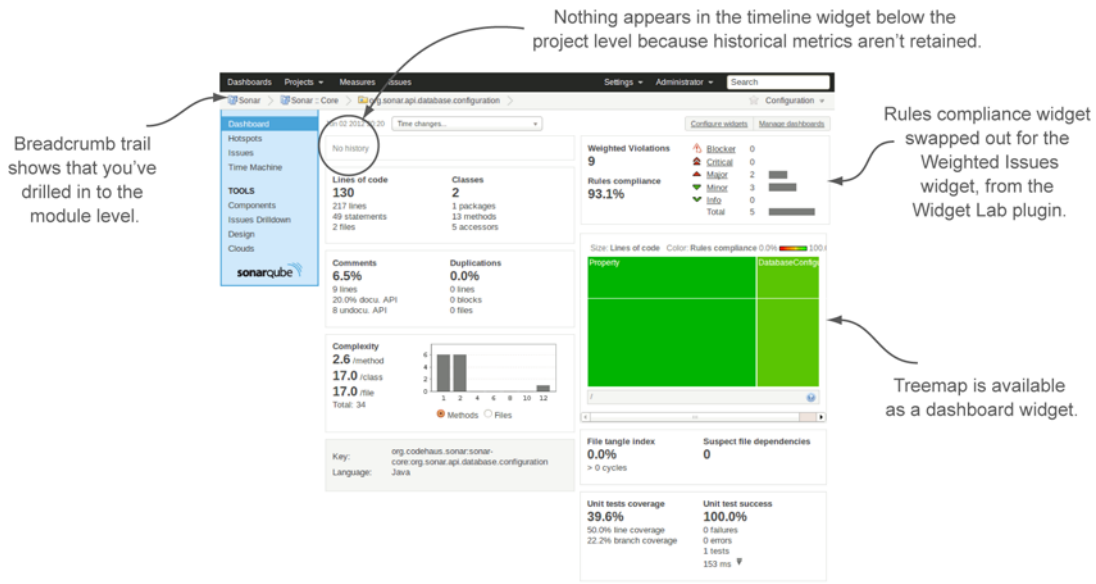


Figure 8.12 As with the drilled-in version of the Components view, the default dashboard looks almost the same from level to level. This version of the dashboard reflects a few of the suggestions we've been making, including the use of the Widget Lab plugin's WI rules compliance widget in place of the standard rules compliance widget at upper right, and the addition of a treemap widget.

help you keep yourself oriented and allow you to walk back up the trail. As with the Components drilldown, figure 8.12 shows there's little to differentiate a component-level dashboard from the one at the project level except the breadcrumb trail.

8.3.2 No package history

Once you're drilled in to components at any level below the project, the dashboard link in the left rail also takes you to a dashboard for that component. Most of the left-rail links work correctly for components, with the exception of the Time Machine.

The dashboard in figure 8.12 is supposed to start at upper left with a timeline widget, which is borrowed from the Time Machine dashboard. The colorful graph the widget offers is a nice addition to what might otherwise be a mass of grey. But when you get to the package level, you don't get a graph from the timeline widget. It only says "No history."

You get similarly disappointing results when you check the Time Machine dashboard at the package level, because although SonarQube generates its metrics at all levels, it only keeps them long-term at the project and module levels. Metrics beyond the last analysis aren't kept for packages and classes, which helps keep the SonarQube database size down to a manageable level and improves performance. So the timeline widget says "No history," and the history table widgets only show you the most recent value for each metric. No sparklines or pretty graphs anywhere in sight.

This behavior is the default. You can easily have SonarQube archive package metrics (globally or only for selected projects), but be aware that doing so will swell your database and potentially degrade performance.

8.4 Related plugins

The related plugins for this chapter play on the "other ways to view your data" theme. The first one, Tab Metrics, provides insight into the metrics collected at the file level. The second, Widget Lab, gives you an alternate dashboard widget for issues.

8.4.1 Tab Metrics

The Tab Metrics plugin can be installed easily through the update center and requires no configuration. In the file detail view, it adds a new Metrics link, which lists the current value for every metric available for the file, as shown in figure 8.13.

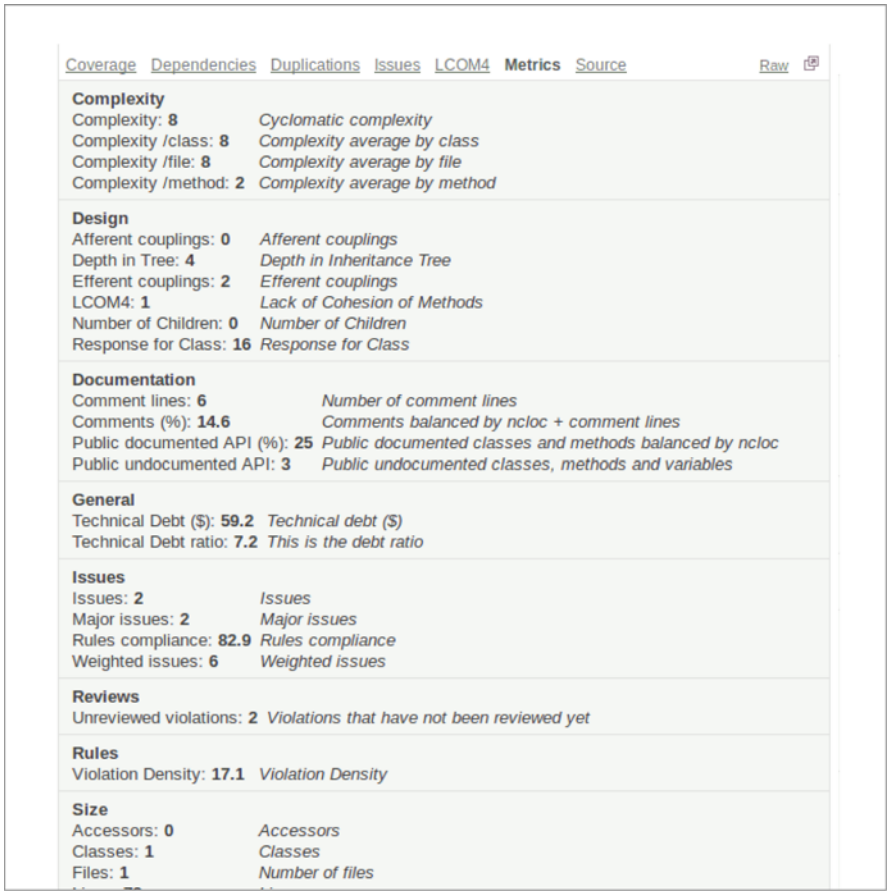


Figure 8.13 The Tab Metrics plugin adds a tab to the file detail view that lists every metric available for the file.

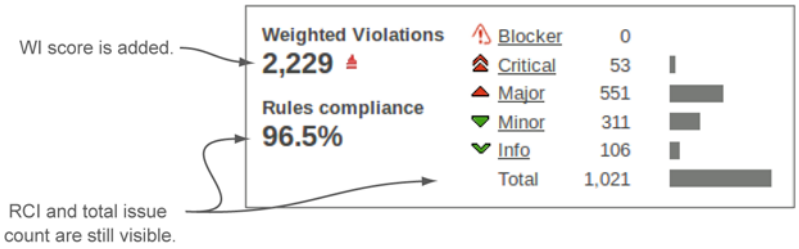


Figure 8.14 You can use Widget Lab's WI rules compliance widget to put WI front and center on your project dashboards.

The metrics are shown grouped by SonarQube's own internal categories. To the right of each metric name and value is the metric description. Often the description is the same as the name, but sometimes it offers a useful expansion.

Be aware that the plugin shows both SonarQube's core metrics and metrics introduced and computed by other plugins.

8.4.2 Widget Lab

Widget Lab was already covered earlier in this chapter, when we talked about picking a metric. If your focus is issues, we urged you to consider using the Weighted Issues metric as your target. Normally that's not a very visible metric. You can make it appear in filters, but at a project level there's no way to see it. That's where Widget Lab's WI rules compliance widget comes in. It gives you all the metrics in the standard rules compliance widget, plus WI at upper left, as figure 8.14 (repeated from figure 8.2) shows.

8.5 Summary

SonarQube analyzes your projects against up to seven different quality axes (depending on the language). For some projects, that can add up to a lot of things that need to be worked on. Instead of being overwhelmed by the size of your technical debt, we hope this chapter has helped you get started by choosing one or two metrics to focus on and creating a plan of attack for moving them in the right direction.

As you pick your target metric, remember that it's best to avoid percentages because they can be skewed by changes in project size. If you decide to make issues your initial focus, we think Weighed Rule Violations makes a great target metric.

We've given you a number of strategies to consider (although certainly not an exhaustive list). As you decide which strategy or set of strategies you'll use, don't hesitate to start with the "holding your ground" method if you need to. It's a great way to bring attention to a new push for quality while the team gets used to SonarQube and to the idea of having its code quality measured. At the other extreme, if it's called for, you may need to re-architect. In the middle are a number of strategies that can work well together: moving the goal posts, the Boy Scout method, and SonarQube time/worst first. Once you're comfortable with SonarQube and making good progress on

your first metric, you'll want to add more to your focus, until you've got them all under control.

Whatever metric and strategy you begin with, your campaign may benefit from using some of the other ways SonarQube provides to view your data, such as the Hotspots and Time Machine dashboards for deeper insight into project history.

Also in this chapter, you've seen how to drill in to your project for module- and package-level metrics. You've learned about project snapshots and the importance of events, and now you know how SonarQube decides which snapshots to keep and which to discard.

In the next chapter, we'll explore trending more fully as part of a discussion of Continuous Inspection, which can help keep your team focused on code quality from day to day.

9 Continuous Inspection with SonarQube

This chapter covers

- Understanding Continuous Inspection
- Triggering your analysis with Continuous Integration
- Monitoring quality evolution

With half the book behind you, you now have a thorough grounding in what SonarQube is telling you, including an in-depth understanding of the metrics behind the Seven Axes of Quality and a handle on how to tackle what those metrics show.

In this chapter we shift gears from theory to practice, with the introduction of Continuous Inspection, a practice that will boost your return on investment in SonarQube. So far, we've mentioned Continuous Integration (CI) a few times and hinted at its advantages. Now we'll go in depth, starting with Continuous Inspection's big picture and how SonarQube perfectly fits into a streamlined development process.

We'll look at why it's important to automate the Continuous Inspection practice and how easily that automation can be done. We'll give you a step-by-step guide to integrating SonarQube with a CI system to automate your everyday analysis, and we'll offer advice on best practices.

Once you've automated your analysis, you'll want to dig in to some of SonarQube's star features: differential views and version comparisons. In chapter 1, we touted SonarQube's ability to help you see not just a project's current state, but its quality evolution over time. Now it's time to make good on that promise with differentials, which we believe to be one of SonarQube's most valuable features. You'll see how to check your projects' progress—from day to day or version to version—and how to configure differentials to meet your needs.

9.1 Introducing Continuous Inspection

Google the term *Continuous Inspection*, and you'll notice that several of the top results are related to SonarQube. What's interesting is that although SonarQube dominates the CI, it's not a concept that originated with SonarSource. It seems to have started in manufacturing, many years before SonarQube's birth, and gradually taken root as a software concept. At the time, CI for software was something you had to cobble together yourself, laboriously adding to your build script each type of inspection you wanted (style, bugs, duplications, and so on). But of course SonarQube rolls all those inspections into one easy analysis, so it shouldn't be surprising that in the last few years, SonarQube has been recognized as a must-have tool for CI.

But before we look at using SonarQube for CI, let's examine some of the concepts behind it to help make the big picture clearer.

9.1.1 What and how?

First, "What should you inspect?" The answer seems pretty clear: an application's source code. But what are you looking for? That's where the Seven Axes of Quality—alternately referred to as a developer's Seven Deadly Sins—come in. They represent developers' most common bad habits (copy and paste, poor unit-testing, dirty code, and more). What you *really* want to do is keep an eye on what we covered in chapters 2 to 7.

With Continuous Inspection in place, you should easily be able to answer these questions:

- How much duplicate code was added last week?
- How well unit-tested are the files committed yesterday?
- Did we introduce any critical issues during the last iteration?

That might seem easier said than done, but with CI and SonarQube's differentials, answering is easy.

To truly take advantage of a CI process, it should be a recurrent, automated activity that doesn't steal more than a few minutes (if any!) from your day. Once you set it up, it should work without intervention, making fresh analysis results available on a regular basis, so that every morning you sit down to your cup of coffee and a fresh quality analysis. The only thing you'll still need to handle is your coffee preparation. (Sorry, SonarQube's not quite that good yet!)

Effectively implementing Continuous Inspection presupposes a regular build process—CI at best, but at least a nightly build. That's what we'll look at next.

9.1.2 Life before and after Continuous Inspection

To achieve Continuous Inspection, your best bet is to first implement a CI environment if you haven't already. We'll talk about your options in section 9.2. But first, let's look at the advantages of CI.

In a traditional, pre-CI approach, developers check their changes in to the central source control management (SCM) system and get other developers' checked-in changes—ideally, several times a day. Then, on occasion, the "build wizard" puts on her wizard hat to manually perform a build. In this environment, changes—and potential problems—can percolate for weeks or months before coming to the surface.

Add a CI tool, and your build (compile, run unit tests, deploy, and so forth) is triggered on a regular basis: automatically for every commit, or nightly if a build-per-commit seems like too much. Problems come to the surface quickly, and the build wizard can retire the funny hat.

To accomplish this, configure your CI server to poll your SCM at predefined intervals (every few minutes, several times a day, and nightly are popular options), or set up event handlers in your SCM to fire off a build for each commit event—literally have the SCM push the commit event down to your CI system. Then instruct the CI tool to run all required build steps (compile, run tests, deploy, and so forth), and you're done. Figure 9.1 shows what this system looks like.

Our experience has shown that moving from manual to automated builds can be a huge mental leap. If you aren't familiar with CI practices, then you're probably shocked by the thought that your system would be built multiple times a day by a … machine. But consider the advantages. First, you minimize the time it takes to create your application's executable. In fact, you don't spend any time at all on it. It happens regularly and automatically, which means you always have the latest ready-to-ship build for testing, sales presentations, or beta evaluation. All unit tests are executed each

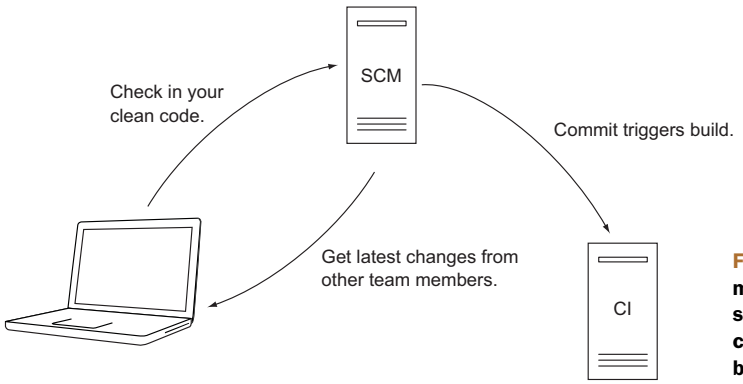


Figure 9.1 Source code management with a CI server that pulls source code changes and triggers builds

time you build your system; and if something goes wrong, all team members are immediately notified. No more dark arts to make sure everyone has committed all their changes. You configure your build job once, and CI knows when to trigger it.

Once CI is set up and humming, it's time to invite SonarQube to join the game. SonarQube integrates easily into most CI setups. You need to follow only two steps:

1. Decide how often you'd like to analyze your project.
2. Create a CI job that runs the analysis.

Figure 9.2 shows the system when you're finished. What you see is a basic CI workflow. As illustrated, SonarQube is triggered automatically by build jobs on the CI server. When the analysis is complete, developers and other team members can get the latest quality reports without worrying about manually triggering the analysis—in essence, their check-ins did that for them.

Now you have an idea of the details. Use CI tools to build your code base automatically, and you can take advantage of that automation to add Continuous Inspection to your system. But let's back up and look at why you'd want to.

9.1.3 The big picture

CI addresses a unique challenge: to help you and your team track and reduce your project's technical debt on an ongoing basis.

Financially speaking, *debt* is an obligation of a creditor to the debtor. For instance, when you take a loan from a bank, you're committed to make a monthly payment

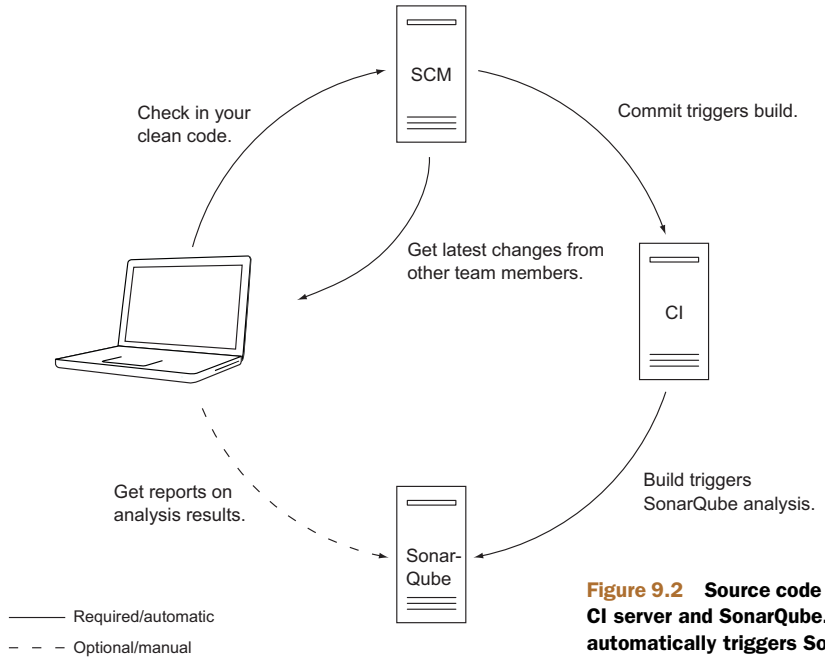


Figure 9.2 Source code management with CI server and SonarQube. Build automatically triggers SonarQube analysis.

until you cover the original amount plus the accrued interest. Usually, the loan is repaid in a fixed period of time. Miss a payment, and the total amount you owe—the debt—increases due to additional interest (and fees); thus you need more money to meet your obligation.

Similar principles apply to technical debt. The difference is that there is no initial obligation. The debt starts to build when the first commit takes place, and it's increased every time you add new code to the project. It's like when you get a new credit card. On the first day, the balance is zero, but every purchase (commit) is a charge to your card (technical debt) that eventually you'll have to repay.

But who is responsible for increasing your technical debt? Every duplicated line, every new issue, and every file not covered by unit tests adds to your technical debt. If you don't take action to pay it down, your technical debt will grow quickly and eventually cost you much more effort—and money—later in the project.

Technical debt comes in three flavors. As you can see, there is a straightforward relationship between the types of technical debt and the Seven Axes of Quality:

- *Code*—Issues, duplications, and absence of commenting and documentation
- *Design*—Poor design, increased complexity, low cohesion and high coupling, and architectural problems
- *Testing*—Low or no unit-test/integration-test coverage, and tests that are hard to maintain

There are plenty of reasons to make charges against your credit card, just as there are plenty of reasons why technical debt is added. And like paying your monthly credit card bill, you know you should pay down your technical debt regularly. But there's always the temptation to avoid dealing with it: "Paying down technical debt will reduce productivity and leave less time for new features." In the short term, that may be true; but by ignoring your debt, you'll end up in a vicious cycle. As shown in figure 9.3, the longer you avoid paying your technical debt, the harder it will be to maintain your system, sending your productivity into a downward spiral.

There is only one way to eliminate this deadlock: pay down your technical debt continuously during the project's lifetime. Make this your goal, and you'll find SonarQube an invaluable tool for reaching it. The rest of this chapter shows you how. We'll start by automating SonarQube analysis with a CI server and then show you how to track changes across time using differential views.

9.2 Triggering your analysis with CI

There are lots of CI servers out there—a dizzying array—but direct CI/SonarQube integration is available for only a few: Jenkins and AnthillPro, both of which are free and open source; and Atlassian Bamboo, a commercial offering. In this section, we'll look at the direct integration of Jenkins in depth and Atlassian Bamboo briefly.

Figure 9.3 Not paying down technical debt will make your system hard to maintain and will eventually decrease productivity.

NOTE Even servers that don't have direct integration can still be used to trigger SonarQube analyses. It's a matter of firing the proper command after the build. The SonarQubeSource documentation offers advice at http://mng.bz/43GI for doing that for three other popular servers: Apache Continuum, CruiseControl, and JetBrains TeamCity. Even if your CI server isn't one of those three, the advice you'll find at this site may be helpful.

If your team doesn't currently have a CI server, you'll need to pick one and set it up. Our favorite is Jenkins. Unfortunately, we don't have room here to show you how to get it up and running, but the Jenkins community documentation can guide you (http://jenkins-ci.org/). Like some of the other tools we've mentioned (and like SonarQube itself), Jenkins is free, open source, and dependable. Plus, paid support is available if you want it. It's also the most popular open source CI tool because it can be used with any programming language; and, last but not least, it's easy to set up even without previous experience.

From this point on, we'll assume you have a CI server up and running. The next step is to set up a CI job. We'll use Jenkins in our examples because, as we've mentioned, it's a favorite—not just with us but generally. Basic knowledge of managing Jenkins jobs isn't required for this section, but it will help you better understand what we'll tell you.

> **Learning about Jenkins**
>
> To learn more about Jenkins, you can download a free copy of John Fergusson Smart's *Jenkins—The Definitive Guide* (http://mng.bz/2dhG). If you're in a Microsoft house, Jenkins can handle your builds beautifully; but you might also look at the well-written book *Continuous Integration in .NET* by Marcin Kawalerowicz and Craig Berntson (Manning, 2011, www.manning.com/kawalerowicz/).

9.2.1 Jenkins setup

The procedure to integrate SonarQube with Jenkins is straightforward. It consists of the following tasks in Jenkins:

1. Install the SonarQube Jenkins plugin.
2. Configure your SonarQube installation.
3. Configure SonarQube Runner.
4. Configure a job that triggers a SonarQube analysis. (If you're dealing with a Java or C# project, also make sure the job performs a build.)

Next, we'll look at each of these steps in detail.

INSTALLING THE JENKINS PLUGIN

Before you start, double-check that both your SonarQube instance and your Jenkins server are up and running, because you'll need them during installation. Then, install SonarQube's Jenkins plugin.

This can be easily done from the Jenkins update center. Point your browser to the following URL: http://[*yourJenkinsServer*]:8080/pluginManager/available (if you changed the port of the CI server, change the URL accordingly), or use the Jenkins UI's navigation by choosing Manage Jenkins in the left rail menu and then Manage Plug-ins. Finally, click the Available tab to get a list of all available plugins. On this tab, you can use the search filter at upper right to quickly filter Jenkins' long list of plug-ins, as shown in figure 9.4.

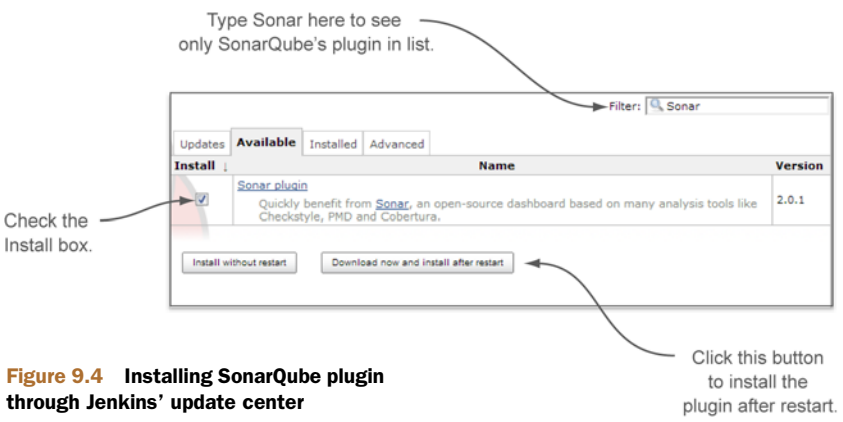


Figure 9.4 Installing SonarQube plugin through Jenkins' update center

Figure 9.5 When the plugin is downloaded, you're notified.

Select the Install check box, and click Download Now and Install After Restart. (You could click Install Without Restart, but the restart doesn't work properly on some platforms.) When Jenkins finishes downloading the plugin, you see a message similar to what's shown in figure 9.5. Restart your Jenkins installation to activate the plugin and proceed with the configuration.

SETTING UP YOUR SONARQUBE INSTALLATION IN JENKINS

With the plugin installed, you're ready to configure it with some server-level settings. Start by choosing the Manage Jenkins link in the left rail. Then choose Configure System, and find the SonarQube section on the page. It will probably be toward the bottom. Figure 9.6 shows what you're looking for.

You need to tell Jenkins about your SonarQube installation. Click the Add Sonar button to add a few fields and some buttons to the page. Then click the Advanced button, and you'll see a slew of additional fields as shown in figure 9.7. It's time to fill them in.

First, give your SonarQube install a name. Skip the Disable field for now—it disables SonarQube for all jobs, no matter what. If Jenkins and SonarQube are running on the same machine, and you haven't changed SonarQube's default port (9000), then you can skip the next two properties (Server URL and Server Public URL). If not, enter the proper values in these fields.

Next you need to set up your database connection and configuration. Click the Help icon (the question mark) next to the Database URL input text to see some URL examples.

After that, enter the credentials you selected when you created SonarQube's database user (see Appendix A for detailed instructions on installing and configuring SonarQube). Finally, enter the database driver. Again, the Help icon will give you examples for the most popular databases. Leave the last two properties empty.

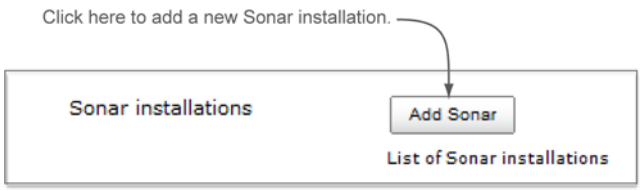


Figure 9.6 Adding a new SonarQube installation in Jenkins

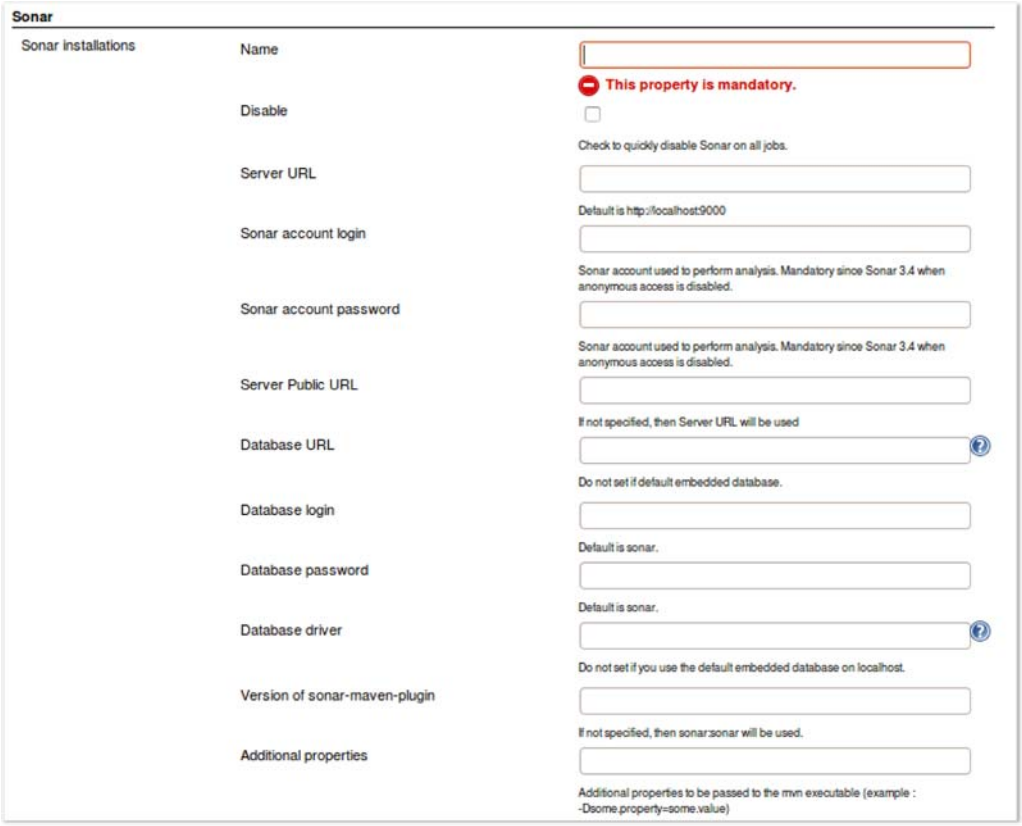


Figure 9.7 The SonarQube plugin's available configuration options

Now proceed to the triggering configuration. Figure 9.8 shows the available options. We suggest that you leave all of them unchecked, because the setup we'll recommend later will require the analysis to be triggered by SCM changes.

To finish the SonarQube plugin configuration, click the Apply button, not Save. If you do click Save, return to the configuration page—you're only half done. Next you need to configure Jenkins' SonarQube Runner.

CONFIGURING SONARQUBE RUNNER IN JENKINS

Find the SonarQube Runner section on the page, and click the Add Sonar Runner button. As figure 9.9 shows, doing so again adds options to the page.

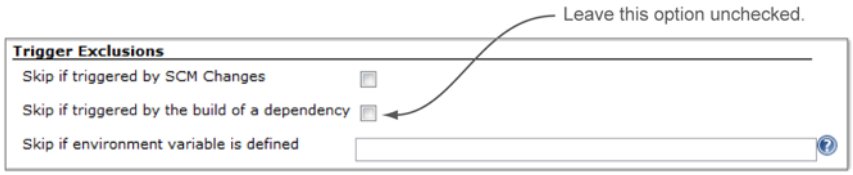


Figure 9.8 SonarQube plugin: available triggering configuration options

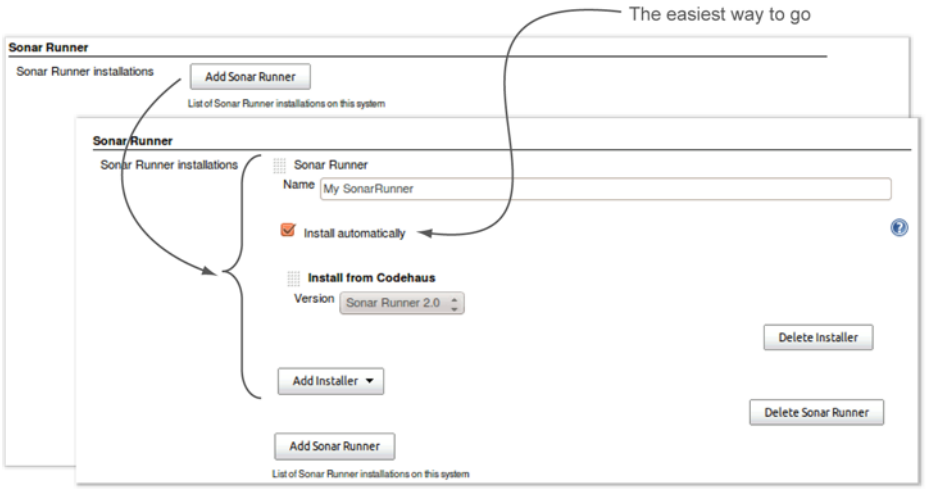


Figure 9.9 Setting up a SonarQube Runner installation is easy. Click the Add button, name the installation, and choose Install Automatically.

Give your installation a name; because it's possible to have multiple instances of SonarQube Runner installed on a Jenkins server, this will help you tell them apart. Then, assuming you're after the easiest route, select Install Automatically, and you're finished. Click Save.

Now you're ready to configure your job to run an analysis.

ENABLING SONARQUBE ANALYSIS IN A BUILD JOB

Create a new job (you'll land automatically at its configuration page), or navigate to the configuration page for the existing job that you'd like to use for analysis. Whether you choose an existing job or set one up from scratch, make sure it includes a step to compile your code if you're dealing with Java or C#, because SonarQube performs byte-code analysis for both languages.

Now you're ready to enable SonarQube in your job, but there are two ways to do that, as figure 9.10 shows. First you'll have to figure out which one to use.

Figure 9.10 SonarQube analysis is available as either a build step or a post-build action, but the only time you want to use the post-build option is for Maven projects.

Figure 9.11 Configuring standalone SonarQube Runner properties

The first option lets you trigger a SonarQube analysis as a build step. Unless you're in a Maven shop, this is the one you'll use. The second option is a post-build action, and it's only appropriate for Maven projects. It's available for all Jenkins jobs, even for non-Maven projects; but if you use it on a non-Maven project, your build will always fail (because you don't have the right prerequisites in place). We'll look first at the build step option, which is shown in figure 9.11.

If you have multiple JDKs configured in Jenkins, the JDK drop-down menu lets you choose among them. Similarly, if you have multiple instances of SonarQube Runner configured, you'll see a Sonar Runner drop-down menu (it disappears if you have only one SonarQube Runner instance). The next two inputs present an either/or proposition. If your project's workspace includes a sonar-runner.properties file (whether it goes by that name or not), provide its path—relative from the project root—in Path to Project Properties. Otherwise, use the Project Properties field to enumerate your project's analysis properties. The final input, JVM Options, isn't typically needed, but it gives you the ability to pass additional JVM arguments to the analysis just in case.

You're done at this point. But before you click Save, be sure your SonarQube build step comes after your job's compile step (you can drag/drop steps to reorder them).

The Maven-only, post-build option can be turned on by picking SonarQube from the Add Post-Build Action menu. This gets you an all-defaults analysis. But if you need more flexibility, clicking the Advanced button presents a host of options, as shown in figure 9.12.

> **TIP** It's always a good idea to click the Advanced button whenever you see it in Jenkins. Many interesting things may be hidden underneath, and the fact that it's not expanded by default doesn't mean none of the values are filled in.

SonarQube uses the Branch and Language properties to specify a SCM branch or a language other than Java. If you need more information about these properties, jump to appendix B, which is about running and configuring a SonarQube analysis. The Root POM field lets you specify where to find your pom file if it's not in the standard spot. The MAVEN_OPTS input lets you pass additional options into Maven for the analysis, such as `-X` to show debug logging. The Additional Properties field lets you define

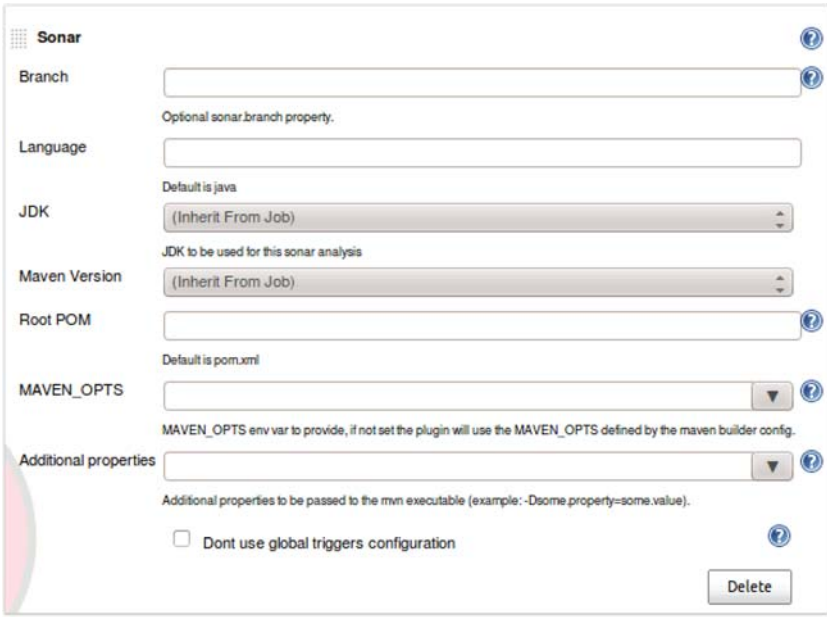


Figure 9.12 Advanced configuration for a SonarQube post-build action

any additional properties that may be needed for the analysis. A valid input string looks something like this:

```
-Pintegration-tests -Dappserver=tomcat -Ddatabase=mysql
```

Finally, you can override the global triggering configuration (those check boxes we told you to skip in the server-level configuration) by selecting the relevant option and adjusting the settings to fit your project needs.

Whichever route you use to configure your analysis, when you're done with the configuration, be sure to click Save or Apply. The next time your job runs, you'll see in the console log that after Jenkins completes the build (your job *does* contain a compile/build, right?), a new SonarQube analysis is triggered with the parameters you configured.

We've covered everything you need to know to integrate your Jenkins installation with SonarQube, and you've seen how to enable quality analysis in build jobs. Next we'll give you an overview of SonarQube integration with other popular CI servers.

9.2.2 Other CI systems

In general, there are no plugins for other CI servers. The fact that Jenkins is dominant has stopped development efforts on most other fronts—with one exception.

Marvelution (https://marvelution.atlassian.net) offers a stable open source plugin (https://marvelution.atlassian.net/browse/BAMSON) to integrate SonarQube with the Bamboo CI server. Bamboo is a popular commercial CI and release-management

system. It's developed by Atlassian (www.atlassian.com/software/bamboo) with lots of features and great integration with other Atlassian products such as JIRA and Confluence. Marvelution's web site offers user administration guides; so if you're already using Bamboo, take a serious look at the Marvelution plugin and open the door to the advantages of Continuous Inspection with SonarQube.

9.2.3 Best practices

So far, we've shown you how to automate your SonarQube analysis through a CI environment. But we think the puzzle is missing a piece. Usually, CI builds are triggered after every change of the source control files or at predefined time intervals (every few minutes or a couple times a day).

Because it's fair to assume that a programmer normally commits small code changes 3 or 4 times a day, on a medium-sized project with 5 developers that means the CI server builds the system 15 to 20 times a day. You do want your SonarQube analysis to run regularly, but 20 times a day is too often. Analyzing after each commit is overkill because frequent, small commits mean there are few changes in the source code and the analysis results may not change. For one thing, there's the system load consideration. The cycles required to run an analysis are usually well worth it, but it does take a while to run an analysis; and if it isn't going to show you anything new, why not save those resources?

Plus, you'll see when we look at differentials in the next section that running an analysis that you know isn't likely to include any changes is counterproductive.

So how often you should run SonarQube? Remember the following rule:

Analyze once or twice a day but only if there are changes in the source control repository.

Following this rule is easier than it may seem. For projects that build on a CI basis—after every commit—you split the analysis job off from the build job and run it nightly. (Okay, it isn't truly *Continuous Inspection*, but *regular* or perhaps *continual* inspection. Even if we have to waffle on the words, it's still incredibly valuable.) For projects that only build on a nightly basis—or a couple times a day at most—you integrate analysis into the main build job. Then Jenkins makes it easy to trigger the job (set it up to run) only when your cronned poll of the SCM indicates code changes.

Because we've told you to be choosy about *when* you analyze, you may be wondering whether you should restrict *what* you analyze. Should you include integration, acceptance, or performance tests? Should you analyze only the main language of your project, or all of them? Again, the answer's simple: make your analysis as broad as you can.

As an example, consider a typical JEE web application. The persistence layer, the business logic, and part of the user interaction logic are written in Java. The web interface may be developed with JSF pages using XHTML templates and plenty of JavaScript. Plus several modules include XML processing. So there are four different kinds of source to analyze.

If each language is segregated into a different module of a multimodule project, then you're in luck: SonarQube handles this automatically. But even if the lines in

your project aren't so cleanly drawn, you can still handle this easily. Set up a separate build step for each language (if it's a Maven project, use the post-build action for the main language and build steps for the rest). Make sure you set the `language` property appropriately for each run and differentiate the runs by either using the `branch` property or setting a different project ID for each one.

Once you've finished configuring your build jobs, your SonarQube analysis should be fully automated. You'll have good, fresh data rolling in all the time. Now we'll move on to differentials, which help you take full advantage of your regular analysis by letting you easily compare the current state to a previous analysis.

9.3 Monitoring quality evolution

Let's say you've presented SonarQube to your boss and she's excited about it. Now she wants you to measure the test coverage of a critical system for the next couple months and verify that it's trending in the right direction.

You're thrilled that she's buying in, so you scoot back to your desk to check the project dashboard. Great Scott! Last night's CI analysis shows that your project is more than 75% covered by tests! Thrilled, you make a note of the number and head out to lunch.

A few days later, you check the dashboard again. Testing coverage is around 72%. You're glad to see it so high, but you can't find the note with the previous number, so you can't tell your boss if coverage is increasing or not. This time you'll be sure to put the note in a more secure place.

Fortunately, there's a better way: SonarQube's differential views.

9.3.1 Exploring differential views in the project dashboard

To start working with differential views, go to the dashboard of a project you've analyzed at least twice (preferably on two different days). At the top of the screen is the Differential drop-down menu, as shown in figure 9.13.

If you're following along on your own instance of SonarQube, you've probably noticed that the options in *your* drop-down menu are different than what's shown in figure 9.13. And that's the power of this service. You can fully customize what you see in the drop-down menu at the global and project levels. We'll show you how in a minute, but first let's see what happens when you select a value in the drop-down list. Try picking the first option, which should be Δ Since Previous Analysis (*Some date*), and notice how the widgets change when the page reloads, as shown in figure 9.14.

NOTE The reports you see when you select a period, say Δ Over 5 days, don't change dynamically. Go a few days without analyzing, and what you're looking at will be the change over eight days—that is, five-day results that are three days stale. You'll need to analyze each day to see different numbers day by day.

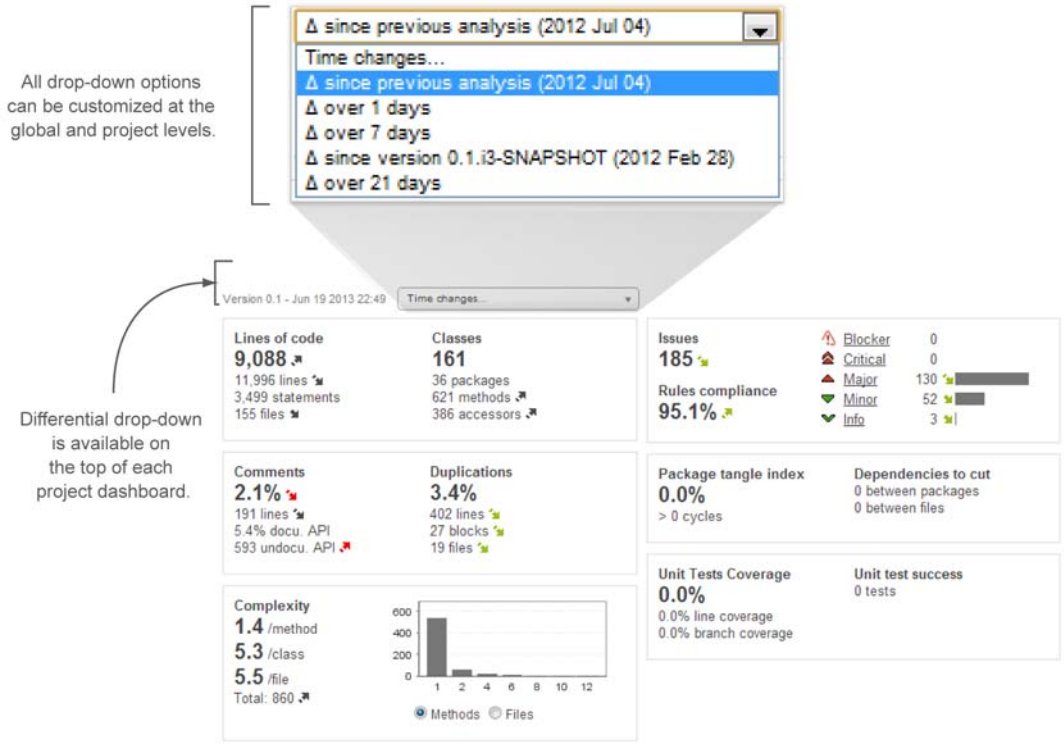


Figure 9.13 The Differential drop-down menu in the project's dashboard

If you take a closer look at the dashboard you'll notice that all the trending arrows have been replaced by colored (red, green, or black) numbers in parentheses. Furthermore, two widgets (issues and unit-testing coverage) have been enriched with new metrics. Let's examine them step by step.

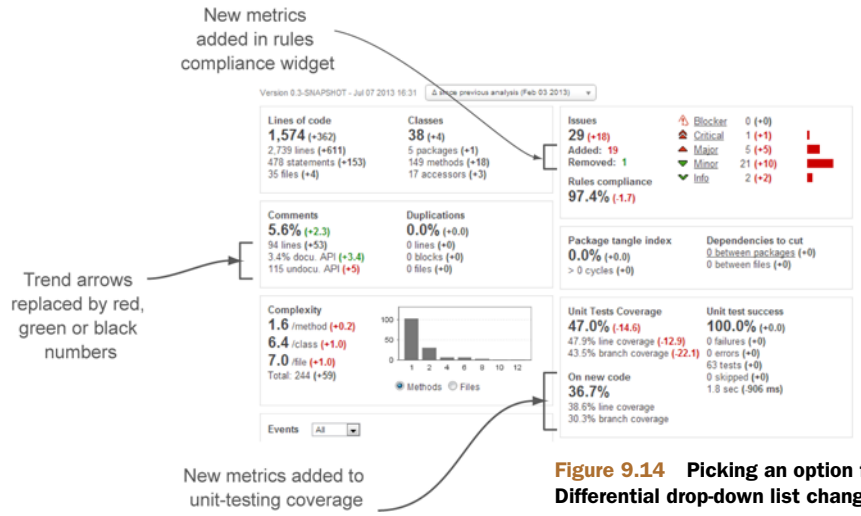


Figure 9.14 Picking an option from the Differential drop-down list changes the widgets.

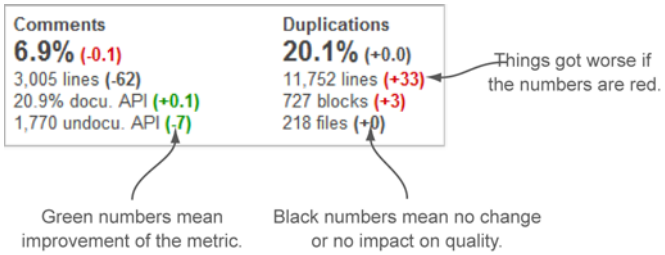


Figure 9.15 Colored numbers indicate improvement, deterioration, or no impact on quality analysis.

THE MEANING OF COLORED NUMBERS

At the right of each metric is a number showing the difference between the latest analysis and the snapshot analysis you selected in the Differential drop-down menu. We've said before that SonarQube uses colored trend arrows—green for good and red for bad. When you're in differential mode, the same applies to the numbers that replace the trend arrows. Good changes are green, bad are red, and value-neutral are black.

Look at the comments and duplications widget in figure 9.15. In this case, the density of comments has decreased from 7.0% to 6.9%, so in SonarQube, the difference is colored red. Similarly, there are 33 new duplicated lines and 3 new duplicated blocks of code. They negatively affect the quality of the project, so they're shown in red.

On the other hand, two of the documentation metrics have improved, so their Δ values are shown in green. Looking at your own dashboard, you'll see that color has nothing to do with whether the value of the change is a negative or positive number—it's all about whether the metric is improving or deteriorating. This means metrics with no change and metrics that don't affect the quality of the project, such as duplications files and lines of comments in our example, are shown in black.

NEW METRICS IN THE UNIT-TESTING WIDGET

In addition to using the same red/green coloring for differential values, the unit-testing widget adds metrics in differential mode. But to activate the feature, you need to install and configure the SCM Activity plugin we discussed in chapter 2.

Once you've done that, differential mode adds a new section to the lower left of the unit-testing widget, as shown in figure 9.16. It's titled On New Code, and it shows the three coverage metrics for code added in the comparison period.

Figure 9.16 Unit-testing widget when a differential view is enabled

Figure 9.17 **The rules compliance (issues) widget when a differential view is enabled**

Our experience has shown that monitoring this section regularly and politely nagging developers who've checked in fresh and untested code keep an application's overall code coverage at the highest levels. In other words, Continuous Inspection (and correction) works!

NEW METRICS IN THE ISSUES WIDGET

The issues widget also adds metrics in differential mode. Just below the total number of issues, you can see how many issues were added and removed in the comparison period, as shown in figure 9.17.

Look again at the differential numbers, and you may notice that the differential values on the right (for Blockers, Criticals, and so on) don't add up to the Added and Removed values on the left. That's because the numbers on the right are net changes and the ones on the left are gross values. But sum each set, and the numbers should match: (323 -326) = (-7 -2 +12 -6) = -3.

So far, we've looked at differentials on the dashboard, but they extend into some drilldowns as well. Next, we'll drill into issues while in differential mode.

9.3.2 *Differential views in the issues drilldown*

The issues drilldown you saw in chapter 2 had a few extra features compared to other drilldowns: severity and rule lists. It offers one more plus by honoring differentials too, letting you examine how many issues for each severity, rule, and so on have been added in a differential period, as shown in figure 9.18.

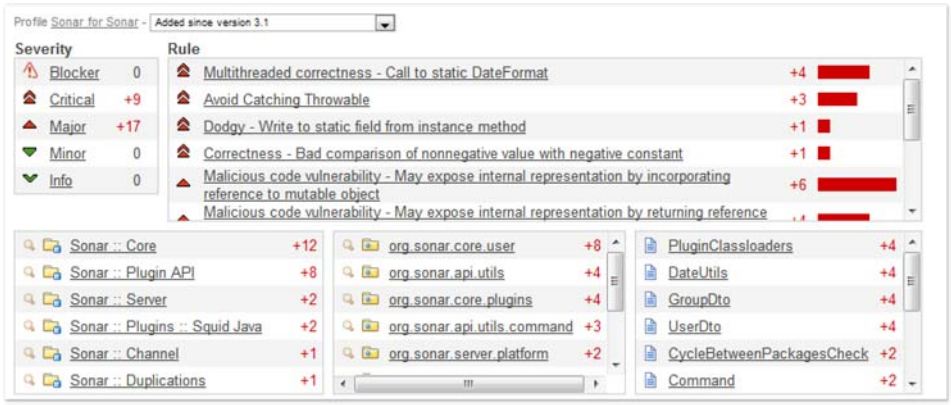


Figure 9.18 **Issues widget when a differential view is enabled**

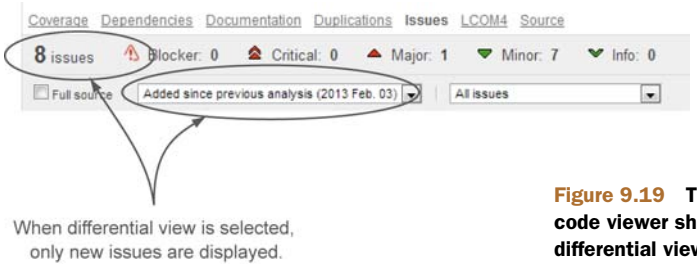


Figure 9.19 The Issues tab in the source code viewer shows only new issues when a differential view is selected.

Only new issues are shown here; existing issues are omitted. And don't look for any green in this differential view—it only shows added issues, not issues that were removed.

9.3.3 Differential views in the source code viewer

Click any filename to reach the Issues tab of file detail view. When you do so in differential mode, the header changes slightly, again to show only new issues, as shown in figure 9.19. This feature is available only when you have the SCM Activity plugin installed and enabled for the current project.

To decide which issues to show in differential mode, SonarQube applies an algorithm.

If there was an issue with the same rule, then it checks for the following:

- The issues have the same line number.
- The issues have the same line hash.
- The issues have the same message.

If at least two conditions are true, the issue is considered old; otherwise it's marked as new and displayed in the differential drilldown. As we've hinted, differentials aren't available on all tabs—just Issues, Coverage, and Source. Yes, you can use differentials to see which lines of code have been added in the comparison period!

9.3.4 Choosing differential periods

When we first showed you the differential drop-down menu, it held some values that probably don't match what's in your drop-down menu. For starters, our drop-down menu had five periods and yours probably has three, if it's a new SonarQube installation. Plus, some of our periods were different than yours. Well, now it's time to talk about those differences. As you've seen, you can put up to five values in the drop-down menu. Chapters 14 and 15 will show you how to change those values. Here we'll talk about what you might want to change them to.

The first three are set at the global level and will be the same for all projects, unless you explicitly override them at the project level. By default, the three global periods point to the previous analysis, 5 days ago, and 30 days ago. At the project level, you can add two more periods for each project. The periods are named (not surprisingly) Period1 through Period5.

SonarQube offers the following ways to define a threshold period:

- Use the string "previous_analysis" to compare the latest analysis with previous one.
- Set an integer number of days before the current analysis. Note that this can yield approximate results. SonarQube finds the first available snapshot analysis inside the date range. So if there's not a snapshot that's exactly five days old, you'll get a comparison against a four-day-old or three-day-old snapshot.
- Predefine a date in yyyy-MM-dd format. SonarQube finds the first available snapshot analysis in the date range.
- Use the string "previous_version" to compare the latest analysis with the previous project version (corresponding to the last time you changed your project's `sonar.projectVersion` analysis property).

There is no "right" definition for these periods, but here are some tips based on our experience:

- Keep "previous_analysis" as the first period.
- Decide how frequently you'll inspect your analysis results, and define corresponding periods by specifying numbers of days. For instance, we bump the default 5-day period up to 7 days so we can check week over week; and a 30-day period would give you a month-over-month comparison.
- Define a "previous_version" threshold, either on a global or (preferably) a project level.

Earlier we said that you could set your differential periods to specific dates, but we warned that you'd want to read up on events first (see chapter 15). That's because any snapshot that doesn't have an event flagged against it is subject to being deleted during housekeeping. If you *do* decide to set your differentials to dates—say January 1 each year—you'll want to not only run a starting-point analysis that day but also make sure those analyses are marked with events. Because you can set version or other events retroactively through SonarQube's interface, they're probably the best options to use.

Once you make your changes, the drop-down menu text changes immediately to reflect them, but remember that under the covers, those text values still correspond to Period1, Period2, and so on. This means that if you adjust Period2 to be seven days instead of five and immediately check your dashboard to see the seven-day differential, you'll be disappointed. Differential modes *look* dynamic, but they're not. So you won't *see* a seven-day differential until after your next analysis.

But there is a way to satisfy—at least partially—the immediate urge to see the difference between one *version* of an application and another. And that's next.

9.3.5 The Compare service

Filed under the Tools menu in the left rail, you'll find Compare. It appears at both the global and project levels and leads to an interface that allows you to put the metrics

Figure 9.20 The Compare tool lets you compare selected metrics from multiple versions—of the same project or different ones!

for specific project versions side by side for comparison. Access it from the project level, and it's automatically populated with up to six of that project's most recent versions, as shown in figure 9.20.

The Compare service doesn't do the math for you, like differentials, so you have to figure out for yourself whether code coverage increased or decreased from version to version. But unlike differentials, it lets you do ad hoc comparisons on a whim—in a project or across multiple projects.

By default, Compare shows a basic set of metrics, but you can easily add metrics to or remove them from the list. You can also easily reorder both the metrics and the projects—just mouse over the labels to reveal those controls.

At this point, you've plumbed the depths of Continuous Inspection and quality-evolution monitoring through differentials and version comparisons. Now we'll close the chapter, as we usually do, with an overview of related plugins.

9.4 Related plugins

A couple of plugins related to CI and inspection offer related metrics and utilities. Let's look at the advantages of adding them to your process.

9.4.1 Cutoff

Have you ever wanted quality reports on just the code changes made last month? Unless you happen to have tagged your repository a month ago, it seems like an impossible task. Try to figure it out manually and you'd need weeks.

And that's where the Cutoff plugin comes in. At both the global and project levels, it lets you define either a specific date, like 1/1/2013, or a cutoff period, such as 60

days, and it tells SonarQube to ignore files that haven't been touched since then. It's especially useful for teams that maintain legacy systems and want to focus only on their own changes—not what was written years ago by long-gone developers. It lets them define the date they picked up maintaining the system, and it narrows SonarQube's reporting to just the files they're interested in.

You need to be aware of a couple of things with this plugin. If both the date and period are defined, the period is ignored in favor of the date. Also, it relies on the system date. So, double-check that your system date is correctly set before you run a new analysis with a date threshold. Furthermore, checking in minor changes to an "old" file resets its date, meaning that all its legacy issues are included to your "only new" analysis.

9.4.2 Build Breaker

The Build Breaker plugin is a simple yet useful plugin with a sole purpose: to break the CI build when new alerts are raised during an analysis.

We haven't spent a lot of time yet on alerts, but the basic concept is that you can define warning and error thresholds on specific metrics. When a project's value crosses the threshold, SonarQube raises an alert, and the Build Breaker plugin lets you take advantage of your CI server's notification mechanisms to raise a hue and cry. That means you need to set some alerts (covered in chapter 13) to take full advantage of this plugin.

As an example, assume that you want to be notified if your duplications rise over 2%. All you have to do is to install this plugin and create an alert in the project's quality profile. Then, if your threshold is hit during an analysis, the Build Breaker plugin returns an error status to the CI server, and the build is marked as failed, as shown in figure 9.21.

The beauty of this integration is that, assuming you've configured your CI server to send notifications (emails, instant messages, and so on) for build failures, whenever an alert is raised on the SonarQube side, all team members are automatically notified by the CI engine, eliminating the need for manually visiting SonarQube to check for

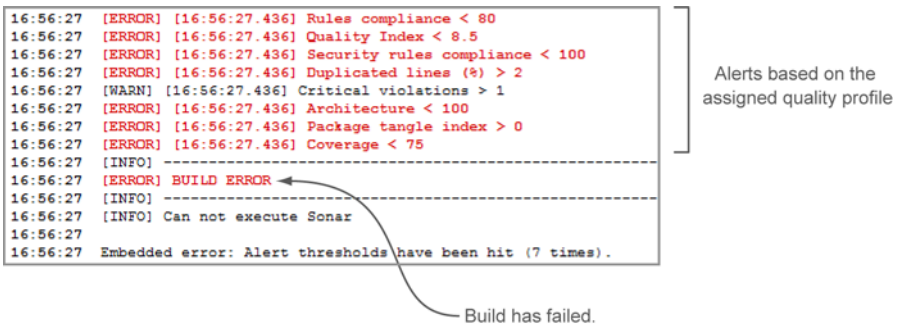


Figure 9.21 CI log output when analyzing SonarQube with the Build Breaker plugin enabled

new alerts. Even better, because this plugin is installed on the SonarQube side rather than in your CI server, it can be used with any CI server that respects a subprocess's return code.

9.5 Summary

This chapter was a real workout! First we looked at why CI and Continuous Inspection should play an important role in your development process. You saw that automating your project analysis is the basic foundation to apply Continuous Inspection.

Thanks to some free plugins, SonarQube smoothly cooperates with the most popular CI servers: Jenkins, AnthillPro, and Bamboo. You saw how to install and configure SonarQube integration for Jenkins and how to trigger analysis in a build job—including multiple languages. Best practices say that you'll set this up in a job that only runs once a day or so.

But there's no real Continuous Inspection if you can't monitor specifically how your source code quality is changing, so next we looked at SonarQube's differentials, which give you details on the changes since a given point in time. Differential views, which are fully customizable on both the global and project levels, allow you to compare the current quality status with predefined thresholds—up to five of them—such as the latest build, latest version, or a custom date. All SonarQube's core widgets take advantage of this service by showing the differences between the current analysis and the selected period. Further, a couple of them (issues and unit testing) even add new metrics in differential mode.

You also saw the Compare tool, which gives you the ability to compare any two project versions (even any two projects) on an ad hoc basis. Good differentials require a little planning, but you can run the comparison tool on a whim.

Closing this chapter, we showed you some plugins that either extend the differential views feature (Cutoff) or improve integration with CI servers (Build Breaker).

Now you know how to continuously track what's going right—or wrong—in your project. In the next chapter, we'll look at the tools SonarQube offers to help you manage your code reviews.

10 Letting SonarQube drive code reviews

This chapter covers

- Reviewing code in SonarQube
- Creating manual issues: when the rules aren't enough
- Tracking issues
- Planning your work with SonarQube's action plans

In this chapter, we'll talk about how to use SonarQube in a code review, one of the most loved/hated, revered/maligned duties of a coding team that (almost) never gets done. There are a lot of reasons for skipping code reviews, among them that there's no starting place or that people will feel picked on. But centering your code review around SonarQube can alleviate or eliminate many of the classic excuses. It will also help you organize and manage your code-quality efforts through issue confirmation, commenting, assignment, and severity adjustment. When you find the occasional dud issue (it happens once in a while), we'll show you how to handle it without tossing the baby out with the bathwater.

Once your issues are commented and assigned, you'll want to keep track of them. SonarQube provides three mechanisms: email, search, and a dashboard.

Beyond that, there's also the ability to group issues and schedule them into action plans. After we've shown you everything SonarQube has to offer on the subject, we'll finish with a discussion of the fundamental reasons to have code reviews. Reasons that address the growth and development of your team members. Reasons that existed before SonarQube came along and that will remain long after you've paid down all your technical debt (yes, it *will* happen eventually!).

We'll talk about those reasons, and we'll tell you how we've seen effective code reviews structured to achieve the goals of giving developers a safe place to talk about code—whether it's what they did yesterday that they're proud of, what they did yesterday that maybe they shouldn't be proud of, or what they would like to get the team on board for doing tomorrow.

The excuses for skipping code reviews are as varied and colorful as the people making them, but like taking the time to buckle your seatbelt, you don't have time *not* to have code reviews. With SonarQube on board, you'll find them both painless and productive.

10.1 Reviewing code in SonarQube

Let's say a coworker installed SonarQube on the spare box under her desk before the last sprint. She's been feeding it code ever since, and now that you're between sprints again, she's showing you the project's SonarQube dashboard. If your team is typical, you're staggered by the number of issues alone, not to mention the number of duplicated blocks. And you don't even want to think about the complexity scores.

You may be tempted to shoot the messenger, but you'd be better off scheduling a code review. Not the kind where two developers sit down one-on-one to go over what one of them has just done, but the other kind, where everyone is in a room together (literally or virtually). It's a semiformal meeting with a public agenda, and it's what we'll focus on here. It can be held with or without SonarQube; but *with* SonarQube will be a lot better than without, because as good as it is at finding problems, SonarQube is about more than just showing you what you've done wrong. It also provides a platform for managing solutions.

There are many ways to run code reviews and many ways you can use the solution-management tools SonarQube provides. Using the two together (SonarQube's tools in the context of a code review) makes a particularly effective combination, so that's the context we'll use to explore what SonarQube offers.

10.1.1 Issues: a starting point

One of the common reasons people give for avoiding code reviews is that there's no good starting point. Navigate to the SonarQube issues drilldown, and you're likely to find that this is no longer the case, as figure 10.1 shows.

In the issues drilldown, most teams are presented with a wealth of options for picking a starting point. The obvious choice is the highest issue severity with a non-zero

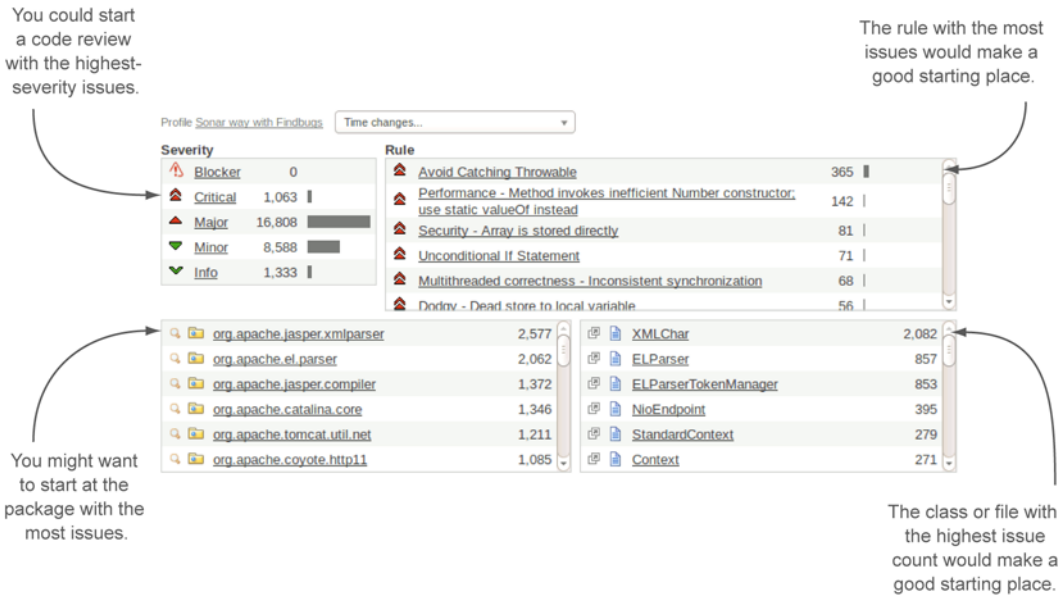


Figure 10.1 Arriving at the issues drilldown from the left-hand menu (not visible here) shows you all issues in all classes and all packages.

count (Critical, for the project shown in figure 10.1), but some might want to start in the worst package, or in the single class with the most issues.

Whatever you choose, drill down until you're looking at issues. In each issue block, you see the issue title, a little extra detail on the issue, and, if you're logged in, a row of links, as shown in figure 10.2.

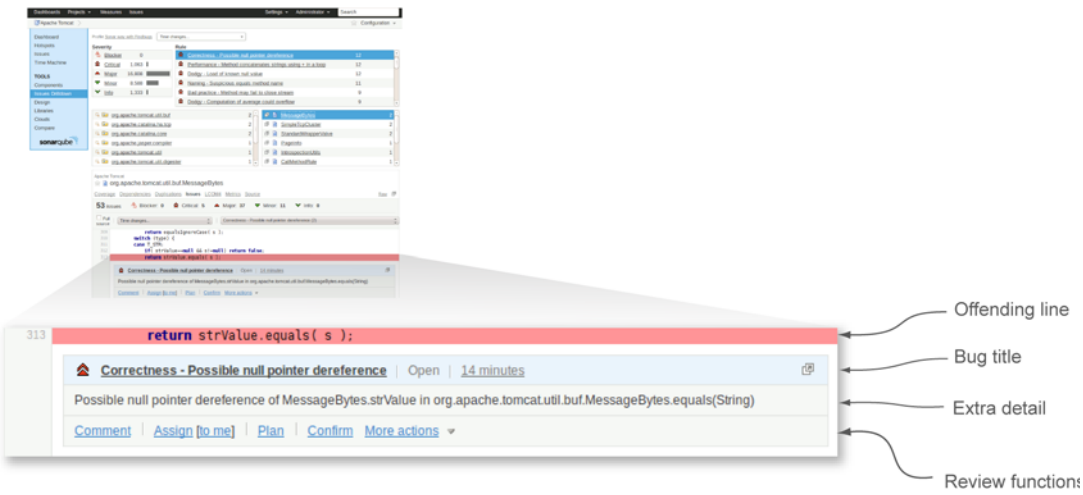


Figure 10.2 SonarQube does more than show you what the problem is. If you're logged in, it also gives you links to workflow functions at the bottom of each issue block that let you actively manage the problem.

Figure 10.3 Clicking the title of an issue adds more detail in line. Sometimes the information it offers is helpful. Other times it's just a longer version of the issue title.

Sometimes it's not obvious to every member of the team why an issue was flagged, or why it's considered an issue. Even when the problem is obvious, it's not always clear how to fix it. Clicking the issue title here adds a little more information in line: the rule description. Sometimes it's detailed and helpful, as with the one shown in figure 10.3. Other times, it's only a slightly longer restatement of the issue title (for those cases, chapter 14 will show you how to augment the description to make it more helpful).

Once everyone on the team understands an issue, you'll probably want to do something about it. At the very least, you'd probably like to mark it as reviewed, and maybe annotate it with the outcome of your team discussions. This is where the last line of the issue block comes into play. Each link there is a workflow option. For each issue, you can use SonarQube's workflow to track the notes and decisions the team makes about an issue.

10.1.2 Confirm, comment, and assign: the simplest workflow options

The first three workflow options are simple, both to understand and to use: confirmation, commenting, and assignment. The simplest option is confirmation. You use this once an issue has been evaluated and you're sure it actually is a problem. (For the cases where you can't confirm an issue, you may want to use the false-positive option, which we'll cover later.) The ability to confirm issues is particularly useful when you're dealing with a legacy application with a large backlog of technical debt. You can't address every issue right away, but at least you can keep track of your progress as you work through evaluating your technical debt. Using it couldn't be simpler: click the Confirm link. There will be a blink, and when you look again, the text will have changed to Unconfirm.

Of course, once you've taken the time to understand an issue in its context, you may want to do more than simply confirm it. You may want to annotate it with your findings. That's where the comment function comes in. It's almost as easy to use as Confirm. To start, click an issue's Comment link, and you'll see a text box added to the interface. When you submit the comment dialog, your comments will appear at the bottom of the issue block in a running log. Figure 10.4 shows both the before and after, with the comment dialog on top and the result at the bottom.

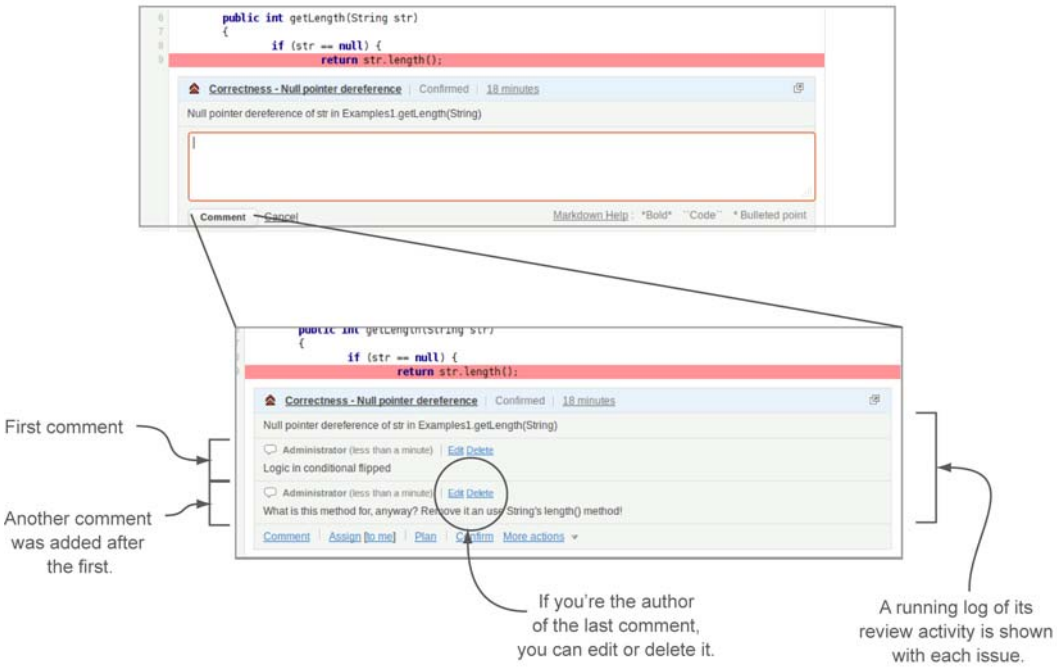


Figure 10.4 **You can make as many comments as you like on an issue. They're shown in a running log.**

You can comment on an issue as many times as you like, and each comment is added to the bottom of the log. If you're the author of a comment, you can edit or delete it.

If you're browsing through SonarQube by yourself, you might use a comment to start a discussion. In a group setting, comments are often used in a "this needs further investigation" kind of scenario. What you don't want to do is make comments in SonarQube that have lasting value to the code base; those belong in the code. When it's time to move beyond simple confirmation and commenting, you can assign an issue to any SonarQube user (we'll talk about setting up user accounts in chapter 12). To do so, click Assign in the issue block. Figure 10.5 shows the assignment form and results.

Once an issue is assigned, the name of the assignee is shown in the workflow line, and the workflow option changes from Assign to Assigned To, followed by the name of the assignee. Issue assignments appear not just on the issue itself but also in the Issues dashboard and listings covered later in this chapter. They're also viewable to the assignee through the Eclipse integration discussed in chapter 11, where you can not only see the issue in context but also do something about it!

Don't think an assignment is a permanent sentence. If priorities or workloads change, it's easy to reassign or un-assign issues. To do either, reopen the Assign dialog by clicking Assigned To and submit it using the Assign button with either a new name or no name at all in the Name field.

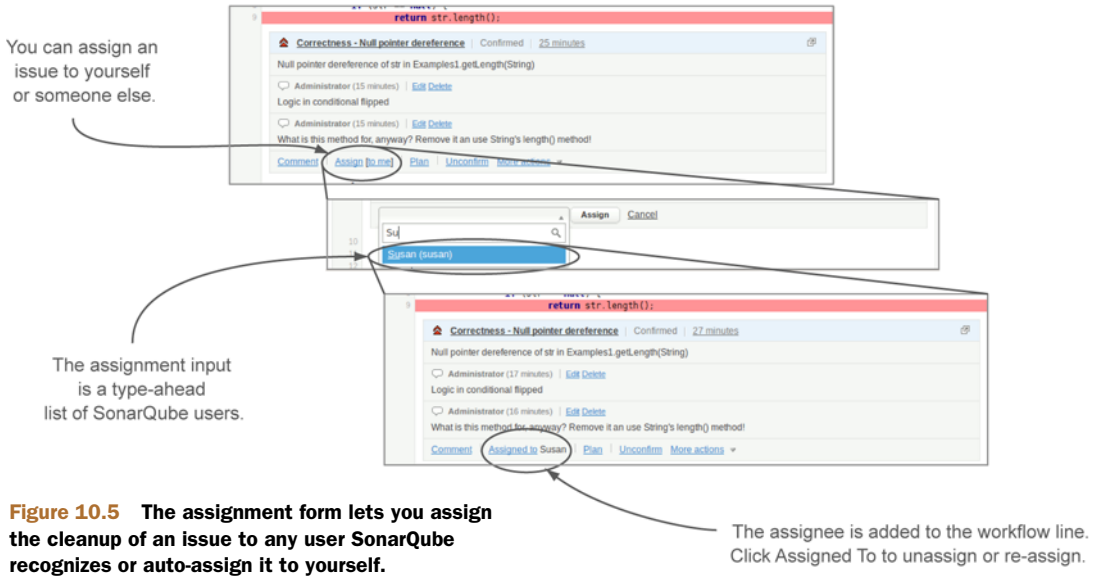


Figure 10.5 The assignment form lets you assign the cleanup of an issue to any user SonarQube recognizes or auto-assign it to yourself.

10.1.3 False positives: sometimes SonarQube gets it wrong

As you and your team work through the issues in your project, you may find some cases where SonarQube (or the rules engines it runs) just plain gets it wrong. Before you start to lose confidence in the system, know that this kind of thing doesn't happen often. There are certain situations that the rules engines aren't yet sophisticated enough to evaluate properly. But in the vast majority of cases, the things that are flagged in SonarQube as potential problems actually are problems.

The code shown in figure 10.6 is flagged as violating the Preserve Stack Trace rule because it catches one issue and throws a new one. What the rule engine missed is that the stack track of the original issue *is* explicitly preserved in line 1619.

That means the issue in figure 10.6 is a false positive. When you find one of these, SonarQube gives you a handy way to take care of it, but it's a mechanism you'll want to use sparingly. The False-Positive link tucked under the More Actions menu at the

Figure 10.6 SonarQube sometimes flags code that complies with the rules.

bottom of the issue block flips the issue status to False Positive. The issue marker stays on the page until you browse to another file, but the next time you come back to this file, the issue will be gone. (It's still in the database; you just won't see it by default anymore.)

There's no automatic opportunity to comment when you mark an issue false positive, but we highly recommend that you follow (or precede) any use of the false-positive function with a comment. A brief note about why you think it's a false positive will go a long way toward making your thinking clear to anyone who comes after you. Don't be afraid to be wordy. The more detail you give, the more your colleagues can see your reasoning and trust your judgment, and the less likely it is that you'll be tracked down to explain in person.

Commenting on false positives is particularly important when the line of code that saves an issue—the one that makes it a false positive—is different from the line of code flagged with the issue. In the example in figure 10.6, line 1619 saves the issue. But figure 10.7 shows what happens after an edit and subsequent reanalysis.

NOTE Marking an issue as a false positive makes SonarQube blind to that particular rule on that particular line until the line changes.

Originally, the issue in figure 10.6 was saved by setting the root cause from the original exception into the new exception. Remember, the code originally looked like this:

```
} catch (PrivilegedActionException pae){
    IllegalArgumentException iae =
        new IllegalArgumentException(location);
    iae.initCause(pae.getException());
    throw iae;
}
```

This "saving" line disappeared

This line was flagged with the issue

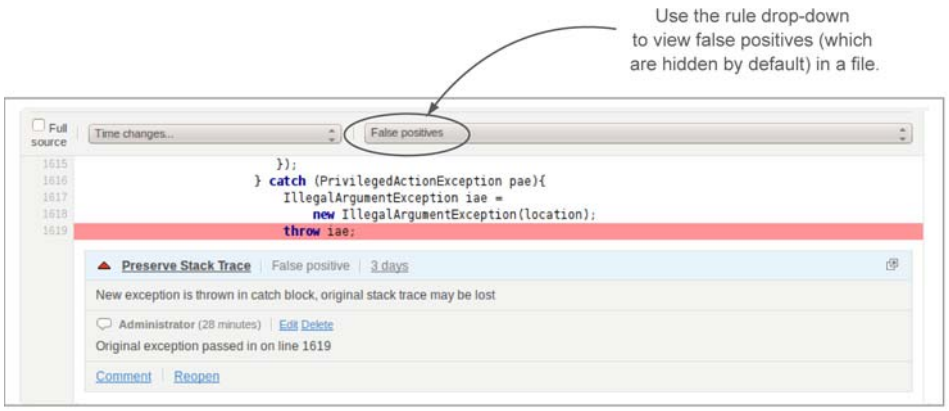


Figure 10.7 Once an issue is flagged as a false positive, SonarQube won't re-report that issue as long as the flagged line hasn't changed.

But by the time you reach the state shown in figure 10.7, edits to the file have removed that call. This is no longer a false positive.

Unfortunately, the line flagged with the false-positive issue hasn't changed, so SonarQube won't re-raise the issue. This implies two things:

- You should use the false-positive mechanism sparingly.
- You should review your false positives occasionally to make sure they haven't reverted.

Fortunately, there are a few saving graces in this situation. Because SonarQube always records the "who" and the "when" of any action performed on an issue (more on this soon), you know whom to ask about this particular false positive. And because the reviewer commented, there is *some* record of what she was thinking at the time.

Unfortunately, her comments didn't contain quite enough detail, because what's now on line 1619 certainly isn't the preservation of the stack trace. On the face of it, it looks as though the reviewer was either fogged by cold medicine or temporarily insane when she marked this as a false positive. Based on the issue's time stamp and the source control logs, she should be able to prove her sanity. But if her comments had been just a little more detailed, she wouldn't have to.

The final thing to consider in this situation is how the whole mess could have been avoided. Often SonarQube presents you with things you may see as no big deal. The situation shown here in its original state is obviously one of them. In this case, the easy way out was taken. The issue was marked as a false positive. But there was another way to address the question.

Recall that the flagged code originally looked like this:

```
} catch (PrivilegedActionException pae){
    IllegalArgumentException iae =
        new IllegalArgumentException(location);
    iae.initCause(pae.getException());
    throw iae;
```

This line preserves the stack trace…

…but this line is flagged with a Preserve Stack Trace issue anyway

A tidier way to handle the problem would have been to use the `IllegalArgument-Constructor` that accepts both a `Throwable` and a `String`. Here's the new version of the problem code:

```
} catch (PrivilegedActionException pae){
    IllegalArgumentException iae =
        new IllegalArgumentException(location, pae.getException());
    throw iae;}
```

Original exception passed in to constructor

Because in the new version the original exception is passed in to the constructor, the rules engine is satisfied that the stack trace will be preserved, and the issue goes away entirely.

It's slightly more work but far closer to bulletproof.

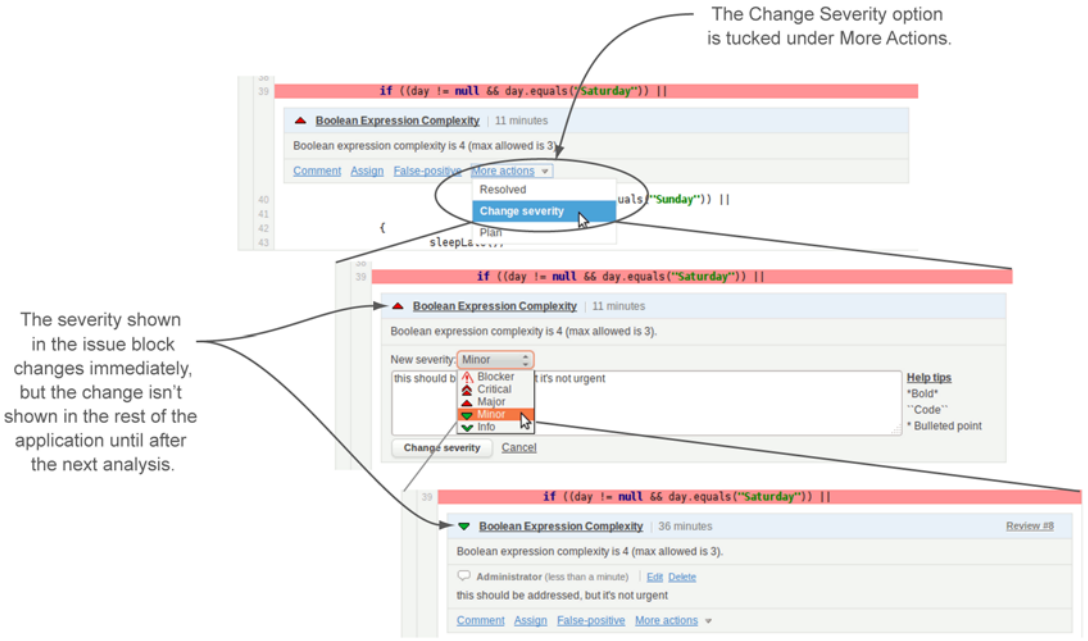


Figure 10.8 Look for the Change Severity option under More Actions. The dialog that's added presents a severity drop-down and a comment input. Once you submit the form, SonarQube confirms your severity change by immediately showing the new severity in the issue block.

10.1.4 Changing severity: not every issue is that bad

After full-out false positives, the next point of wiggle room is issues that aren't as bad as SonarQube makes out. For instance, the Boolean Expression Complexity rule checks the number of clauses in conditionals and flags anything with more than three. By default, it's a Major.

Imagine that your teammate, Joe, has checked in some changes that got flagged with (among other things) Boolean Expression Complexity. The rule's threshold is set to three. Joe's code has four.

Susan thinks you should just mark it false positive and move on, but after team discussion, you all agree that it could (and should) be simplified. You also agree that it's not of Major importance. This is where Change Severity comes in. You'll find it under the More Actions menu. Choose it, and a Severity drop-down will be added to the workflow area as shown in figure 10.8.

After you submit the dialog, the severity in the issue block is updated immediately, but the change won't appear anywhere else in the interface until after the next analysis.

10.1.5 Altering the code to make SonarQube turn a blind eye

In dealing with issues, SonarQube gives you one final tool: the `//NOSONAR` comment. SonarQube will *almost* completely ignore issues on any line that ends with `//NOSONAR`.

Why would you want that? If you deal with multiple code branches, it may be easier to hide a false positive from SonarQube with `//NOSONAR`, rather than having to mark the false positive in every branch you've got under analysis. Another reason is that sometimes there are cases where you must violate a rule. For instance, there are (poorly written?) libraries whose methods throw `Exception` itself, the raw, generic type, rather than some subtype. To use those libraries you must either catch `Exception`, which SonarQube flags as an issue, or rethrow it, also an issue.

When you find yourself in this situation, you *could* mark the issue as a false positive. That would certainly clear it out of your dashboard. The only problem is that it's *not* a false positive. You actually are catching or rethrowing `Exception`.

You could try ignoring it, but experience shows that that doesn't work well. So the next best option is to end the line that SonarQube marks with the issue with the `//NOSONAR` comment. Best practices here say that after `//NOSONAR`, you'll end the line with the reason you're having SonarQube ignore issues.

If you're coding in Java or C#, the languages that are analyzed partly with SonarQube-native code and partly with other rules engines, you may also have the option of using engine-specific markup to hide some issues from SonarQube. For instance, the PMD rule engine for Java recognizes a `//NOPMD` comment, which has an effect similar to the `//NOSONAR` comment, but it's limited in scope to just PMD. Because SonarQube uses PMD for some of its rule checking, if PMD can't see an issue, SonarQube won't see it either. Similarly, Findbugs and FxCop offer method-level annotations and attributes to turn off specific rules for that method. Again, if the rule engine can't see an issue, it won't be reported to SonarQube.

One more tool is available for Java: the `@SuppressWarnings` annotation, used at either the method or class level. You specify either `"all"` or specific rule keys, for usages like this:

```
@SuppressWarnings("all")
@SuppressWarnings("rule_key")
@SuppressWarnings({"rule_key","another_rule_key","and_so_on"})
```

Note that these aren't changes you can make from the SonarQube interface. You must return to your IDE to add this markup, then rerun analysis to see it take effect. If you're using the `//NOSONAR` comment, be aware that SonarQube will henceforth ignore *all* issues on that line, not just the one it's currently flagging. So this is another powerful tool that should be used with a light hand.

NOTE Use `//NOSONAR` sparingly because it makes SonarQube ignore all issues on a line.

Earlier we said that the `//NOSONAR` comment makes SonarQube *almost* completely ignore issues on the commented line. There is one rule available, although it isn't included in any of the provided profiles, which counts instances of `//NOSONAR`. Its default severity is Info, and it gives you a way to track these uses. See chapter 13 for a discussion of rule profile management. All the methods in this section will suppress

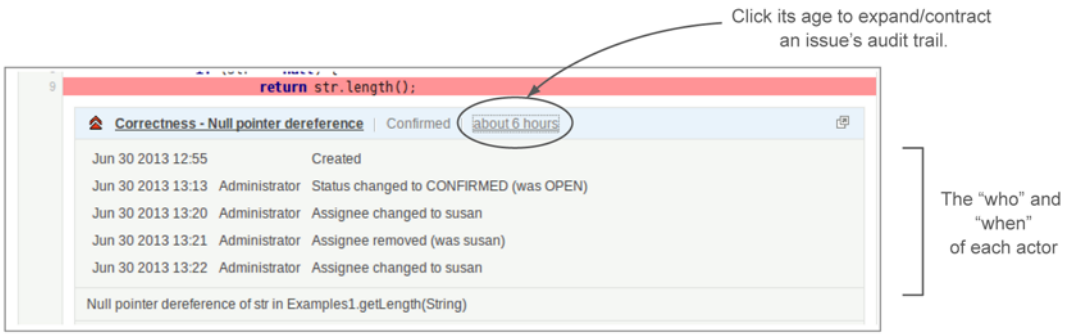


Figure 10.9 Tucked under the issue age, so it doesn't clutter the interface by default, is the issue's audit trail. Here you can see what actions have been performed on an issue, when, and by whom.

issue detection, but the `//NOSONAR` comment is the only one that's easily trackable, so it's the one we prefer.

10.1.6 Viewing the audit trail

We mentioned earlier that SonarQube records every action performed on an issue. Now it's time to show you how to find that audit trail. You may have noticed that SonarQube displays each issue's approximate age in the top line of the issue block. Click the age, and you'll see everything that has happened to the issue since it first appeared in SonarQube, as shown in figure 10.9.

10.2 Creating manual issues: when the rules aren't enough

SonarQube gives you a number of mechanisms to manage the code issues it finds. As good as they are, the tools that SonarQube uses to find anti-patterns in your code will never cover all the bases. Even if they did, you'd probably still find that you wanted to use SonarQube's issue mechanism to track things that don't break coding rules, like duplications you've decided to target for immediate elimination, or parts of the API that need commenting, or just everyday refactoring.

Because the SonarQube issue mechanism allows you to track work on your code base almost directly *in* your code base, it has a compelling immediacy—the kind that makes you want more. That's why SonarQube lets you create manual issues.

10.2.1 Why you would want extra issues

Returning to our imaginary code review, it's progressing nicely. You've worked through a good number of your top-severity issues, and the team is starting to settle into it when you stumble across a method signature that looks like this:

```
public String removeFromInventory(String userId, Date requestDate,
    String departmentId, String organizationId, String itemId, int qty,
    String reason, float price, int discountPercentage);
```

Once the team finishes discussing the benefits of request objects versus long tangles of arguments, you all agree that this should be addressed. You've gotten into the swing of

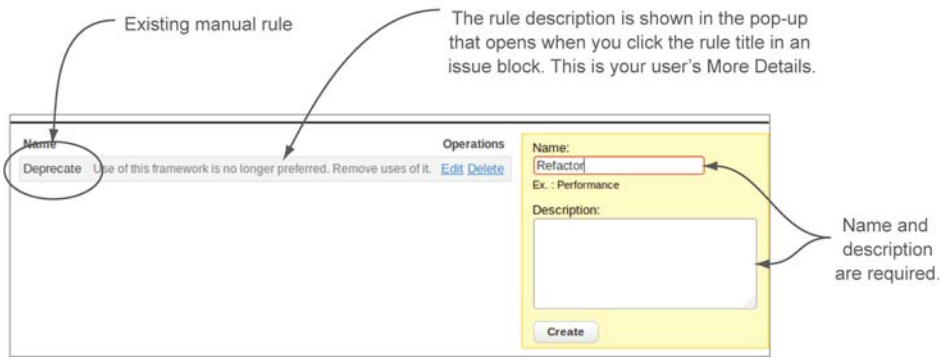


Figure 10.10 Manual rules consist of a name, and a description, which is used in the pop-up you get when you click the rule title in an issue block.

assigning issues and would like to do that here. But…it doesn't break a rule. There is no issue to assign.

Fortunately, SonarQube allows you to mark your own issues on the fly. These home-grown issues will appear in all the same places and act in all the same ways as the automatically generated ones. Beyond that, they also give you the ability to manage something else: your test code. You won't see SonarQube automatically flag issues on your unit tests, but you *can* create manual issues on them. So with manual issues, you have the flexibility to manage *all* your quality concerns through the same mechanisms.

10.2.2 Making manual issues

To begin marking manual issues, you'll first need manual rules. SonarQube administrators can create them through the Manual Rules management screen (Settings > Configuration > Manual Rules), which is shown in figure 10.10.

Enter the name of your new rule and its description, and click Create. Once the rules are in place, anyone who can use the issue workflow functions can create a manual issue. The way to start is by clicking the plus sign that pops into the left column as you mouse over code. Figure 10.11 points it out. When you do, a familiar-looking form is added to the interface, as shown in figure 10.12.

In the manual issue form, the rule and comment are required. Once this form is submitted, you've got a new issue that looks and acts just like any other; you can perform all the additional workflow functions you'd like on a manual issue.

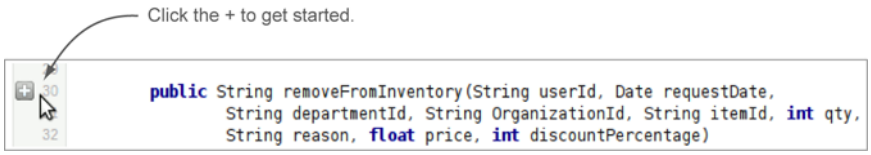


Figure 10.11 To get started with a manual issue, click the plus sign that seems to follow your mouse in the left column of views that show code.

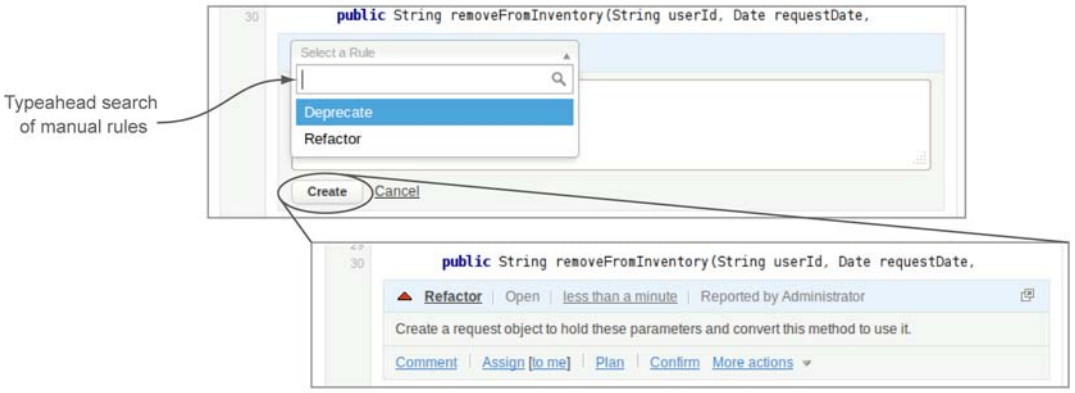


Figure 10.12 The manual issue form lets you choose the rule to use and requires you to fill in a comment. Once entered, manual issues looks just like their "automatic" siblings.

10.3 Tracking issues

Now that you've confirmed, commented, severity-adjusted, and assigned out all those issues, you might be wondering how you're going to keep track of them all without having to re-find each one in the issues drilldown every time you have a question about it. Fortunately, SonarQube provides three mechanisms for keeping an eye on issues.

The first mechanism is your inbox. Individual users can choose to be notified by email of changes to issues they created or which are assigned to them. It's off by default, but turning it on is pretty simple. Choose Configuration from the top bar, then My Profile in the left rail, and turn this option on under Notifications.

The other two tracking mechanisms are the public interfaces that this section covers. At the global level, there's a cross-project issue search; and at the project level, there's a dashboard that's dedicated to issues.

10.3.1 Life cycle of an issue

Before we go any further, it's useful to understand the life cycle of an issue. There are five possible statuses in the life of an issue: open, confirmed, resolved, reopened, and closed. Issues that are open, confirmed, or reopened are considered *active*. Issues that are resolved or closed are *inactive*. Figure 10.13 shows the status transition diagram for issues.

Basically, once opened, an issue can only be closed by SonarQube. That happens when a subsequent analysis shows that the underlying cause has been eliminated. In the meantime, a developer might mark an issue as resolved (using the Resolve workflow option in the More Actions menu or via the IDE integration we'll discuss in chapter 11), indicating a belief that he has fixed the problem. If he's right, the next analysis will close the issue. If not, it's reopened. If you notice before the next analysis

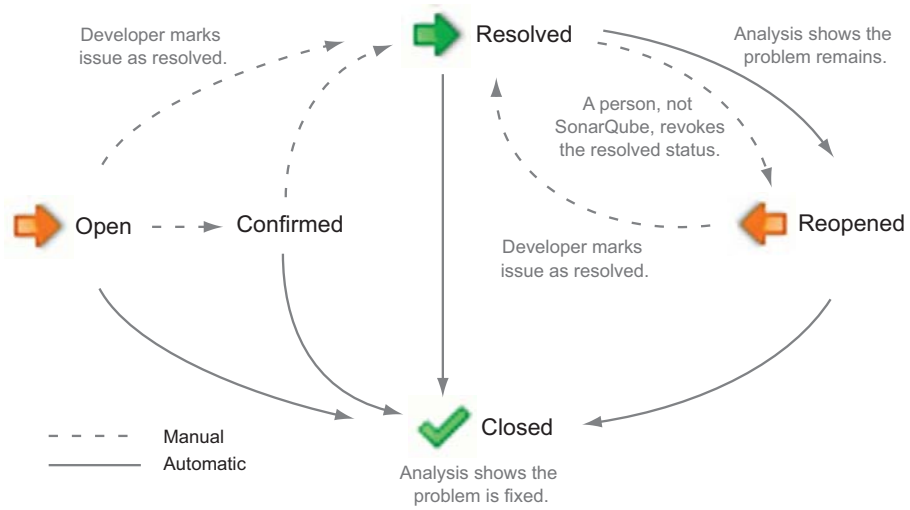


Figure 10.13 The status-transition diagram for most issues

that your resolve-happy developer is wrong about his fix, you have the option of *manually* reopening the issue.

Inactive issues have one of three resolutions: false positive, fixed, or removed. Fixed issues are ones that have either been marked resolved, or ones that have been closed by SonarQube. Removed issues are ones associated with a rule that's no longer in the profile the project was last analyzed with. There is one exception to the issue life cycle you've just seen: manual issues.

MANUAL ISSUES

Manual issues are never closed. SonarQube relies on its underlying rules engines to tell it when an issue has gone away: when to close an issue. But because a manual issue wasn't raised by a rules engine, no rules engine can put it to rest, as shown in figure 10.14.

The best that can happen is that you (or the assigned developer) manually resolve the issue, just as you manually created it. Like other issues, it can also be manually

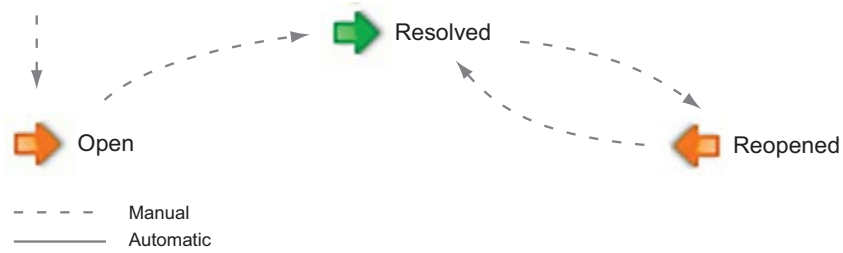


Figure 10.14 Manual issues are never closed because there's no rules engine that can "rule" on them.

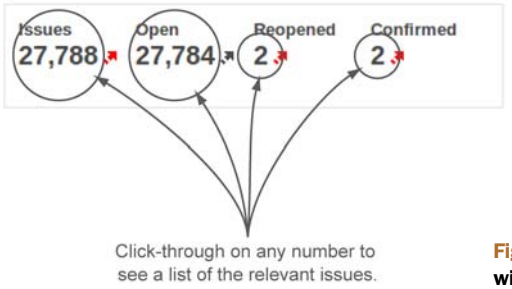


Figure 10.15 The unresolved issues by status widget gives an overview of all active issues.

reopened. The upshot is that if you decide to use manual issues, you may need to do a little extra policing to keep your projects tidy.

ISSUES DASHBOARD

We mentioned earlier that SonarQube provides two levels of visibility into issues in aggregate: the project level and globally, across projects. At the project level, you can use the Issues link in the left rail to reach the Issues dashboard. Whereas the main dashboard is dedicated to helping you track your project's quality metrics, this dashboard is devoted to tracking your work on improving those metrics.

UNRESOLVED ISSUES BY STATUS WIDGET

The first is the unresolved issues by status widget, which is shown in figure 10.15. This widget gives an overview of active issues: basically, the work left to be done. You can click-through on any of the numbers to be taken to a list of the relevant issues.

Some of the widgets on this dashboard update as you make changes to issues, but this particular one doesn't. The numbers in this widget are considered metrics, and metrics are updated only during an analysis. So don't be alarmed if you work through reviewing a big batch of issues only to see these numbers unchanged when you check the Issues dashboard. SonarQube hasn't lost your changes; it just hasn't updated yet.

UNRESOLVED ISSUES PER ASSIGNEE

The unresolved issues per assignee widget gives issue assignment counts per developer. Click-through on a developer's name to see a list of the particular issues assigned to her. These numbers are updated on a live basis, not during analysis.

MY UNRESOLVED ISSUES

The my unresolved issues widget shows a live list of issues. This time you get issue severity, title, and age. Figure 10.16 shows the widget.

This paginated widget is limited by default to five issues per page, but that number is configurable. See chapter 14 for more detail. You can click-through on the issue title to see an issue summary much like what you get in-line in the issues drilldown, as shown in figure 10.17.

Almost universally throughout the SonarQube interface, you can click the title of an issue to get this pop-up. The one exception is in the issues drilldown, where you're

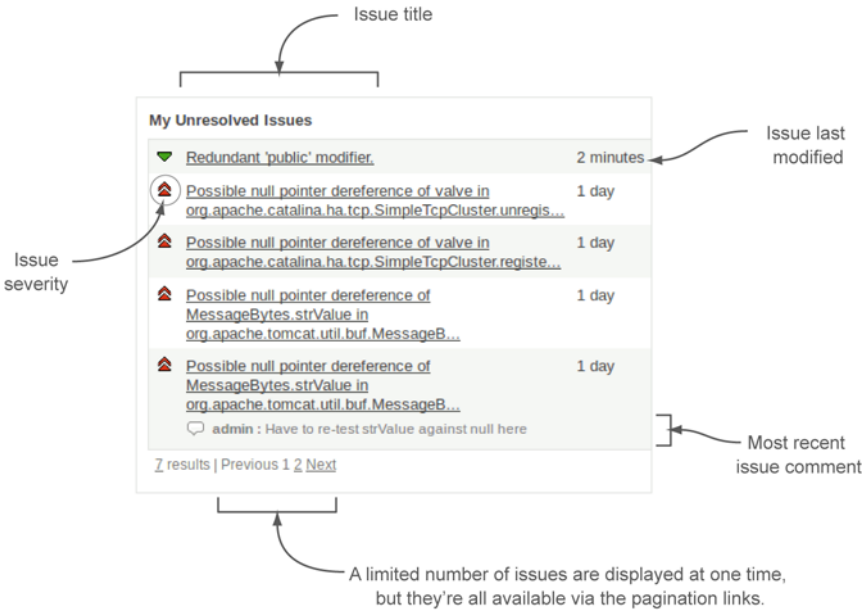


Figure 10.16 The my unresolved issues widget shows issue severity, title, last modified date, and most recent comment on the first five issues assigned to the logged-in user.

basically already looking at this presentation. There the "issue title" is actually the title of the rule the issue breaks. In that context, clicking the title adds detail about the rule to the issue block, as you saw earlier.

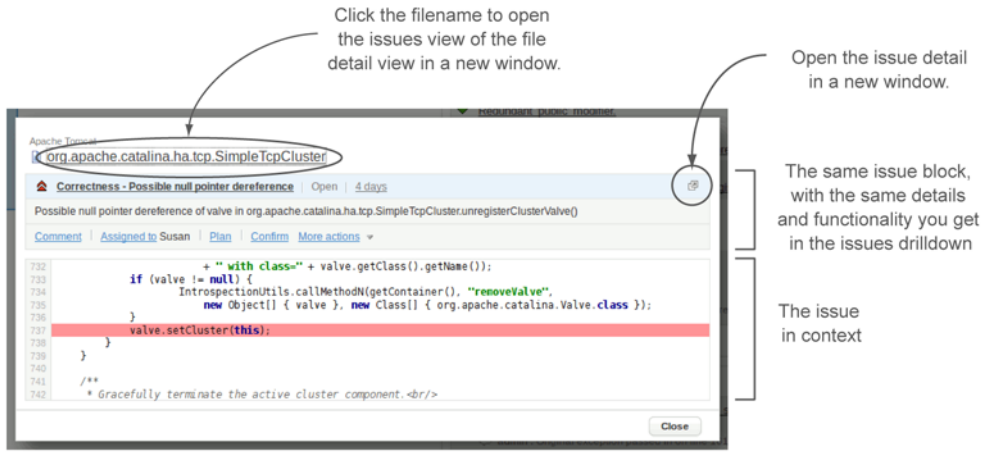


Figure 10.17 Clicking the title of an issue displays an issue summary presentation very like what you see in the issues drilldown. It doesn't pop up in a new window, but rather as an overlay in the current page. You can close it using either the Escape key or the close button at lower-right on the screen.

10.3.2 Tracking squashed issues

Earlier in this chapter, we looked at three different ways to hide or moderate an issue other than fixing it: marking it as false positive, changing the severity, or hiding it altogether with the `//NOSONAR` comment. Each of these methods is powerful and potentially dangerous, and ought to be monitored. Fortunately, the tools exist to do that.

FALSE-POSITIVES WIDGET

The false-positives widget, which is already on your Issues dashboard, is nearly identical to the my unresolved issues widget. It tracks what you would expect—false positives—and gives you a quick way to drill down to the details.

For the other two methods, you have to install plugins and add their widgets to your dashboard. Neither of those operations is hard, and the results are worth the effort.

TAGLIST WIDGET

The taglist widget allows you to track instances of the `//NOSONAR` and `//TODO` comments.

The `//NOSONAR` comment you've seen. The `//TODO` comment reminds the developer that more needs to be done in some area of the code. Some IDEs automatically insert `//TODO`s into auto-generated methods; and because most Java IDEs offer TODO-list tracking, some coders use it manually to set reminders.

You have to add a rule to your profile for each of these tags. Once you do that and add the widget to your dashboard, your next analysis should show you something similar to figure 10.18.

The taglist widget offers click-throughs to the files that hold the flagged comments. Drill down here, and you land at the file's source view, where you can use your browser's find functionality to jump to the comment of interest. Or, because you had to turn on issues to be able to make these counts, you could click over to the file's Issues view and use the issues drop-down menu to zip right to the comments you'd like to see, as shown in figure 10.19.

MANUAL SEVERITY WIDGET

The manual severity widget is one of the widgets offered in the Widget Lab plugin. To use it, install the plugin and add the widget to your dashboard. It immediately begins showing any issues where the issue severity was adjusted.

Figure 10.18 The taglist widget tracks `//NOSONAR` and `//TODO` comments, offering click-throughs to the files involved.

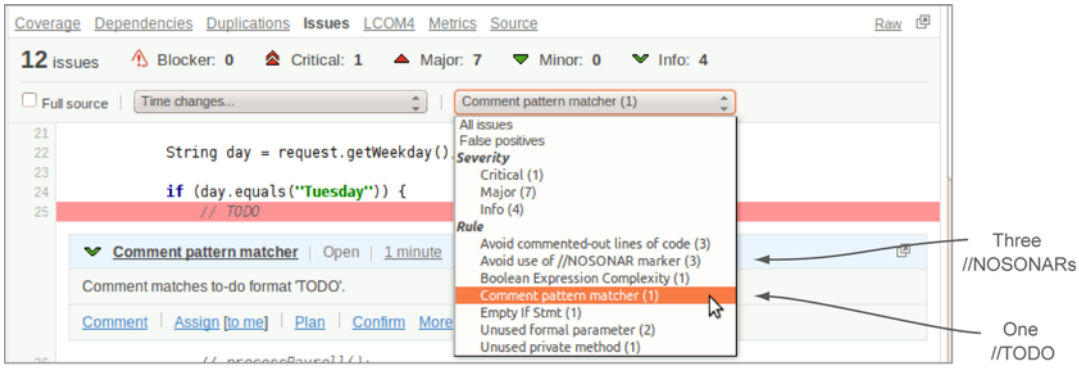


Figure 10.19 The taglist widget's underlying mechanism is a pair of issues, so you can either search the Source view for the relevant comment or flip over to the Issues view and use the drop-down menu to see the flagged comments.

It looks much like the other issues widgets you've seen, with one exception. Instead of showing just the issue's current severity, it shows both the original and current severities, as shown in figure 10.20. At no other place in the SonarQube interface can you see the original severity of an adjusted issue, or even get an indication that the severity was adjusted.

10.3.3 Searching issues

From the Issues dashboard, you can easily track the active issues in a project and keep an eye on the squashed issues. But if you want to see issues across projects, or non-active issues, you'll need to use the issue search.

To run a search, start from the Issues link in SonarQube's top menu bar. The search interface lets you choose which project to search in, which severities to show, and which statuses and resolutions to search against. You can also choose to search by creator and assignee.

The Project, Assignee, and Reporter fields work as type-ahead user lists, just like in the assign interface. Once your results are shown, you can again click an issue title to get to the issue detail pop-up. Now you can find all your issues, but how do you manage them? We'll look at that next, with action plans.

Figure 10.20 The manual severity widget shows both current and original severities.

10.4 Planning your work with SonarQube's action plans

Most teams are faced with hundreds, if not thousands, of issues after an initial SonarQube analysis. Obviously, you'd like to fix them all. And ideally it would be done yesterday, but that's not practical. SonarQube's action plans help you structure the work.

10.4.1 Why bother with action plans?

Let's return briefly to our code review scenario. You walked out of your first code review feeling pretty good about what the group accomplished. You worked through reviewing and assigning quite a few issues, and as a bonus, you caught some things that SonarQube didn't flag but that need work anyway.

Now, as you start week two's code review, there's grumbling. Most people didn't have time to work on all the issues they were assigned last week, and you're going to assign more? Plus, the commits over the past week introduced new issues, and some folks are saying that eradicating them should be top priority, while others say you should concentrate on the legacy Blockers first. To head off a full-scale war, you decide to take a different tack and create some SonarQube action plans. Then you can decide as a team which new issues to fix immediately and which you'll need to live with for a while.

SonarQube's action plans allow you to group your issues and segment the work into phases, with the option of a due date per phase. Once the issues are grouped, action plans also help you track progress.

You can link assigned or unassigned issues to an action plan, so this week the team agrees to skip the assignments and use the code review meeting to pick which issues to link to the Phase 1 action plan. Then anyone who has free time next week can use the action plan like a job jar and grab something out of it to work on. When they do, they'll have permissions to assign the issues to themselves, so there shouldn't be any frustration about two people working on the same issue at one time.

10.4.2 Managing action plans

Before you can assign issues to an action plan, you need to create it. This requires project administration rights and isn't something that can be done on the fly.

Start from the Action Plans link in the Configuration menu. Figure 10.21 shows SonarQube's action-plan management interface.

The action-plan metadata is simple: a name and, optionally, a description and due date. Each of those three elements is editable, and the edit mechanism is startlingly simple. Click the Edit link for an action plan, and the Create form becomes an Edit form with the plan in question prefilled. This is worth mentioning because it happens without a page reload. So if you're not watching, you could sit around for a while waiting for your edit form to render, like we did the first few times.

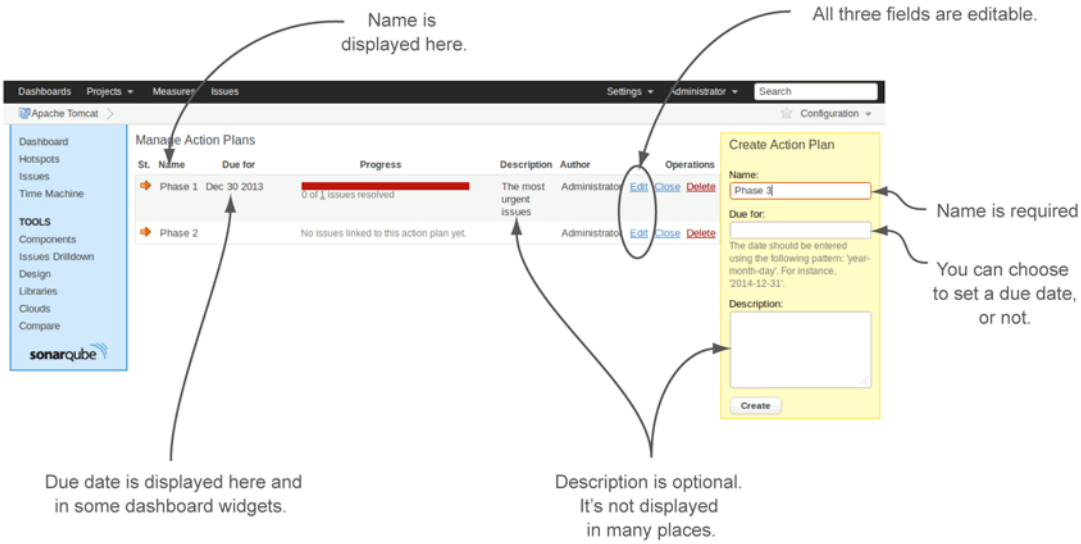


Figure 10.21 The management interface for action plans presents a list of existing action plans on the left and an action-plan creation form on the right.

Once you have issues linked to a plan, which we'll cover in a minute, issue totals and a colored progress bar appear in the Progress column, as shown in figure 10.22. Both the issue totals and the progress bar are linked to lists of relevant issues.

10.4.3 Using action plans

Now that you've created your action plans, you can begin assigning issues to them. You've probably already noticed the Plan option in the workflow section of the issues block. Choose it, and you're given the ability to pick the action plan to assign. After you've linked the issue to a plan, the plan name is added to the issue block, as shown in figure 10.23.

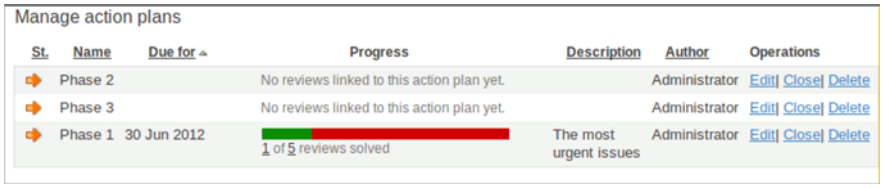


Figure 10.22 When you link issues to an action plan, a colored progress bar is added to the interface. The progress bar and accompanying issue totals are linked to lists of the relevant issues.

Figure 10.23 Once your action plans are created, you can assign issues from the More Actions menu in the issues block. Choose the plan to use, and optionally add a comment.

Plan assignments aren't irrevocable: you can choose a different plan at any time. Alternately, if you want to un-plan an issue, go back to the Plan interface and click the Unplan button that's added to the form when the issue has already been planned.

10.4.4 Tracking action plans

Once you've assigned a few issues to action plans, go back to the Issues dashboard, and you'll see that SonarQube helps you track plan progress with the open action plans widget. The widget, shown in figure 10.24, displays a progress bar for each action plan. The More link goes to action-plan administration and appears only if you have plan administration rights.

10.5 Structuring a code review

We've spent quite a bit of time on how to make the most of SonarQube in the context of a group code review. But if you're not already holding regular code reviews, it may feel like we've put the cart before the horse. If your team doesn't make a habit of sit-

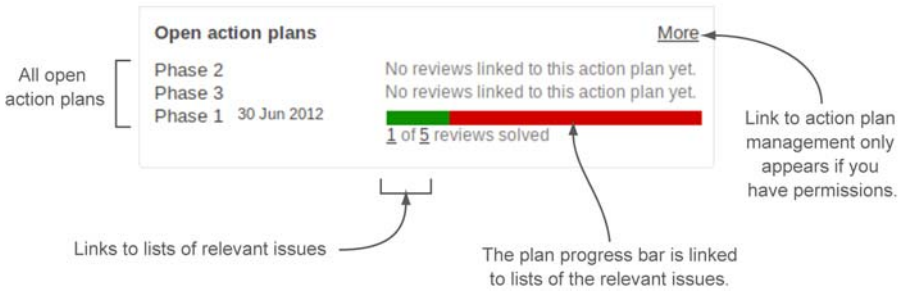


Figure 10.24 The open action plans widget gives you an at-a-glance status of all your open action plans, with links to lists of the relevant issues.

ting down together to discuss code as a group, then you may be wondering how to bootstrap the process. Good code reviews can result in better cohesiveness both among team members and within the code base, but they require a little planning. Next we'll give you the optimal answers to the basics: why, who, when, where, and how.

10.5.1 Why: talking about code

First and foremost, code reviews are important because they may be the only time your team has to sit down together and talk about the code. If your team is geographically diverse, then you may already be aware that more group time is needed on the code. On the other hand, if your team sits together, you may think there's plenty of time to talk—folks sure seem to talk about weekends and spouses and sports over the cube wall. But while folks are talking about those things—and that chatter is more important than you may think for team relationships—they're not talking about the code.

Even when they're talking about code, it's probably only in pairs or small subsets of the team as a whole. And in those conversations, information is being conveyed, or decisions are being made, that the rest of the team may not be privy to—even when they should be.

Setting time aside to talk about the code lets your team come together on things like these:

- How to tackle thorny problems in the code
- How the code is structured and where to find things
- Whether and how the structure should change
- The best ways to address major new requirements
- How to address technical debt

There are good reasons to do code reviews, but you'll also hear a number of reasons *not* to do them:

- We don't have time.
- They're boring.
- We just end up arguing about style.

- There's no need—we don't have anything to talk about.
- There's no starting point but to open a file at random and see if we find problems.
- People will feel picked on.
- People will feel left out.

The list could go on, but you get the point. As you're probably aware by now, many of these things can be addressed directly by SonarQube. The rest can be handled with a little planning and structure.

10.5.2 Who

For a code review to be truly effective, every coder on the team should be included and involved. If no one works remotely, this should be fairly easy to arrange. But if your team is geographically diverse, you'll want to pull in the remote workers via phone and web conferencing.

You want all the programmers in the room, but the folks you don't necessarily need there are the managers. This may not go over well—especially when the code-review concept is still new. In that case, let the managers in, but ask them to remain inconspicuous. Remember, code reviews are a place for the coders to talk about code, and not everyone is comfortable speaking up in front of management.

Not only that, but it won't take many iterations of having to explain low-level technical details to managers who haven't coded since punchcards to sap all the energy from the room. Make them justify every technical decision, and many coders will go so quiet that the Moon would seem noisy.

But that brings up another point. If the managers are locked out, who's running the meeting? Code review should be run by one of the team's senior developers. If there's a choice, the optimal pick is the one who's both even-tempered and experienced at running meetings. If you can only have one of those things, go for even-tempered.

10.5.3 When

Code reviews should be held regularly. An hour once a week is a good interval, but depending on how your team works, you may find that once or twice per sprint is best for you. The main thing is to develop the habit and stick to it—even after you've cleaned up all your issues in SonarQube, eliminated your duplications, and simplified your complexity.

10.5.4 Where

It's important to emphasize that a code review meeting is just that—a meeting. And like a formal meeting, it should be held in a conference room or some other place that's separate from the normal routine. Unless your team is only two or three people, just having everyone pause at their desks and join the conversation guarantees that you'll lose someone to a phone call or IM or the piece of code she was interrupted on that's frankly more interesting than one more boring meeting.

Ideally your conference room is equipped with a door and a speaker phone (if you have folks who work remotely). Another key piece of equipment is a projector. Rather than having each person follow along on his or her laptop, you should project what you're talking about onto the wall to help keep attention focused on the group's topic and minimize distractions like email.

10.5.5 How

Code reviews should be run with enough formality that people respect the process, but not so much that interesting discussion is squashed. What that seems to boil down to is that you publish a minimal agenda beforehand and keep minutes as you go along. You may wonder why we're advising you to keep minutes when SonarQube offers such great mechanisms for making notes on code. There are definitely some decisions you'll want to use the SonarQube issue workflow to record, but others belong in higher-level documents. Not every decision the team makes can or should tie to an issue. For instance, sometimes you'll discuss sweeping questions that shouldn't be tacked to a single file.

Your agenda and minutes should be posted publicly and available to everyone on the team. In this case, "everyone" *does* include management—they'll feel better once they see from the minutes that you're being productive and talking about code rather than gaming or plotting against them. A wiki is a great place to handle these formalities. You can easily set up templates, and everyone can make contributions or corrections, with changes tracked automatically.

In general, a good code-review agenda consists of a few fixed items, such as reviewing the action items from the last meeting and the changes in SonarQube since then, plus time for the discussion topic queue.

The *discussion topic queue* is a rolling list of code-review topics that anyone on the team can add to. When the poster's topic rolls around, he has the stage. The kinds of things that end up on a queue like this vary widely, from "Cancel button on the left or right?" to "Demo & Proposal—JSR 303 Bean validation." Neither end of the spectrum is more important than the other, and each item deserves serious consideration for several reasons:

- It's important—at some level—to someone on the team.
- Open discussion gets all views aired.
- Agreement here leads to consistency across the team and the code base.

The last point to consider in structuring how your meetings will run is the occasional need to call on the one in the corner. You know, the coder who does great work but who would be just as happy never to open her mouth. Be sure to call on her occasionally during the discussion. Nothing too formal, just "Sam, what do you think?"

Once she sees that no one's going to bite her head off or laugh in her face, she may be more inclined to speak up on her own. And then, when you schedule her onto the agenda to show what she's been working on (you know she'll never schedule herself), speaking to the group is that tiny bit easier because she's done it before.

By the way, just because you need to provide both an agenda and meeting minutes, that doesn't mean you can't satisfy both requirements with one document. Start with a rudimentary agenda with the fixed items and the discussion topics you think you'll get to, and then edit it during the meeting to show who attended, which topics were covered, and any decisions that came out of the discussion. File it by date afterward, and when the meeting's over, you're done except for carrying out your action items.

As the team (and management) gets more comfortable with the process, you might decide to boil some of the formality away, dropping the agenda and minutes. If you decide to do that, we urge you to retain the discussion topic queue so everyone on the team has the opportunity to contribute and to know what's coming from the rest of the team.

10.6 Related plugins

Usually, at this point in the chapter, we tell you about the available plugins that relate to the chapter's topic. In this case, we've already discussed two of the three relevant plugins in the chapter itself, so we'll keep the treatment of them brief. We'll start with the one you haven't seen yet, the JIRA plugin.

10.6.1 JIRA

If you use the JIRA issue-tracking system, then you'll be excited to know that this plugin offers two significant points of integration between SonarQube and JIRA. The first is the ability to track your issue counts as project metrics. With each analysis, this plugin logs in to your JIRA instance to retrieve issue counts by priority. They're shown in the widget provided by the plugin (shown in figure 10.25).

The other integration point, and the reason it's included in this chapter, is the ability to create a JIRA issue ticket for an issue from the More Actions workflow menu, as figure 10.26 shows.

The comments you enter are used both as issue comments and in the text of the JIRA issue. The SonarQube issue and the JIRA issue contain references to each other for easy cross navigation, but that's where the relationship ends. Resolving the issue won't have any effect on the issue or vice versa.

10.6.2 Taglist

We've already mentioned the taglist plugin. It gives you a way to track and count comments. Here we'll go into the details that we omitted earlier. Typically, the comments

Figure 10.25 The JIRA widget shows your project's issue counts, both in aggregate and by priority.

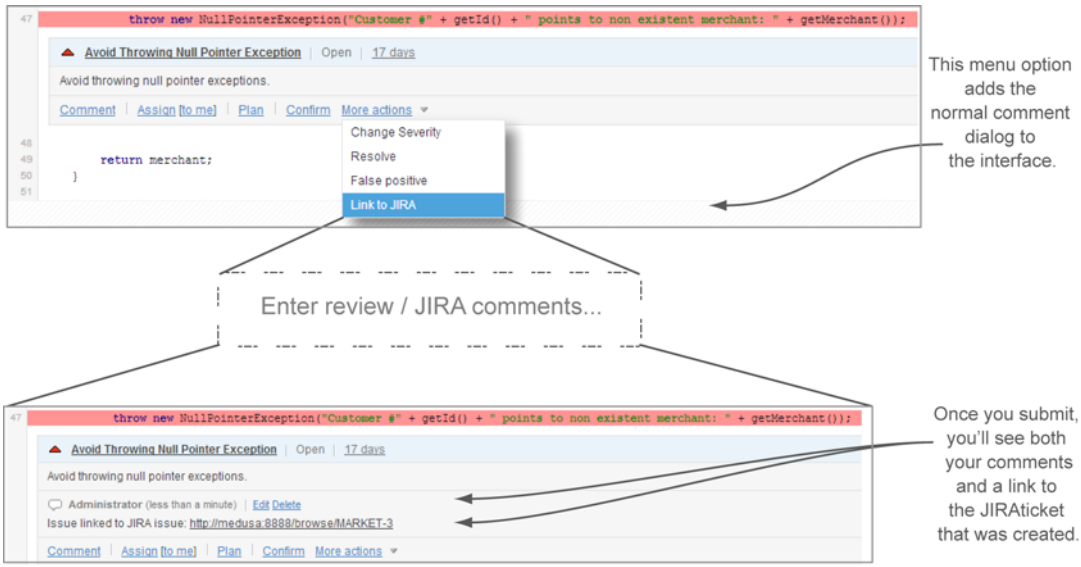


Figure 10.26 **Once you've installed and configured the JIRA plugin, look under More Actions to find the option to add a JIRA ticket from SonarQube. When you submit, you'll see a link to the ticket in the issue comment.**

you want to track are `//NOSONAR` and perhaps `//TODO` comments, although you're not limited to those two.

Once you've installed the plugin and added the taglist widget to your dashboard, you also need to turn on two rules: the "Avoid use of //NOSONAR marker" rule, which comes from SonarQube; and the Checkstyle rule engine's "Comment pattern matcher." You can have as many instances as you like of this particular rule, each one looking for a different comment; chapter 13 will show you how to make those copies. Figure 10.27 shows two variations: the expected `TODO` matcher and one that looks for `TESTER` comments.

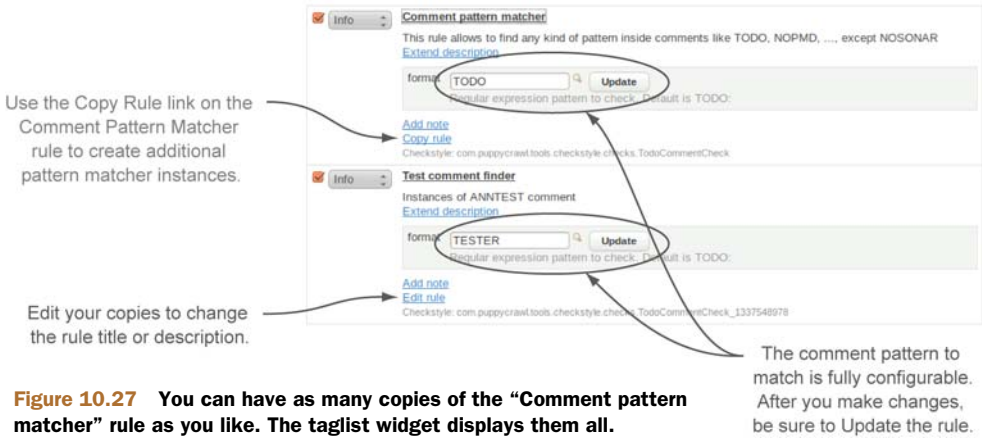


Figure 10.27 **You can have as many copies of the "Comment pattern matcher" rule as you like. The taglist widget displays them all.**

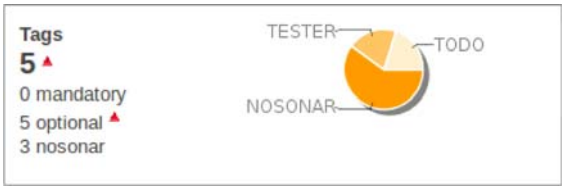


Figure 10.28 The pie in the taglist widget has a section for each relevant rule you've activated: one for the NOSONAR rule, and one for each instance of the Comment pattern matcher rule.

Keep in mind that by setting up rules, you use the issue mechanism to track the comments of interest. The examples in figure 10.27 are set up to flag comments at the lowest possible issue level, Info, but you could flag them at any severity.

With your rules set up, all that's left is to run a new analysis. When it's done, you'll see something like figure 10.28 in the dashboard.

The last thing to know is that the taglist widget makes a distinction between mandatory and optional tags; the difference is the severity the rule was set to. In figure 10.32, all the `TESTER` and `TODO` comments appear as optional tags because we set them to create Info-level issues. But issues of a rule that's set to Blocker or Critical would be registered as mandatory tags.

10.6.3 Widget Lab

The Widget Lab plugin offers a small suite of additional widgets, including a widget to show instances where an issue's severity has been manually adjusted. SonarQube keeps track of manual severity adjustments internally, but it doesn't offer any built-in way to find those issues or to see what the severity was originally. Widget Lab's manual severity widget gives dashboard-level visibility into downgraded or upgraded issues, showing both an adjusted issue's original severity and its current severity.

10.7 Summary

Regular code reviews can enhance your team's effectiveness and cohesion. Using SonarQube's workflow and action-plan mechanisms in a code review can make code reviews an even more effective tool for managing and paying down technical debt. SonarQube's workflows allow you to attach a running comment log to an issue. They also provide a mechanism to confirm an issue, and assign it to a SonarQube user. There are three ways to moderate an existing issue: flag it as a false positive, change the severity, or add a `//NOSONAR` comment to the end of the flagged line.

The manual issue mechanism lets you flag code problems that SonarQube doesn't (such as the need to restructure a method). Once a manual issue has been created, it can be managed and tracked in the same way as any other issue. SonarQube's action plans allow you to group issues and monitor progress. And SonarQube provides a project-level Issues dashboard, for quick oversight of issues and action plans. The global issues search allows you to search issues across projects.

Now that we've given code reviews thorough coverage, we'll turn next to what happens after a code review. In chapter 11, we'll examine what happens when you sit back down to work on what you were assigned in code review: that is, we'll look at SonarQube's IDE integration.

11 IDE integration

This chapter covers

- What's supported
- Setting up Eclipse
- Working your assigned issues
- Running a local analysis

Rules that show you what's wrong and rule descriptions that point you in the right direction for fixing the problems can help green developers become good developers, and good developers become even better. SonarQube gives you both.

But with SonarQube in place, your code quality is visible. Mistakes are no longer private or anonymous—the SCM Activity plugin pastes your ID next to each line of code like a billboard. Now there's team and even management focus on mistakes.

Of course, visibility into code quality is supposed to be a good thing. Co-ownership of the code and team spirit (agile practices) should focus on getting things done, not on placing blame. After all, issues aren't for pointing the finger but for increasing software quality. In the best situations, that's exactly what happens: each issue is a teaching opportunity. But in other situations, the attention SonarQube brings to code quality isn't always positive.

It's been likened to cruising down a highway and suddenly finding yourself in a lazy little speed trap of a town. Sure, SonarQube posts the speed limit; but if you don't know how fast you're going until after the cop tells you what speed he clocked you at—after the next analysis runs—well, it can seem a little unfair.

That's especially true because most developers—most good developers—are passionate about good code. Tell us ahead of time what the problems are, and we'll get them fixed. Embarrass us with them later, and we'll get … resentful.

Fortunately, IDE integration offers a solution by allowing you to check your code the same way SonarQube will before you check it in. But in the world of SonarQube IDE integration, there are definitely first- and second-class passengers. If you work in Eclipse, then welcome to first class. Would you like a glass of champagne before we take off? If you spend your days in some other IDE, you still get to board the plane (we'll look at what's available to you), but your flight won't be as comfortable.

In this chapter, we'll look at both classes and show you what's available to each. First-class passengers will see how to set up SonarQube's plugin for Eclipse and start making the most of it with issue management and local issue scanning. For those not traveling in first class today, we'll discuss how to pull the rules in your profile out of SonarQube for import into your IDE of choice.

If you're not an Eclipse user, and you're jealous because you wish you had access to the same level of integration, consider this: Eclipse is free and open source and supports development in a number of the same languages that SonarQube can analyze, including (but not limited to) C/C++, C#, Cobol, Flex, JavaScript, JSP, PHP, PL/SQL, Python, Natural, XML, and of course Java.

If you're not yet convinced to try switching environments, you might also like to know that the trend has been for the SonarSource folks to migrate language plugins away from third-party rules engines to native, SonarQube-only ones. So, the ability to pull your SonarQube rules into other IDEs and run them without SonarQube may eventually disappear.

11.1 What's supported

As the weekly code review ends, Susan looks down the table. "Okay Joe, how'd you do it?" He nearly suppresses a tiny smirk. "Do what?"

Susan laughs. "Come on, don't B.S. me. I got spanked this week for adding three issues in last night's build and forgetting to clean up one of the old issues assigned to me. Usually you're the goat, but you didn't add a single issue this week *and* you handled all the ones you were assigned. You must have a trick."

Joe considers. "I shouldn't tell you for a few more weeks. I like being the golden child for once." At Susan's look, he hurries on: "But, because I owe you … I installed that Eclipse plugin."

She smirks. "I didn't know there was a plugin for the Easy button."

"Har. No, seriously, the *SonarQube* a plugin. It lets me prescan my code before I check it in so I can make sure I'm not adding new issues."

Susan's eyebrows rise. "That sure would have helped me yesterday."

"Yeah, and I can see all the issues assigned to me right there in Eclipse. For each one, I get everything I see in SonarQube, plus I can double-click an issue to jump straight to the code. And once I'm done, I can set the issue to Resolved without having to open a browser."

As Susan and Joe head back to their desks, her voice floats back. "Huh. Looks like there might be more than one golden child next week."

Susan and Joe can take full advantage of SonarQube from within their IDE because their team uses Eclipse; but as we've said, there are two flavors of IDE integration: Eclipse and everything else. We'll spend most of this chapter on Eclipse, but first let's look at what everyone else gets. (Eclipse users may want to skip ahead to section 11.1.2, "Eclipse support.")

11.1.1 Generic support

Assuming your IDE has integration with the source code analysis engine(s) that SonarQube uses for your language of choice, then the integration you get is the ability to check your code against the same rules SonarQube will use before you check it in. To do that, you have to pull those rules into your IDE manually. On the SonarQube side, that starts from the description widget on your project's dashboard, which shows the profile it's analyzed against, as shown in figure 11.1.

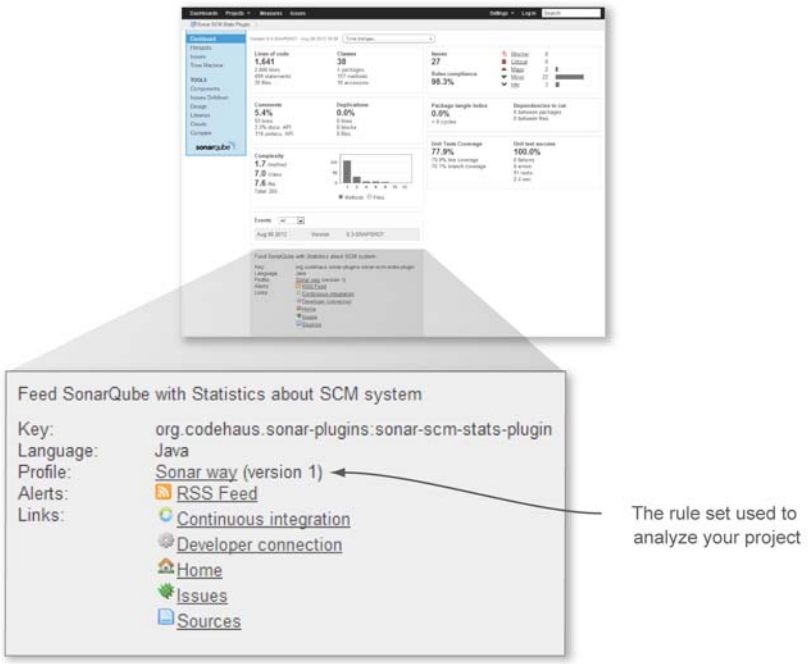


Figure 11.1 The description widget provides some of the static vitals of a project, including which profile it's analyzed with.

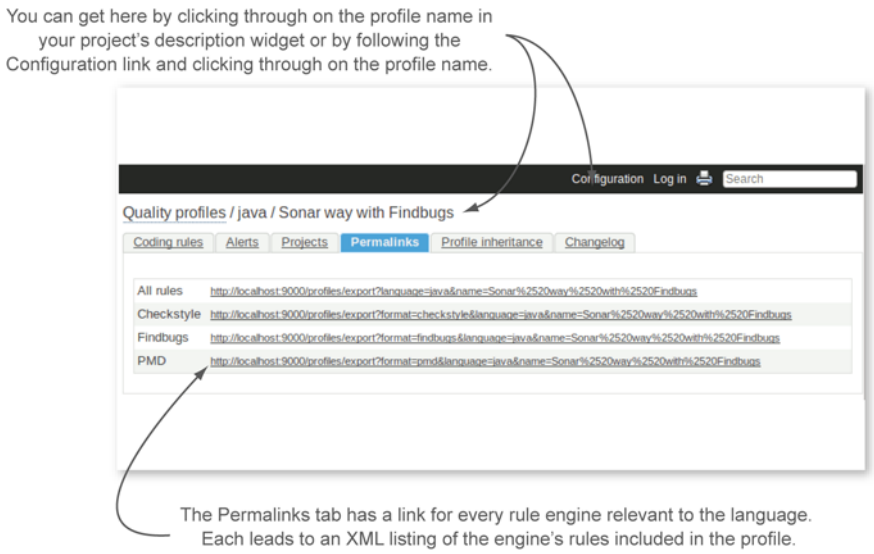


Figure 11.2 Each profile makes its lists of rules available as XML documents: one for each rules engine and one for the profile as a whole.

Click-through on the profile name, and you'll find yourself at the profile's list of rules. Select the Permalinks tab, as shown in figure 11.2.

On this tab is a link for every rule engine relevant to the language—whether or not it's used in the profile. Each one leads to a list in XML format of the rules from that engine that are in the profile. Once you have your lists of rules, the mechanics of importing them into your IDE obviously vary from editor to editor, but it's possible to do in many popular IDEs (including Visual Studio, NetBeans, and IntelliJ IDEA). Typically this is done by adding a plugin for the rule engine to your IDE and then using the plugin's management options to import the XML rule set.

After you import your rules, whether scans are triggered automatically or manually depends on what's offered by your IDE. But at least you can double-check yourself before committing your code changes. Just keep in mind that you won't be running any SonarQube-native rules, and you'll need to reimport if your rule set changes. If those restrictions are too confining, then you may want to take advantage of the Issues Report plugin, which we'll detail at the end of the chapter. It doesn't integrate with any IDEs, but it does give you the ability to run local scans against your *full rule set*—without the need to keep a local copy of the rules in sync with the server.

11.1.2 Eclipse support

If you're working in Eclipse, SonarQube integration is a lot easier and more richly featured for you than for everyone else. You have to trigger scans manually, but you don't have to manually import the rule sets. Instead, they're pulled from SonarQube as part of the scan. There's no need to notice when your profile has changed and reimport

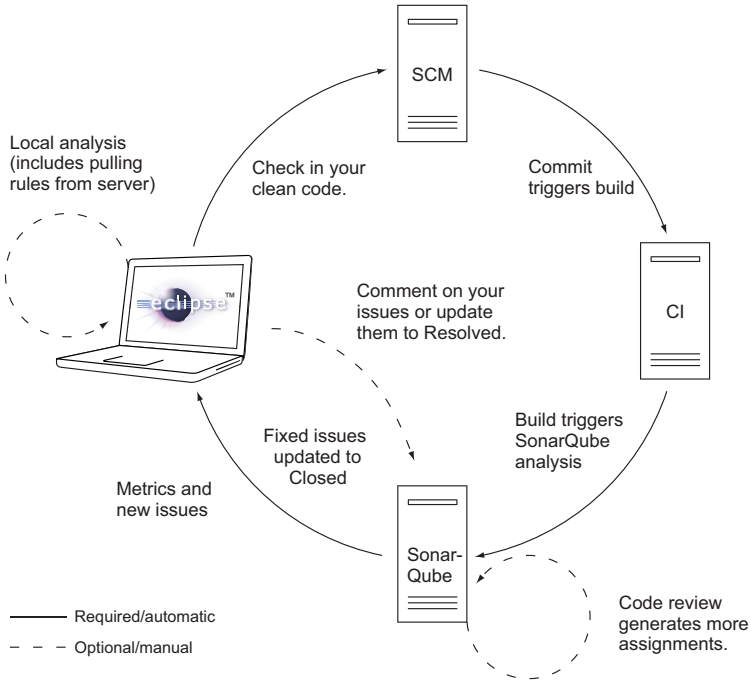


Figure 11.3 The SonarQube plugin for Eclipse helps you close the circle of code quality. It gives you local visibility into existing issues in the files you're working on, helps you manage your issue assignments, and lets you prescan your code changes for issues.

the rules, because that happens automatically. SonarQube integration helps you bring your code quality efforts full circle.

It's not just a one-way trip from your IDE through source-code management and continuous integration into SonarQube. Now you can pull what SonarQube has to say directly into your IDE, completing the cycle, as shown in figure 11.3.

Once you've closed the circle, in addition to being able to pull up a list of issues for a project, package, or file, you also get a SonarQube issue tree in the markers view and a warning in the problems view for each issue, as well as a marginal marker for each issue. If you're working in Java, you'll notice that some of the native Eclipse warnings duplicate SonarQube rules. For an unused method, for instance, you get both a SonarQube issue marker and an Eclipse warning. If the duplication bothers you, you can turn off the Eclipse warnings at the project level (right-click the project in the Package or Project Explorer, and choose Properties) or globally (Window > Preferences). You'll find the warning settings under Java > Compiler > Errors/Warnings.

In addition to displaying the issues on your code as you work, SonarQube integration also closes the circle with issue-management functionality. You can view your assigned issues, comment on them, and mark them resolved.

11.2 Setting up Eclipse integration

The SonarQube integration for Eclipse is pretty nice, but before you can use it, you'll need to set it up. There are four steps:

1. Install the SonarQube plugin for Eclipse.
2. Configure your SonarQube server.
3. Link your workspace project to the copy on SonarQube.
4. Add the SonarQube views to your current perspective.

None of the steps is hard, and, assuming you have a decent internet connection, they shouldn't take long.

11.2.1 Installing the plugin

Eclipse makes plugin installation in general fairly painless. The most important thing to know for any plugin is the URL of its update site. In this case, it's http://dist.sonar-ide.codehaus.org/eclipse.

With the URL for the update site in hand (or on the clipboard), go to the Help menu and choose Install New Software. You'll get a dialog like the one shown in figure 11.4.

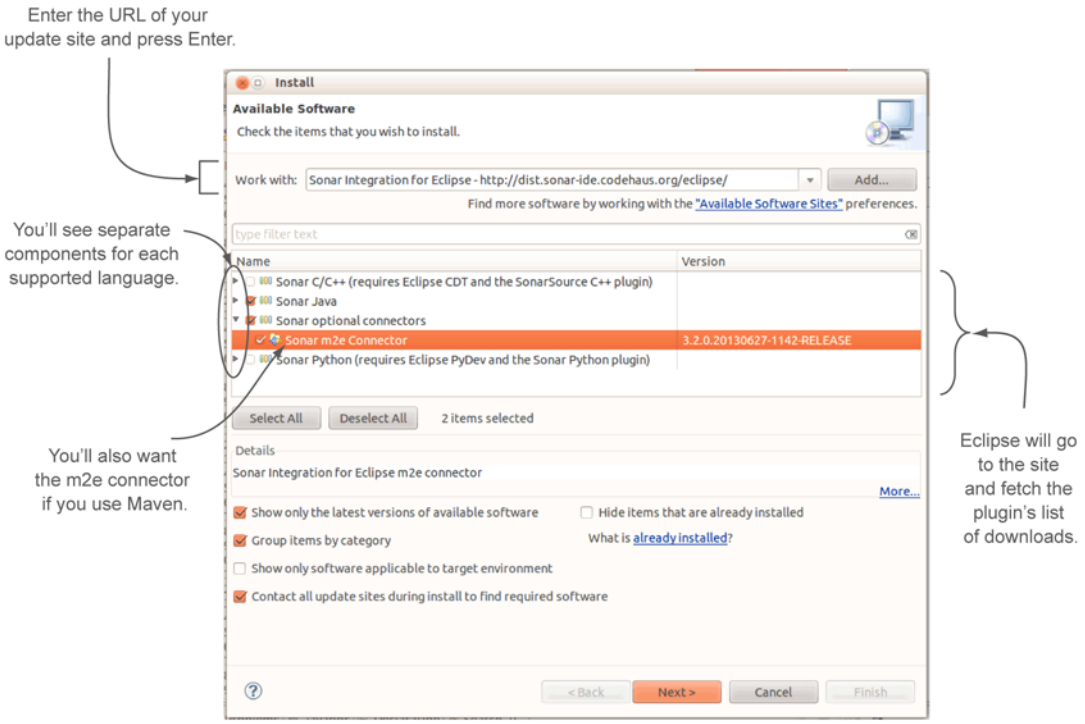


Figure 11.4 Eclipse plugin installation starts with the URL of a download site. From there, Eclipse takes over, checking dependencies, retrieving all the pieces, and getting your agreement to the appropriate licenses.

Enter the URL in the first field, and press Enter. Eclipse checks the site and recursively fetches the list of downloads for the plugin. Alternately, choose Help > Eclipse Marketplace. Search for *Sonar*, and, once the plugin is found in the Marketplace, click its Install button.

> **NOTE** We mentioned earlier in this book that *SonarQube* used to be called *Sonar*. Similarly, SonarQube's *issues* used to be called *violations*. This chapter was last revised when that transition was incomplete. Those name changes were made at the server but not yet in Eclipse. That's why you'll see references in this chapter to *Sonar* and *violations*. Please treat them as synonyms for *SonarQube* and *issues*.

Installation via either avenue is remarkably similar from this point. In the list of plugin components, select the components you want, including those needed for the languages you'd like to analyze in Eclipse plus any optional components you're interested in. Once you've picked all your components, click Next. On the next screen, you have a chance to review a granular list of downloads. Click Next again, and you're asked to agree to the components' licenses. After you do, the downloads begin.

Note that it's common to get security warnings about unsigned content while installing Eclipse plugins. These warnings make you explicitly accept the risk in order to continue with the installation. We typically say OK and finish the install, but you have to make your own security decisions.

After a successful installation, you're urged to restart Eclipse. Go ahead and do that.

11.2.2 Configuring the server

When Eclipse comes back up, you need to tell it where your SonarQube server is. Choose Window > Preferences, and then expand the Sonar option in the Preferences dialog and choose the Servers suboption. A Sonar server at localhost is preconfigured. Click Add to set up a server on another host, or edit an existing entry. You should configure not only the server location, but also your SonarQube credentials. Figure 11.5 shows the dialog.

11.2.3 Project association

Now that Eclipse is configured, it's time to configure the project. From the Package Explorer—or from the Project Explorer—right-click the project, and choose Configure. In the submenu, pick Associate with Sonar, as shown in figure 11.6.

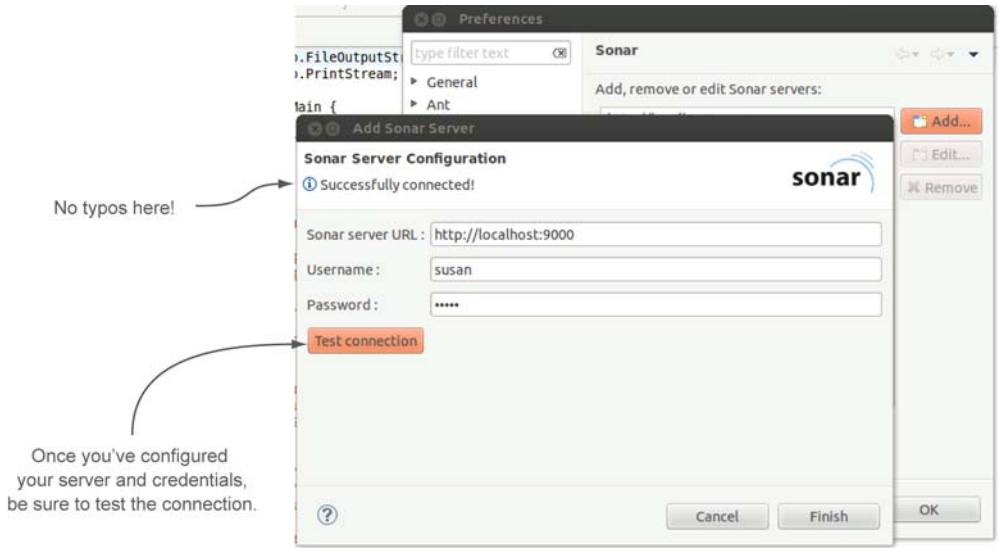


Figure 11.5 At minimum, you need to tell Eclipse where your SonarQube server is. If you want to be able to manage your assigned issues from within Eclipse, you also need to set up the credentials you use to log in to SonarQube. If your SonarQube authentication is integrated with LDAP, don't forget that you'll need to come back here after a password change!

If you're using Maven, or if your project's name coincides with its key in SonarQube, it's nearly effortless from here—the plugin automatically searches your servers for projects with a matching name and fills in the most likely match. But if you're not using Maven and your project names in Eclipse and SonarQube don't match, you have to do a little more typing: click into the Sonar Project field, and enter your project name. When you click Finish in this dialog, Eclipse immediately begins loading your project's data from SonarQube. Once that load is complete, you may notice additional markers in the Eclipse interface, as shown in figure 11.7.

Now you'll want to make a few changes to your current perspective to make the most of the Eclipse plugin. Choose Window > Show View > Other. In the Show View dialog, several options appear on the Sonar menu, but we'll only work with Sonar Issues and Sonar Issue Editor in this chapter.

11.3 Working your assigned issues

At the beginning of this chapter, one of the things Susan got scolded for during code review was leaving one of her assigned SonarQube issues unresolved. Fortunately, the Eclipse integration makes issues easy to find, so you'll never have to look sheepish for forgetting again (unless that's the best excuse that comes to mind).

At this point, we'll assume you've followed along and you have the Eclipse plugin fully configured. You should see issues listed in the Sonar Issues view, and you're about to sort them to find the ones assigned to you. If you don't have any assigned issues, well … go assign yourself some low-hanging fruit like missing curly-brace issues or unused imports. Don't worry about over-committing yourself; this will be easy.

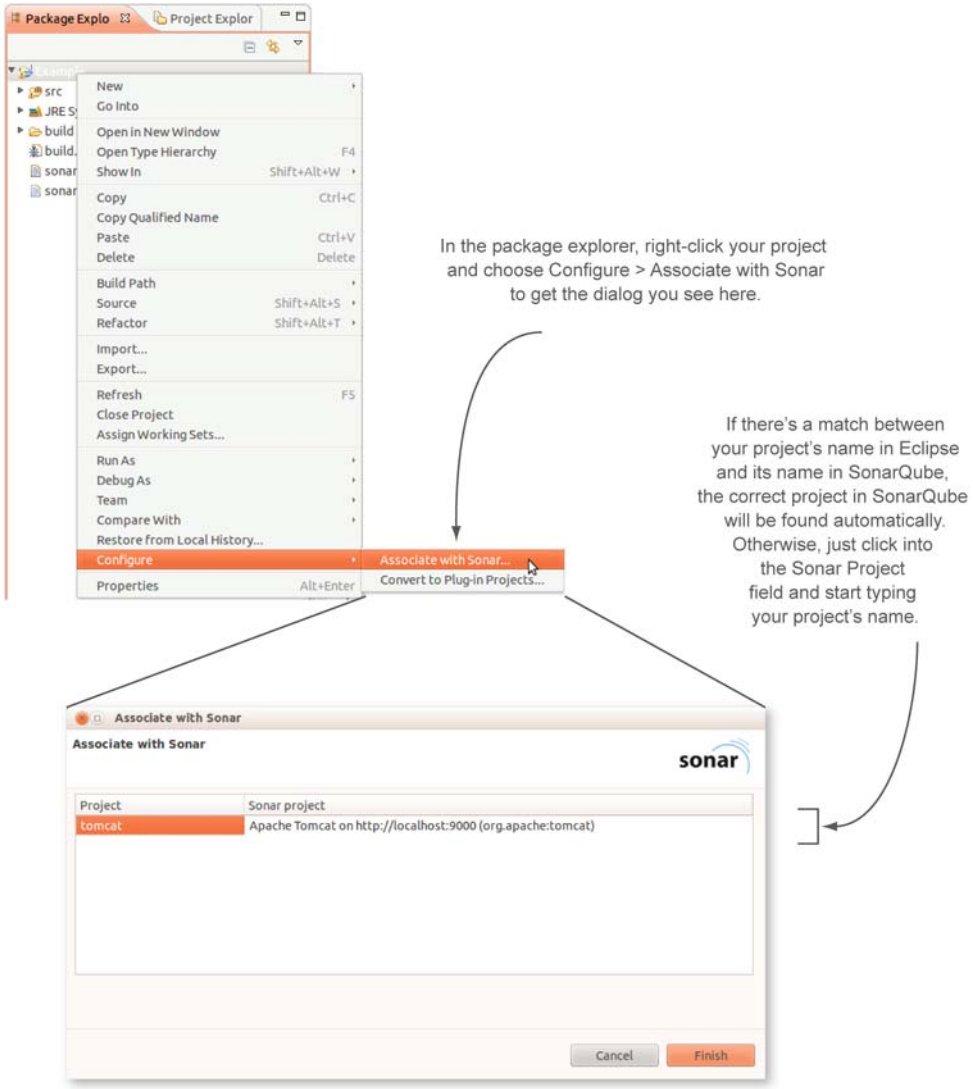


Figure 11.6 To associate your project with SonarQube, type the project name in the Sonar Project field, if it's not prefilled for you.

You have two choices for how to do the assignment. You can head for the SonarQube interface in a browser, or you can click an issue in the Sonar Issues view and then switch to the Sonar Issue Edit view. This view does what you might guess—it lets you edit an issue—with the same functionality you get in the SonarQube interface, because what it shows is a tiny slice pulled directly from the SonarQube interface: the issue summary you saw in chapter 10, complete with all the workflow options including assignment. If you included your SonarQube credentials when you set up the server earlier, it even includes the Assign to Me option.

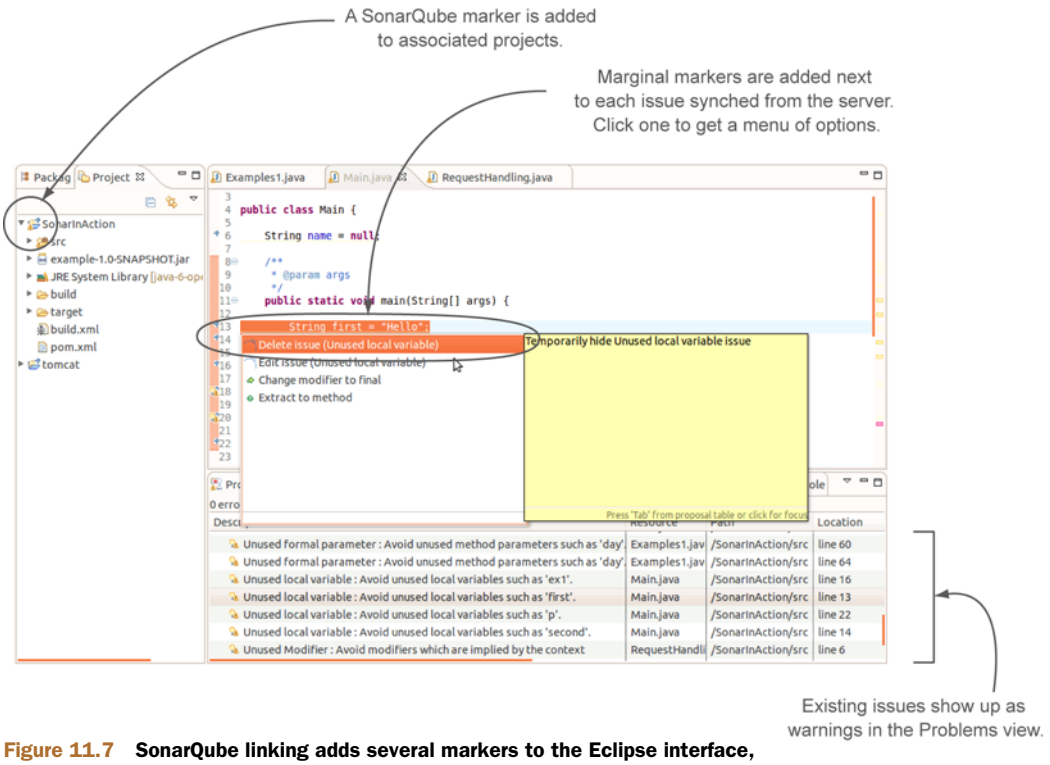


Figure 11.7 SonarQube linking adds several markers to the Eclipse interface, including one on the project and a marginal marker for each existing issue synched in from the server. Existing issues also show up as warnings in the Problems view.

11.3.1 Finding your assigned issues

If you did your issue-assigning from Eclipse, then your Sonar Issues tab updated automatically to show the new assignments. But if you used the SonarQube interface to make the assignments, you may be wondering how to pull them in—they didn't magically appear when you flipped back into Eclipse. Use the Synchronize button in the Sonar Issues tab, as shown in figure 11.8.

At this point, you probably see a lot of issues in the list. If you're lucky, the ones assigned to you bobbed to the top. But they probably didn't. That's easy to fix, though, and you have a couple of choices for how to go about it. Right next to the Synchronize button is the selector for the view menu, which lets you sort issues by assignee. By default, issues are grouped by severity; but if you set the grouping to None (also in the view menu), all your assigned issues will sort together. Of course, if your name starts with a letter from the middle of the alphabet (you can flip the sort from ascending to descending by clicking the column header), this isn't the best solution.

Fortunately, the Configure Contents option on the view menu lets you configure which issues you see. You can even specify multiple issue filters. There are two default filters; you can edit them or add your own, as figure 11.9 shows. Then you can choose

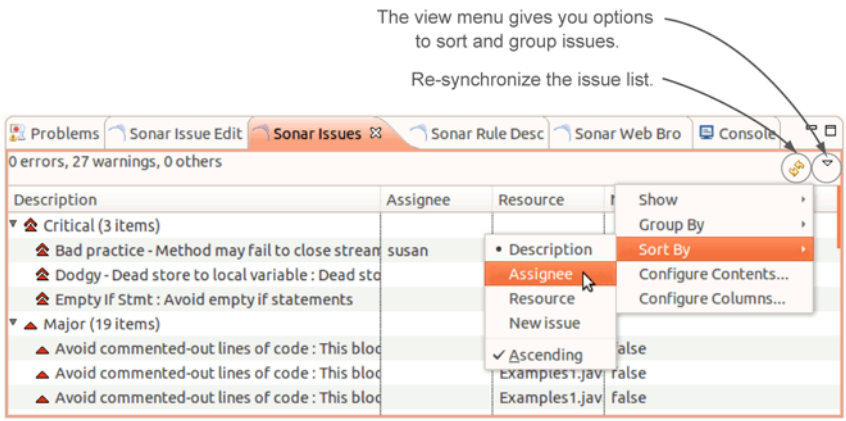


Figure 11.8 **Use the Synchronize button to re-synchronize SonarQube issues.**

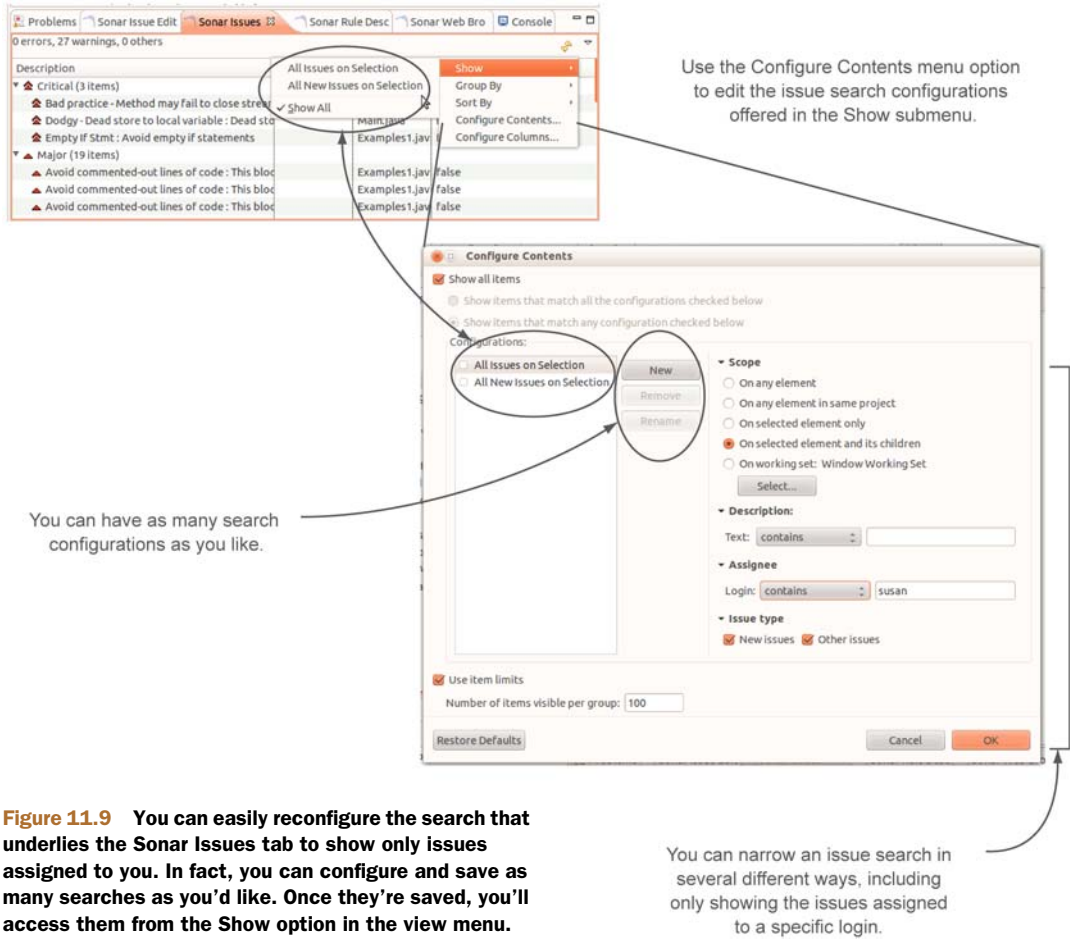


Figure 11.9 **You can easily reconfigure the search that underlies the Sonar Issues tab to show only issues assigned to you. In fact, you can configure and save as many searches as you'd like. Once they're saved, you'll access them from the Show option in the view menu.**

to layer them by turning on several at once (use the check box next to the filter name in the Configure Contents interface), or you can turn them all off to see all the issues.

11.3.2 Finding and fixing the code

Now that you have issues to work on, double-click one. Eclipse opens the associated file and puts your cursor on the right line. All you have to do is add the missing curly brace or whatever was called for by the issue you grabbed.

Once that's done, you may think the issue will automatically be marked Closed, or at least Resolved. The integration is good, but it's not that good—you have to do that part manually. Head back to the Sonar Issue Edit view, make whatever comments you like, and mark the issue Resolved (under the More Actions menu).

We've covered resolving an issue here for the sake of keeping all the issue-related steps together, but normally we'd counsel running a local SonarQube scan first (doing so is covered in the next section). After all, you'd hate to resolve an issue only to have SonarQube reopen it.

11.4 Running a local analysis

By all means, let's run a local analysis. But first, you may want to find your target issue in the list of issues so you can verify that it disappears after the analysis. Doing this won't affect the analysis, but we like to see issues disappear from the list, just to make sure.

Now, in the Package Explorer (or the Project Explorer), right-click the project and choose Sonar > Mode > Local. The Console view immediately pops to the front, and an analysis begins. You see it interact with the SonarQube server briefly at the beginning to register metrics and rule repositories and then to select the appropriate rule profile for your project. From that point on, though, it's a completely local operation. None of these results are stored on your SonarQube server—this analysis is for your benefit only.

The console output during the analysis is similar to what you're used to, but this analysis doesn't take as long as most—it's primarily an issue scan. Other parts of the analysis, such as duplications detection, are left out.

When the analysis is done, you should be able to use the Sonar Issues tab to verify that you have indeed eliminated the issue you were targeting. Notice that *all* the markers for the issue are gone—not just the item in the list on the Issues tab, but also the marginal markers (and their accompanying mouseovers) that are added for each issue.

You've stepped through the whole workflow now, although admittedly not in the optimal order, which is as follows:

1. Pull in assigned issues.
2. Fix issues.
3. Verify your fix with a local analysis.
4. Mark issues resolved.

One thing you haven't done is close a fixed issue. Remember, that only happens when the root issue disappears during a full SonarQube analysis (the analyses you run locally in Eclipse don't count).

As a best practice before any SCM commit, we typically open each file we're about to check in and run a local analysis. Then it's easy to flip through each file, making sure we've fixed what we meant to and haven't added any new issues. When we've run through that cycle enough times to get everything right, it's time to commit our changes. At that point, the Continuous Integration/Inspection system kicks in and fires off a full analysis. It may take a little while for the changes to churn through the system, but eventually SonarQube closes those issues. You've seen the 30,000-foot view shown in figure 11.10 before, but it bears repeating now that you've had the opportunity to walk through every step in the process.

The next time you want to run a local analysis, you'll use a different menu item. By default, you start in remote mode, pulling values from the Sonar server. Right-clicking in your project and choosing Sonar > Mode > Local does two things: it kicks off a local analysis, and it flips you into local mode. So next time through, you'll choose Sonar > Run Local Analysis. If you want to re-pull what SonarQube has on file, then you'll need to use the remote option: Sonar > Mode > Remote.

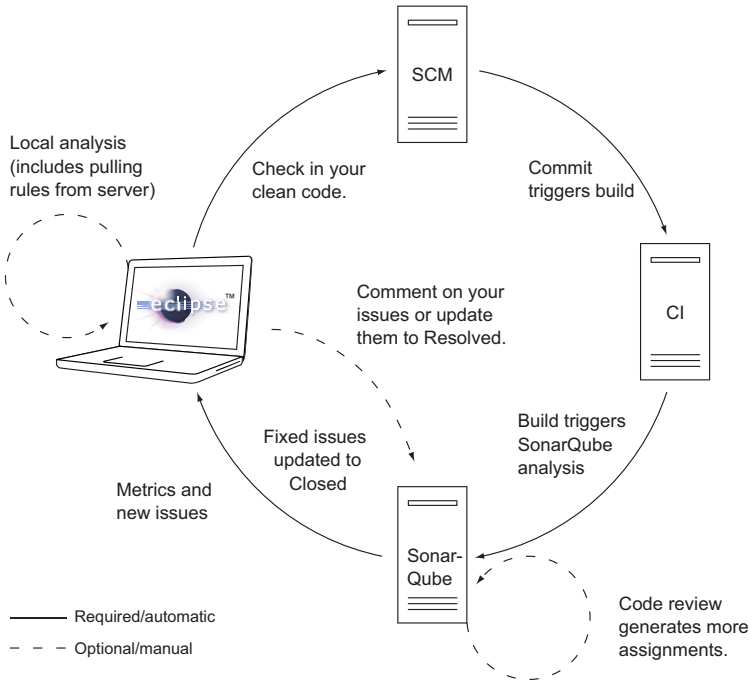


Figure 11.10 If everything's working as it should, you see a virtuous cycle of improved code quality and higher developer productivity from the combination of Continuous Inspection, code reviews, and Eclipse integration.

In fact, let's do that now. But first, add an issue. It could be anything—just something you know breaks a SonarQube rule. Then run a local analysis. When it's finished, you'll see that your new issue is picked up in the analysis, as expected. What you perhaps didn't expect is that it gets special treatment. The normal SonarQube marginal marker is a tiny blue SonarQube logo. But for new issues, the logo is red. Further, new issues are marked not as warnings in the Problems view, but as errors, so Eclipse's error-marking mechanisms kick in: a red X on the project, on each file that introduces new issues, and on the packages that contain them. Run an analysis before each commit, and you'll never inadvertently add new issues again!

On the other hand, if having new issues presented the same way as compile breaks bothers you (it does bother some people), you can easily turn it off in the global preferences. You'll find this option under Window > Preferences. Click Sonar, and you'll see the option to adjust the severity of the markers.

11.5 Related plugins

If you don't use Eclipse, and the makeshift integration available for other IDEs doesn't suit you, the Issues Report plugin offers another alternative.

11.5.1 Issues Report

Although it offers a local analysis option, the Issues Report plugin is installed server-side. Once you've installed it and restarted SonarQube, you're ready to run an analysis that won't update the SonarQube database.

Similar to the functionality in Eclipse, you manually trigger an analysis that checks with the server to pull the current set of rules for your project and runs a local-only analysis. The Eclipse integration presents the analysis results in an Issues view; Issues Report generates an HTML-formatted file.

To take advantage of the plugin, each developer needs an analysis engine (see appendix B) such as SonarQube Runner installed locally. In this case, though, the installation is a rather shallow one—there's no need to configure the SonarQube database location or credentials because you aren't updating the database. Instead, you only configure the URL of the SonarQube host (SonarQube Runner interacts with the SonarQube web services to pull the data it needs for analysis) and make sure `sonar-runner` is in your `PATH` variable. Then you need to set up the project properties file.

Listing 11.1 sonar-project.properties

```
sonar.projectKey=test:project2
sonar.projectName=Test
sonar.projectVersion=1.0
sonar.sources=src

sonar.binaries=bin

sonar.dryRun=true
```

You still need to provide all the required properties

You can choose to provide any of the optional properties

Required for local-only analysis

```
sonar.issuesReport.html.enable=true

sonar.issuesReport.html.location=issues-report.html

sonar.issuesReport.console.enable=true
```

Optional: sets output format

Optional: defaults to .sonar/issues-report.html

Optional: emits brief report summary to console

Once your properties file is in place, fire off an analysis. It won't take as long to run as a full analysis, because it's only an issues check. At the end of the console output, you'll see something like this:

```
20:28:49.515 INFO  - HTML Issues Report generated: /path/to/issues-report.html
20:28:49.515 INFO  - -------------  Issues Report  -------------
20:28:49.515 INFO  - 3 new violations
20:28:49.515 INFO  - +3 major violations
20:28:49.515 INFO  - ----------------------------------------
```

The resulting report starts with a project-level summary of new and existing issues, as shown in figure 11.11, and continues with a file-by-file breakdown.

11.6 Summary

IDE integration is a developer's first line of defense against the embarrassment of adding new issues to a project, and SonarQube enables that for a wide variety of development environments.

If you don't spend your days in Eclipse, your best bet is the Issues Report plugin, which offers a solid subset of the functionality available to Eclipse users. If you really want IDE integration, SonarQube offers the ability to import your rule set into your IDE for local scans, although as more and more of SonarQube's language integrations go native, that ability will slowly diminish. In the meantime, you'll need to remember to reimport your rules when the rule profile changes. But that's better than being caught out in code review.

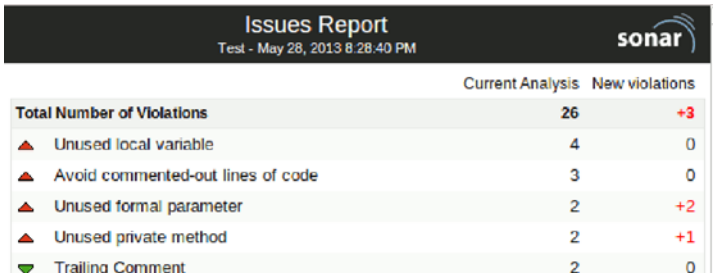


Figure 11.11 The output from the Issues Report analysis is a single HTML file that shows at a project-level and file-by-file level which rules were violated and how many of the issues are new since the last full analysis.

If you do work in Eclipse, then the offerings are far richer. With integration that's rapidly approaching seamless, the SonarQube plugin for Eclipse brings code-quality remediation full circle. We showed you how to install the plugin, point it to your SonarQube server, and connect your local project to the copy on SonarQube. We walked through setting up an issue query to retrieve your assigned issues, and we showed you how to use an issue to jump right to the problem it describes.

Once you make your fixes, the plugin lets you prescan your code to make sure you haven't inadvertently introduced any new issues before you check in your changes. Then you can mark the issue resolved without having to leave Eclipse.

In the next part of the book, we'll dive into administration—users and groups, rules and profiles, and projects and SonarQube itself. You'll learn to tune your rule sets to eliminate the rules that don't fit your situation, add the ones that do, and adjust the remaining rules so their priorities line up with yours. You'll see how to fine-tune your project settings and SonarQube itself. And you'll learn how to set up project rights so you can delegate project administration to a team's project or development lead, rather than handling it all yourself.

Part 3

Administering and extending

This part of *SonarQube in Action* covers administration. In the previous part of the book, we discussed how to use SonarQube's capabilities to get the most out of it on a day-to-day basis. Now we'll explain how to tune SonarQube to your environment, starting with permissions in chapter 12 and rule profile administration in chapter 13, and moving on to global and project-level settings in chapters 14 and 15. We'll end the book by showing you how to further customize SonarQube by writing your own plugins in chapter 16.

12 Security: users, groups, and roles

This chapter covers

- Creating users and groups
- Roles: who can do what
- System administrators

So far, you've been working under the default accounts and permissions. You've made all your changes as Admin, and all your users have presumably been accessing SonarQube anonymously. In this chapter, we'll help you set up a slightly more sophisticated system that takes advantage of SonarQube's role-based authorization, which determines who can do what; and we'll show you how to grant those roles at the project level to groups and individual users. Once your users have their own accounts in SonarQube, they'll be able to manage their own preferences, so we'll look at the settings on a user account and discuss the features available to a user.

We'll end the chapter with a look at the plugins that let you delegate authentication to an existing system through the Lightweight Directory Access Protocol (LDAP), Central Authentication Service (CAS) protocol, Security Assertion Markup Language (SAML), OpenID, or Protocol Analysis Module (PAM).

12.1 Creating users and groups

The foundations of security in any application are users and their permissions. In a good application (which SonarQube is), these foundations include not just users and their individually granted permissions, but users, the groups they belong to, and user or group permissions. SonarQube can handle both authentication and authorization for you, but there are also several authentication integrations available, which we'll tell you about at the end of the chapter. Using one of them will render user creation moot. But you *can* manage users directly in SonarQube; and whether or not you do, you still *have* to manage your groups in SonarQube. Both are easy, and this section will show you how.

12.1.1 Managing users

Russell sighed. Usually, being the company's SonarQube admin was a piece of cake—once he got the analyses set up, they pretty much ran themselves. But lately he'd been spending a lot of time making piddly little SonarQube changes for a bunch of projects. The Blue team wanted to tweak their analysis exclusions—over and over again—while the Red team wanted some bad snapshots deleted and the Green team needed their Manual Measures updated. No single change took much time, but it was like being attacked by gnats—a never-ending swarm of annoyances—and each one pulled him away from the things he was supposed to be doing.

Fed up, Russell was considering handing out the default SonarQube admin credentials he'd been using (in appendix A, you'll learn that the default admin credentials are login: *admin*, password: *admin*, and we'll urge you to change the password), when he remembered he'd never dug into SonarQube's security. As he fired up his browser, he wondered if there was a way to let one or two people on each team administer their own projects without having to give them the keys to the kingdom.

Because Russell wanted to grant rights to other users, he figured the Users option was a good place to start. The interface he found is shown in figure 12.1.

The interface for user management is straightforward. Just fill out and submit the Add New User form, and voilà! Your newly created user is added to the list in the center of the screen. Once created, you can't edit a user's login, but you can use the links provided to the right of each user to edit her name and email address, or use the Change Password link for a password reset. When you use the Edit links, the Add New User form is replaced with the Edit User form. At a glance, the forms are similar: blink, and you could miss the swap. So if you think you've been waiting a long time for the page to refresh and take you to the editing screen, make sure you read the title on the form at right on the page—it may already be in front of your face, as figure 12.2 shows.

As you'd expect, the Delete link by each user gives you a confirmation dialog. What you may not expect is that clicking the link doesn't actually delete the user. But that makes sense when you consider that the user you're axing may have entered some of the issues we talked about in chapter 10, and truly deleting the user would eliminate all record of his contributions and wreak havoc on the issue comment logs. Instead, the Delete link deactivates the user, removing him from all the security interfaces.

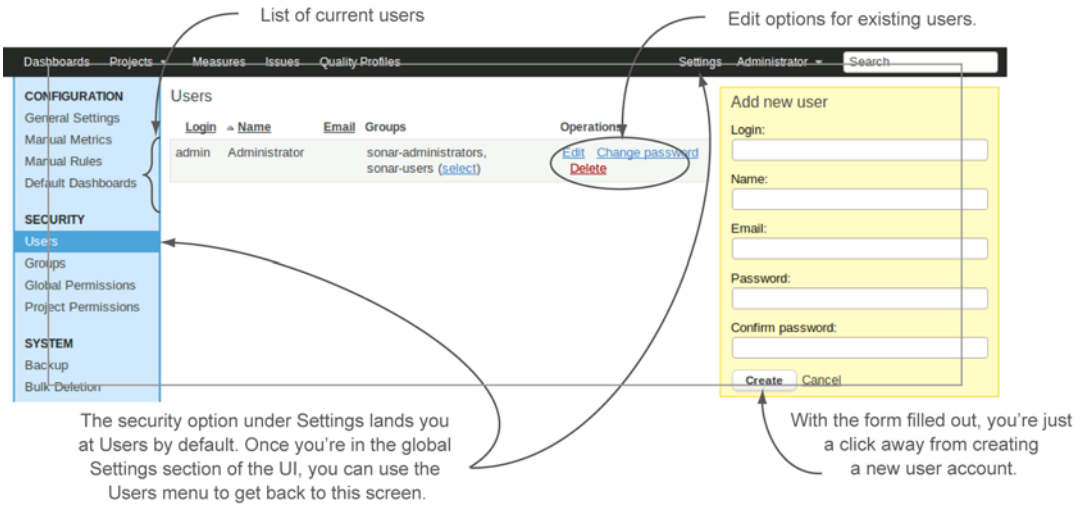


Figure 12.1 The Users interface gives a list of current users, links to manage each one, and the form to create new users.

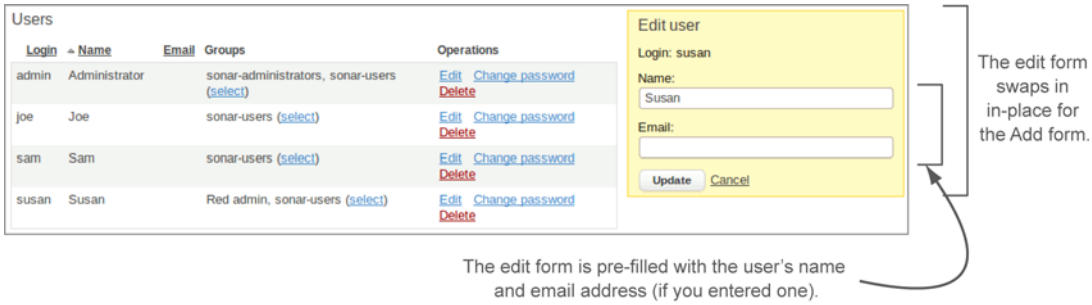


Figure 12.2 When you click the link to edit a user, the Edit User form replaces the Add New User form. This is typically a blink-and-you'll-miss-it operation.

Of course, *deactivate* implies *reactivate*, but there's no button for that. Instead, you re-add a user with the same login. If you do, you'll see a dialog like the one shown in figure 12.3. When you resurrect a previously deleted user through the Add interface, you revive his old account, but the name and email address are the values you entered on the Add form—as though you had done an edit on the original account.

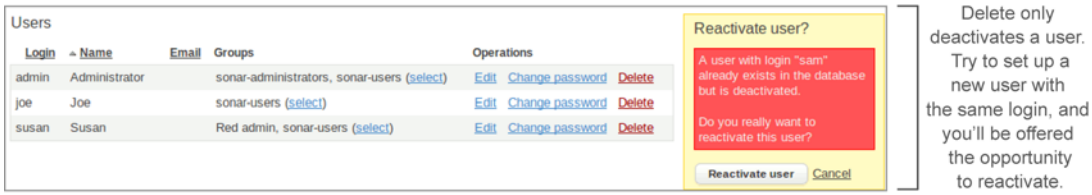


Figure 12.3 Once you've deleted a user, you can get him back by re-adding a user with the same login. If resurrection wasn't intended, the additional confirmation shown here gives you the chance to go back and try another user name.

The other things you'll notice inline with each user are a list of that user's groups and a link to a separate interface where you can add the user to multiple groups in one easy go. But first, you need groups. SonarQube starts you out with a couple of default groups. But Russell needs an admin group per team, so in the next section we'll look at setting that up.

12.1.2 Personalization: what users can manage for themselves

First, let's see what having a user account gets you. The administrator isn't the only person who can change a user's password. When users log in to SonarQube, they have access to a brief menu of options tucked under their names in the top menu bar. What they see after picking the My Profile option is shown in figure 12.4.

We'll show you how to configure the global email settings in chapter 14; once you have that set up, users can also have SonarQube send them emails. They can choose to receive notification when "their" issues are edited, either throughout the system or on specific projects. They can also choose email notification for new issues, alerts, and false positives either globally or on specific projects.

The final thing users can do for themselves is set up their own filters—so they can pick and choose the projects they see in a list based on a number of different criteria—and dashboards. (We'll cover filter administration in chapter 14 and dashboard administration in chapters 14 and 15.) Once you're comfortable with filter and dashboard administration, it will be in your best interest as a harried administrator to

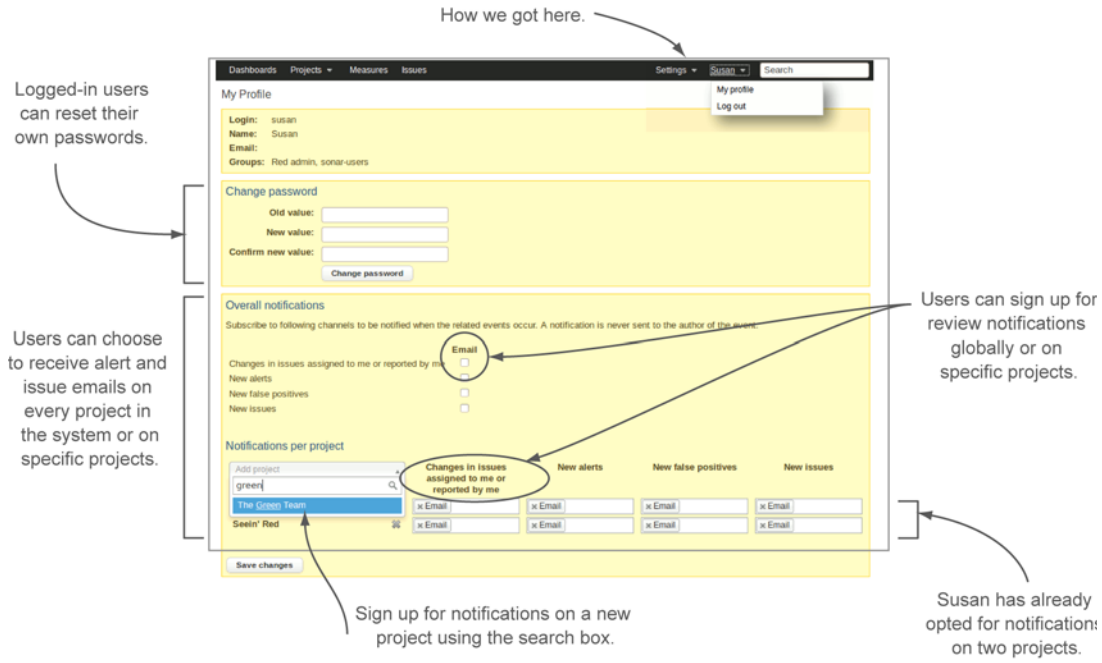


Figure 12.4 Users can perform their own password changes and set their email preferences, as well as manage their own filters.

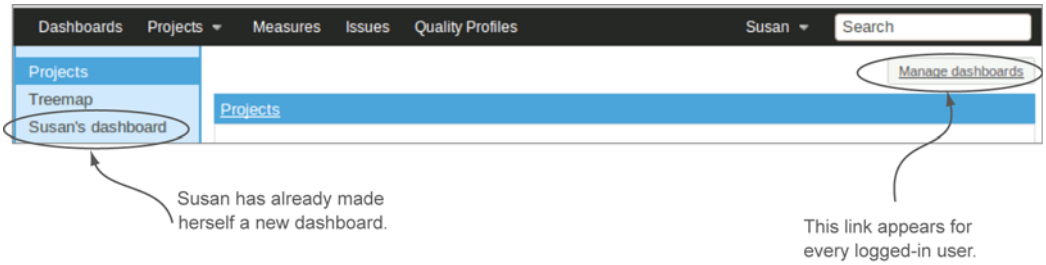


Figure 12.5 Users can make their own private dashboards with whatever filters and widgets they like. New dashboards appear in the left navigation rail just like any other dashboard. Users can control the order in which dashboards appear.

teach your users how to take advantage of these capabilities (because every filter and dashboard they make for themselves is one you don't have to make for them). You administer filters in the Configuration area of the interface. Dashboard administration is available from the Manage Dashboards link that appears for a logged-in user at upper right on each dashboard, as shown in figure 12.5.

There's one more thing to mention before we leave users. The Admin account is just another user. In addition to all the administrative functions available to someone logged in with those credentials, she also sees at the top of the navigation list the same basic user options that any other user sees. And she has the same interface for changing the Admin account's password and setting its email preferences.

Now let's move on to managing groups.

12.1.3 Managing groups

SonarQube gives you the ability to grant permissions to either users or groups. But once you get past a couple dozen people, managing permissions for individual users can be a real pain—which is where groups come in. If you bundle your users into groups and only grant permissions to groups, controlling who can do what becomes much, much easier.

Out of the box, SonarQube provides two groups: sonar-administrators and sonar-users. We'll look at global administration permissions a little later in the chapter, but the global Admin account you've been using so far is a member of the sonar-administrators group; this is what gives the account its global administration privileges, as well as admin privileges on every project. All new users are automatically added to the sonar-users group, which by default has non-admin permissions to every project (we'll examine those privileges in the next section).

Russell, the harried SonarQube admin, needs to create some new groups. As figure 12.6 shows, the group-management interface is even simpler than the one for users because there are fewer moving parts: only Name and Description.

As with users, the Add and Edit forms swap in and out of the same spot at right on the group-management page. The only differences between them are the title at the top and whether the inputs are prefilled.

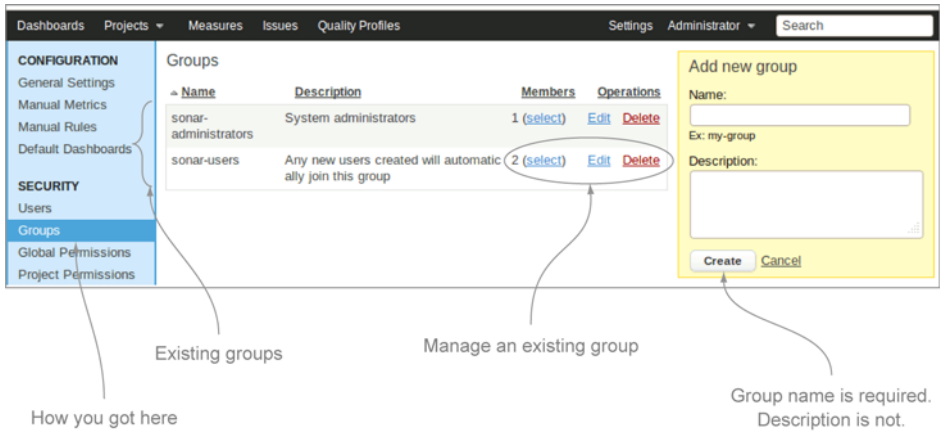


Figure 12.6 With fewer options, the group-management interface is even more straightforward than the one for users.

With some groups in place, you're ready to start adding users to them. Click the Select link in the Members column for the target group: doing so takes you to a page like the one shown in figure 12.7.

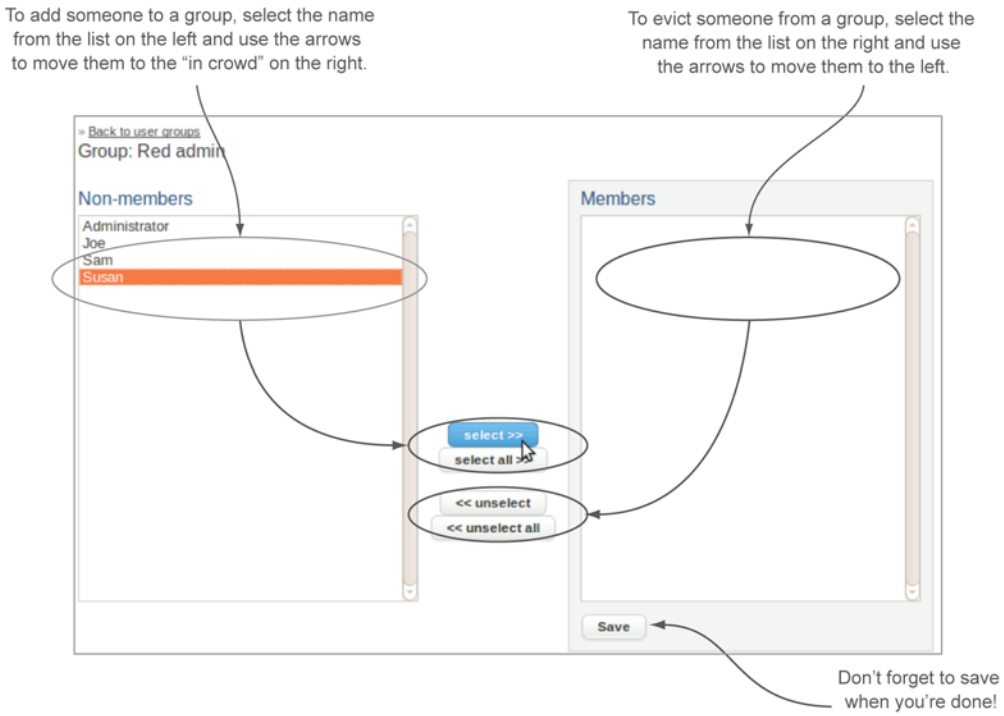


Figure 12.7 The group membership interface presents two lists: the in crowd on the right and everyone else on the left. Use the arrows between the two lists to move people from one side to the other.

NOTE When we looked at users, we skipped the interface to add groups to a user because you hadn't created any groups yet. The interface to do so is much like that used to add users to a group, with a list of in groups on the right, the groups the user isn't in on the left, and arrows in between to move items from one side to the other.

12.2 Roles: who can do what

Now that you have users and groups, it's time to start granting permissions. That's done through project *roles*. Membership in a role grants specific sets of privileges. You can grant a project role to a group or to an individual user; but managing individually granted permissions can be a pain, so you probably want to stick with groups. You can grant three roles: Administrator, User, and Code Viewer. We'll cover their permissions in that order. But first, let's look at the interface—there's a bit more going on in it than in the ones you've seen so far in this chapter, as figure 12.8 shows.

The bottom section of the Roles interface is a list of projects. To the right of each project name is a list of the users and groups who hold each of the three project roles. Use any of the Select links, and you'll find yourself looking at another iteration of the two-lists-with-arrows-between interface to add either groups or users to a project role.

If you're following along in your own SonarQube installation, you may be wondering at this point why you're seeing entries in your projects' role lists if you've never been to this page before. That's what the top section of the Roles interface is about. It sets up the default permissions to be granted on any new project. The top row, with the Administrator role, shows that the sonar-administrators group is granted project administration rights on all new projects. The next two rows show that Anyone and the sonar-users group get the other two roles on every new project. You can use the links next to each role to change those defaults. Click through on any of the links to open the interface shown in figure 12.9.

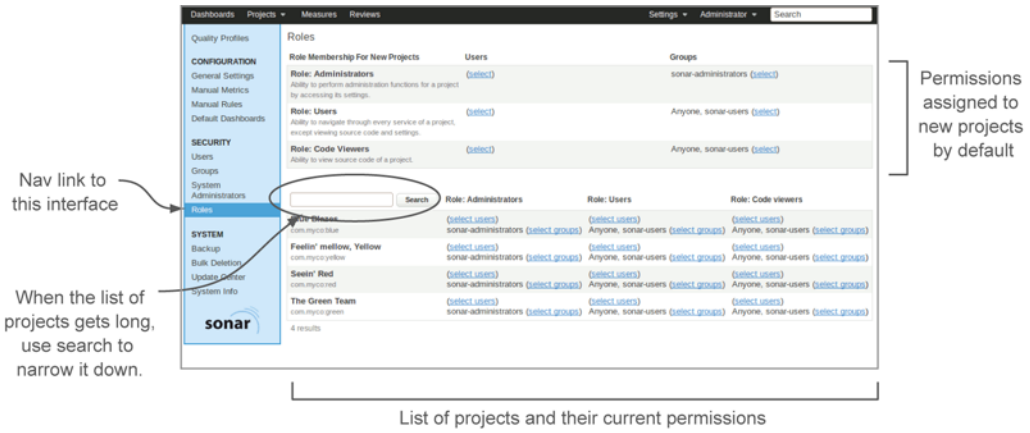


Figure 12.8 The Roles interface is where projects, users, and groups all come together, giving you a dashboard of who has what permissions on any given project.

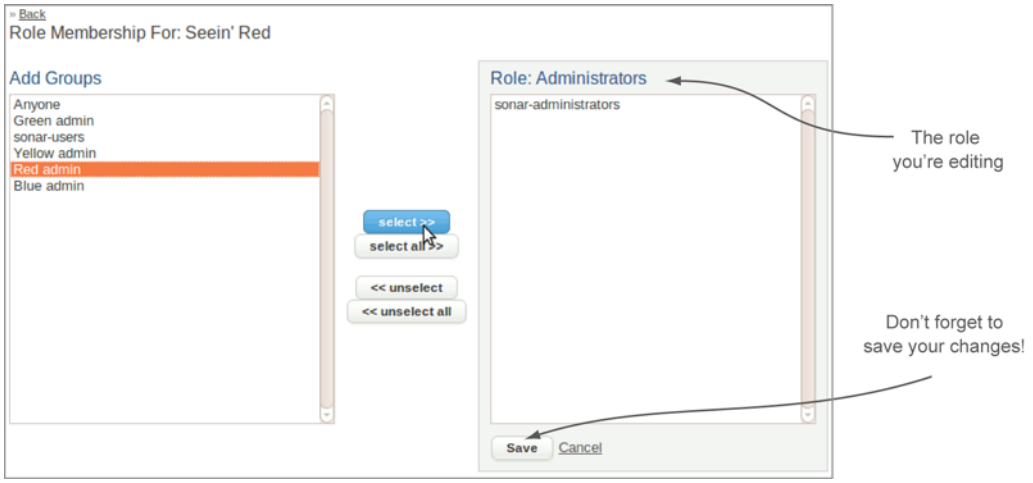


Figure 12.9 To add a group to a project role, highlight it on the left, use the Select button to move it to the right, and save your changes.

But who is this Anyone? *Anyone* is SonarQube-speak for *anonymous*, and it's a phantom group. You can't add members to it, but when you're administering group project roles, you see it in the list as a way to give anonymous users rights to a project. Anyone (anonymous) has the ability to view all aspects of your rule profiles; and as we said, by default, Anyone gets both User and Code Viewer privileges on all new projects.

Just what those privileges are comes next.

12.2.1 Project Administrator role

Russell needs to grant the project Administrator role, which gives you permission to do pretty much what you'd think—administer a project. We'll cover project administration in detail in chapter 15; but in short, it gives you access to twiddle all the project-level settings. That includes choosing your project's analysis profile, granting other users permissions to the project, and tweaking a project's analysis exclusions. With project Administrator rights you also get the ability to create the action plans we examined in chapter 10.

By default, this role is granted only to the sonar-administrators group. Thus only those people with global admin permissions have the lower-level permissions to administer a project—which is what's making Russell crazy.

By this point, he has created a Red Team admin group and put Susan in it. When he grants that group the Administrator role on Red Team's projects, Susan will see her new administration options the next time she logs in, as illustrated in figure 12.10. All that's left for Russell to do is put on his Tom Sawyer hat and convince her that she wants to do the work herself instead of having to wait around for him to do it.

Russell could certainly have granted admin permissions on Red Team's projects directly to Susan as an individual—you can do it either way. But the beauty of using groups is that if Susan moves to another team, he takes her out of the Red Team

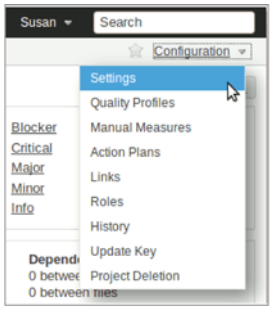


Figure 12.10 Project administrators have an additional menu of options—options that allow them to tweak every setting on a project.

admin group and adds her replacement. Compared to individual user permissions, a team swap like that works out to six of one, a half dozen of the other if there's only one project per team. But if you have multiple projects per team (or if you will in the future), the extra step of creating the group makes life much easier.

One last note before we move on to the other roles: when you're editing a project's list of admin groups, think carefully before you evict the sonar-administrators group from the list. There's nothing particularly special about the sonar-administrators group—it's just another group. If you remove it from a project's list of administrators, you remove the global administrators' ability to configure that project. That's not an irrevocable change, because you don't lose the ability to configure that project's roles, and it may be exactly what you want. But be aware of what you're doing.

Next we'll look at the User role.

12.2.2 User role

Having the User role on a project (as opposed to having a user account, which we showed you how to create earlier) lets you see the project—to know that it exists. If you don't have the User role on a project, you see something like figure 12.11 when you look at a project list that would normally include that project.

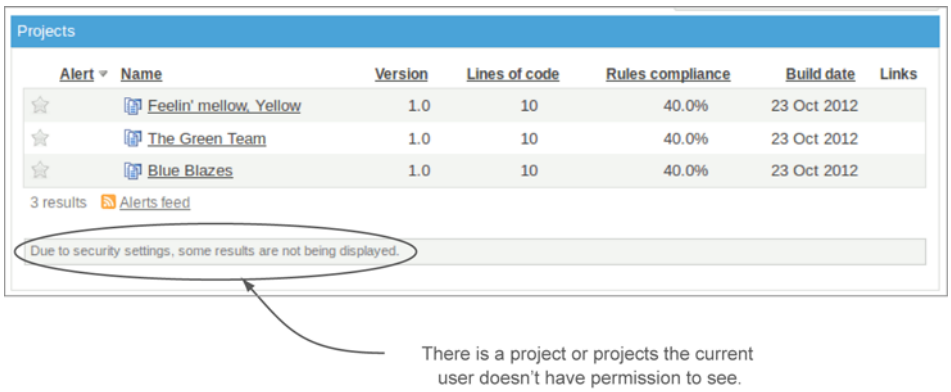


Figure 12.11 A project's users are the ones who are able to see it.

By default, the User role is granted to Anyone and the default group (sonar-users). It gives them the ability to see a project and navigate through every project interface, examining almost every aspect of the project.

To make a project private, you definitely want to remove Anyone and may want to remove sonar-users as well. But if you do, you should probably add back the sonar-administrators group as well as the group you've set up for the project-level administrators. Roles aren't hierarchical—having access to administer a project doesn't give you access to see it. Because you can't administer a project you can't see, you should make sure your project administrators also have the project User role—whether that's through Anyone or through some other group.

If you find that you've somehow locked yourself out of seeing a project, don't panic. Not having the User role on a project means you can't see the project in filters or get to its dashboards or drilldowns. But you can still see it in the Roles interface, and you still have the ability to re-grant yourself permissions to it.

The final role to consider is Code Viewer.

12.2.3 Code Viewer role

There's no mystery about what users with the Code Viewer role can do: they can see a project's source code. Logged-in code viewers (but not anonymous users accessing a project through Anyone's roles) can see the source code and also comment on it, or assign its issues for cleanup, or perform any of the other issue functions we covered in chapter 10, including marking issues as false positives or changing their severity.

Again, just keep in mind that there's no hierarchy to roles. Having the Code Viewer role without also having the User role is useless. But the reverse isn't true. We mentioned that the User role gives you access to *almost* every aspect of a project; viewing the code is the one thing it doesn't give you. Figure 12.12 shows the issues drilldown for a user who's not assigned the Code Viewer role.

Appendix B's list of analysis properties includes `sonar.importSources`, which accepts Boolean values and defaults to `true`. It controls whether the analysis process imports your source code into SonarQube. With source importing turned off, no one can see the code—not even code viewers—because it's not there. With source importing turned on, careful use of a project's Code Viewer role is the best way to control who can see the source code of sensitive projects.

12.2.4 Best practices for roles

Before we move on to global administration, we'd like to take a minute to talk about best practices. We've given you a couple so far: grant roles to groups rather than individual users, and create an administration group for each project or team that holds one or two members of that team. What we can't do is go beyond that in terms of stating best practices, because they're situational.

Work for the U.S. Navy on nuclear secrets? Then you want SonarQube locked down tight as a drum, and Anyone is definitely out. On an open source project? Then

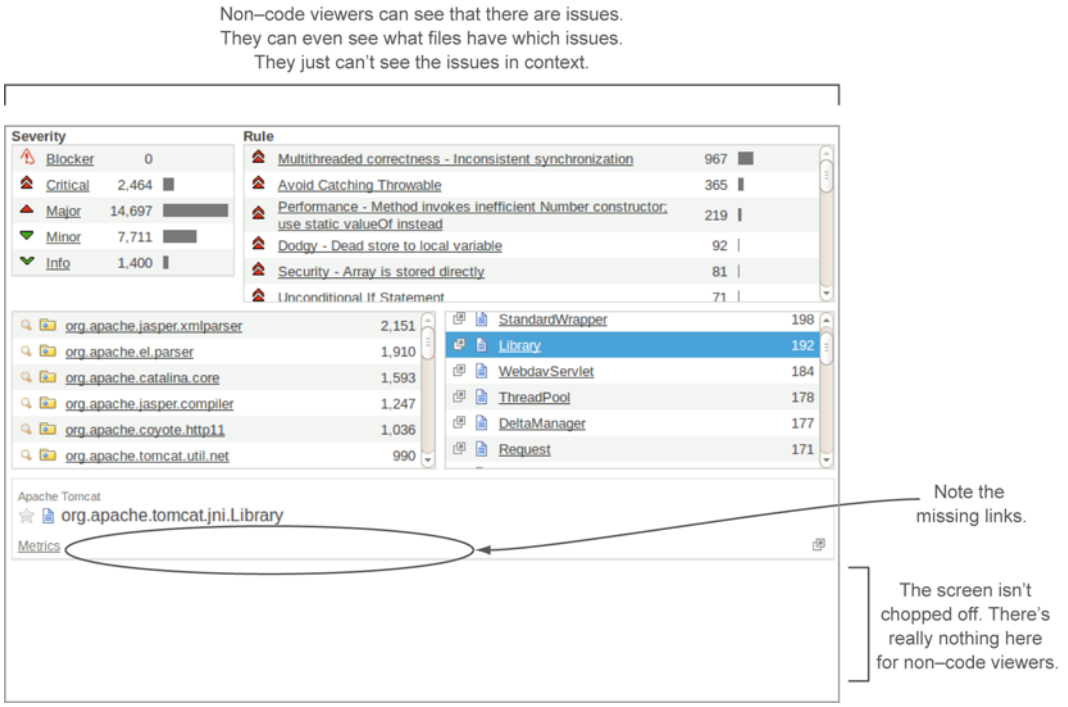


Figure 12.12 The User role grants access to view every facet of a project but one: seeing the code. That's what the Code Viewer role does.

let everyone see everything, and just be choosy about who gets an account (and therefore issue permissions) and about who you put in a project admin group. In between are too many permutations to list, so we won't even try.

Now that you know everything there is to know about project permissions, let's move on to SonarQube's global permissions.

12.3 System administrators

System administration in SonarQube is just another role: it gives you permissions, and you can grant it to either users or groups. There are two system administration–level roles in SonarQube: the Quality Profile Administrators role for profile editing, and the System Administrators role for general system administration. System administration will be covered in detail in chapter 14. But in short, system administration rights let you make all the security changes we've discussed in this chapter as well as install plugins, tweak all global settings, and administer all default dashboards and filters. The Quality Profile Administrators role lets you administer quality profiles (which we'll discuss in chapter 13).

By default, both system administration roles are granted to the sonar-administrators group, and the lone default member of that group is the Admin account you've been

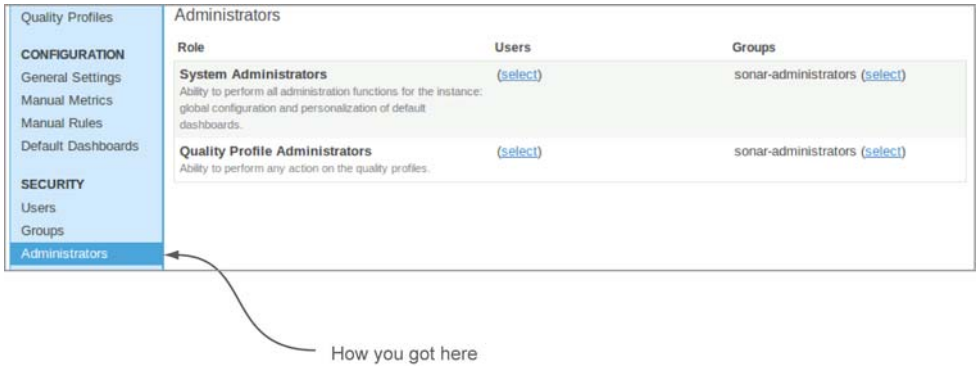


Figure 12.13 The System Administrators interface is the simplest of the security screens, with links to edit the lists of users and groups who have the privilege to make system-level changes.

using so far to make global changes. Now that you're up to speed on administering users and groups, it's time to change that.

Of course, you could continue using the default admin login, but at some point you should share administration rights with someone else. After all, everyone likes to take vacation now and then. That means either sharing the credentials to the admin account or granting individual users admin rights—preferably both to you *and* your stand-in. Why? Well, there's not a lot of audit trailing in SonarQube, but for what is available you should be able to trace which of you made what changes.

The first step is creating the user accounts, if you haven't already. Then add yourself and your stand-in to the sonar-administrators group. And you're finished, because out of the box, the sonar-administrators group has system admin rights. You could use a different group if you wanted; again, there's nothing special about either the default admin account—it's just another user—or the sonar-administrators group. But the sonar-administrators group is clearly named and already in place, so you might as well stick with it.

If you did want to use a different group or grant system admin privileges to individual users, you'd do it in the System Administrators interface, shown in figure 12.13. As you'll see, it's the simplest of the security screens.

Once you've given your vacation stand-in system administration rights, what he sees after clicking the Configuration link at the top of the page changes from the short list of user options to a much longer list that includes those options and a lot more. The discussions in the following three chapters will help you train your stand-in to run SonarQube in your absence so your vacation can be as peaceful and undisturbed as possible. But before we move on to that, let's look at this chapter's plugins.

12.4 Related plugins

The plugins for this chapter don't do anything nifty to the interface. They don't add metrics or rules. They don't aggregate data in new and interesting ways. They just inte-

grate with your local authentication system so you don't necessarily have to worry about manually creating users, and your users don't have to worry about remembering their system password *and* their SonarQube password. Whenever they log in to SonarQube, whether directly or through the Eclipse integration, they'll do it with their normal system passwords.

Most of these plugins are installed like the rest—through the update center—but you configure them by adding options to the $SONAR_HOME/conf/sonar.properties file rather than through the SonarQube interface.

Each plugin has its own options, but one is common to most of them: `sonar.authenticator.createUsers`. Set this to `true` to have users created in the SonarQube database on their first successful login. Set to `false`, this option means you have to manually create every new SonarQube user.

12.4.1 LDAP

The LDAP plugin lets you use the Lightweight Directory Access Protocol for user authentication. Once you have it up and running, it automatically synchs in your users' email addresses and names so you don't need to maintain them manually. The LDAP plugin also offers a fallback for the SonarQube user database, so you can use accounts created directly in SonarQube (such as the Admin account) even if they don't exist in your external authentication system.

You install the LDAP plugin through the update center. You can find its configuration details on its SonarQube wiki page: http://docs.codehaus.org/display/SONAR/LDAP+Plugin.

12.4.2 OpenID

The OpenID plugin lets you set up single sign-on through an OpenID provider. This means you can let users sign in to SonarQube with their Google accounts if you'd like. The configurations for this plugin are minimal: the `sonar.authenticator.createUsers` setting mentioned earlier, `sonar.security.realm`, the URL of the OpenID provider, the URL to refer users to once they've logged out, and the URL of the SonarQube server.

You install the OpenID plugin through the update center. You can find its configuration details on its SonarQube wiki page: http://docs.codehaus.org/display/SONAR/OpenID+Plugin.

12.4.3 Crowd

The Crowd plugin lets you delegate authentication to Atlassian Crowd. This plugin is one of the few authentication plugins that doesn't use the `sonar.authenticator.createUsers` option. If you use Crowd, you still have to manually create your users.

You install the Crowd plugin through the update center. You can find its configuration details on its SonarQube wiki page: http://docs.codehaus.org/display/SONAR/Crowd+Plugin.

12.4.4 PAM

The PAM plugin allows delegation in Unix/Linux and Mac OS boxes to the OS's underlying PAM authentication system. This is the only authentication plugin that's not installed through the update center—or at least not solely through the update center. There is a plugin to install, and it can be installed through the update center. But you also need to add a Java-PAM (JPam) jar to your $SONAR_HOME/bin/lib directory, which means you have an extra step at SonarQube upgrade time. Thus you should make sure you migrate not just your plugins and settings to the new version but the `jpam` library as well.

The PAM plugin has another distinction: it's one of the few authentication plugins that doesn't use the `sonar.authenticator.createUsers` option. If you use PAM, you still have to manually create your users.

The plugin has only a few configurations to set, and you can find them at its SonarQube wiki page: http://docs.codehaus.org/display/SONAR/PAM+Plugin.

12.5 Summary

In this chapter, we've shown you how to create and administer groups and users. We've explained the phantom group named Anyone—which is SonarQube-speak for anonymous—and looked at the difference between what Anyone and a logged-in user can do.

We've explained the three project roles—User, Code Viewer, and Administrator—along with what permissions they give and the fact that there's no hierarchy among the roles. Having the Administrator role on a project does you no good if you don't also have the User role on that project, which allows you to see the project in the first place. And we hope we've convinced you that life will be far easier in the long run if you restrict your role-granting to groups rather than individual users.

You've learned how to grant system administrator permissions and why you should do that for your own user account and for a stand-in or two, rather than just passing around the credentials to the default Admin account. You've seen that Admin is just another user account and sonar-administrators is just another group.

And finally, you've seen that there are integration plugins that allow you to delegate authentication to your existing authentication system, lightening your administration burden (which is always a good thing, right?).

Now that, like Russell, you know how to share administration privileges judiciously, rather than handing around the keys to the kingdom, the next three chapters will take you in depth into what your administration options are and how to use them.

13 Rule profile administration

This chapter covers

- Making your own profile: copy and modify
- Profile inheritance
- Rule editing
- Alerts: knowing when your metrics have crossed the line
- Tracking profile changes
- Administrative miscellany

A lot of things happen during an analysis. Your code is examined for duplications, complexity, and a host of other things. Most important (to this chapter, anyway), it's measured against a set of rules: a *rule profile*. Out of the box, you get at least one rule profile for each language SonarQube can analyze. Those default rule sets make great starting points; but as you've probably realized by now, they're not perfect for every situation. In fact, almost every coder has a bone to pick with at least one rule in any given rule set, whether it's that the severity is wrong, or the rule threshold is too low, or it shouldn't be in the profile at all.

In this chapter, we'll show you how to go about fixing that. You can edit the default rule sets, but we'll explain how to peel off a copy and start customizing it to meet your needs. But it's not just rule sets that are customizable. You can augment the global description of any rule and make profile-specific notes on rules. You can also edit some rules, and we'll show you how. We'll also cover how to check a profile's change log and how to compare one profile to another.

Profile inheritance is another useful feature. It allows you to set up a base rule set and then augment child profiles with the additional rules needed for a specific project or team.

Although profiles are primarily about rules, you can also use them to set *alerts*: simple Boolean thresholds that help you raise the alarm if any of your metrics (even the ones that have nothing to do with rules or issues) cross the warning or error thresholds you set.

13.1 Making your own profile: copy and modify

As Russell deleted yet another demanding email, he mused that being the SonarQube admin wasn't always a life of wine and roses. Not one team was satisfied with the rule profile he had set as the default, and the natives were getting decidedly restless. But although all the teams disliked the rule set, that was the only thing they agreed on. Red Team wanted it stripped down to just the most important rules so they could focus on the worst offenders. Green Team wanted to add rules, and Blue Team wanted to pick and choose—add some and delete others—and twiddle severities to boot! Meanwhile, management was adamant that everyone needed to adhere to the same minimums.

He'd been putting it off, but Russell realized it was time to dig in to rule profile administration. Because everyone's basing their requests on the current default profile, working from a copy or copies of that profile is a great strategy. A more sophisticated approach, which we'll look at in the next section, might have Russell create a base profile, perhaps based on the minimum rule set that Red Team and management wants, and set up team-specific child profiles that inherit the basics from the core rule set. Whatever strategy Russell uses, his first step should be to create a new profile.

13.1.1 Copy or start from scratch?

There are two ways to make a new profile. You can create one from scratch or copy an existing one, which is supremely easy (as shown in figure 13.1).

Each profile is displayed with a Copy button. Provide a unique profile name in the resulting pop-up, and you're in business. If you'd rather start from scratch, that's easy too. Just click the Create link to the right of the language in question. A creation form pops up, as shown in figure 13.2.

Depending on the language, the Create Java Profile form may include upload inputs for selected rules engines. The formats of the files you need to feed into these engines are rule-engine-specific. We'll look later at how you can export the rules from

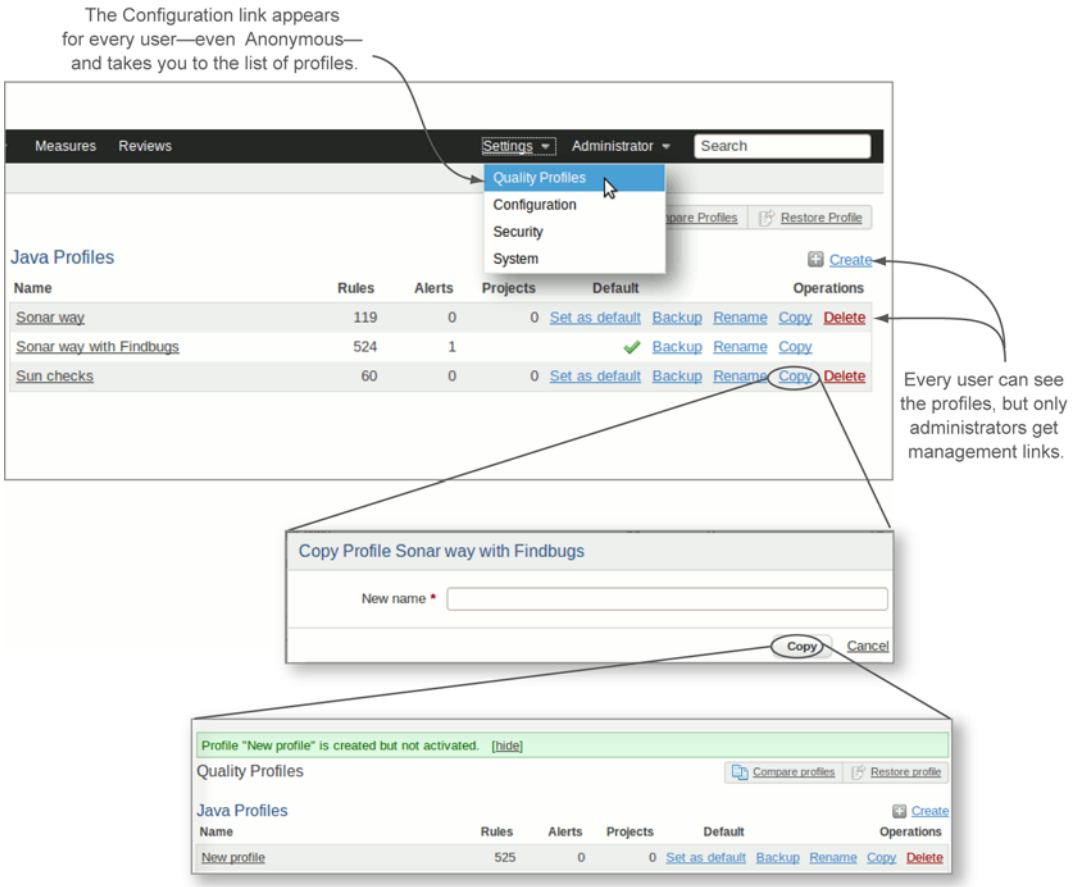


Figure 13.1 **Clicking the Configuration link at the top of the page takes you to a list of rule profiles. Logged-in administrators can easily copy any profile with the click of a mouse. The only hard part is coming up with a good name for the new profile.**

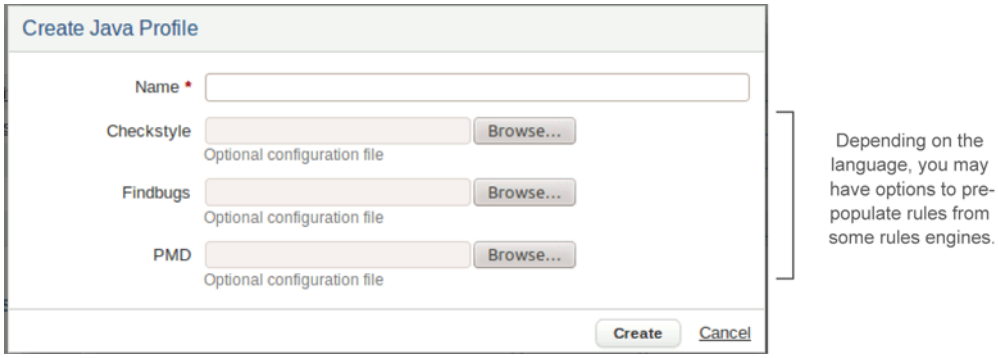


Figure 13.2 **You create a profile from scratch via a form that's added inline on the page. Depending on the language, you may get the opportunity to seed your new profile with rules from some of the language's available rules engines, uploaded in an engine-specific format.**

an engine from one profile for import into a new one. When you submit the form, your new profile is added to the list.

Once you have your new profile, it's time to begin editing. Click-through on the profile name to land at the Coding Rules tab of the profile editing interface; this is its list of rules. If you started from scratch, you're looking at a blank list (remember, by default this page shows the profile's active rules, and you haven't added any yet) with the search form at the top. If you copied another rule set, then there are plenty of rules to look at here.

As we've said before, any user, even Anonymous, can get to the list of rules in a profile. But for administrators looking at a modifiable profile, the rules are presented in an editable search results format, as shown in figure 13.3.

13.1.2 Your first profile edits and their quality implications

By default, the Coding Rules tab shows all the rules currently active in your profile. Each rule is displayed with two controls: a check box and a drop-down menu. Scroll

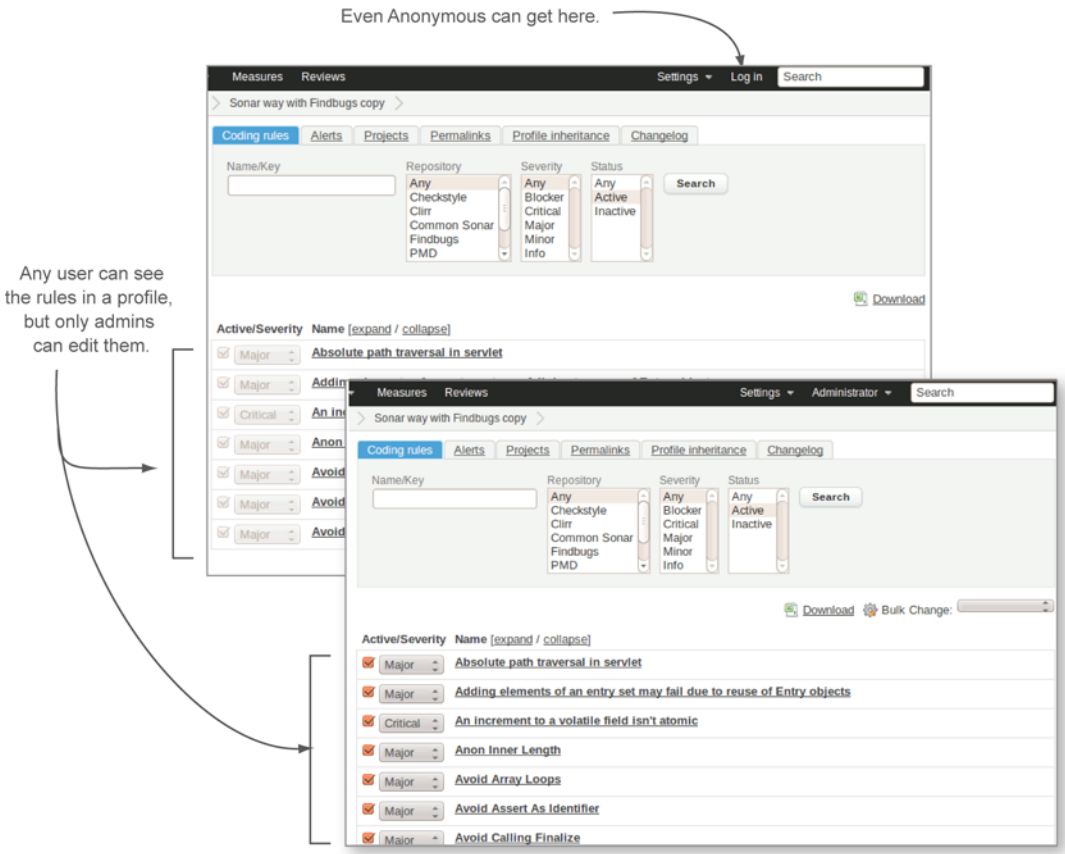


Figure 13.3 The rules that make up a profile are presented in a search/search result interface. The default presentation is all rules in the profile.

up and down the page, and you see that the check box to the left of every rule is checked, indicating that it's part of the current profile.

To drop a rule from your profile—one of the two things most people are itching to do at this stage—uncheck it. There's no submit button on this interface; your changes take effect immediately, but the search results aren't refreshed interactively. If you remove rules from your profile by unchecking them, they stay on the page until you refresh it—giving you the chance to turn them back on if you mis-clicked.

The other thing most folks are eager to do is adjust rule severities. Nearly everyone thinks the default severity on some rule is a bit of an overreaction. Rule severities are set per profile, so if you want a rule's severity changed everywhere, you have to do that manually. But it's a simple enough change to make, as figure 13.4 shows.

The drop-down menu to the left of each rule lists all possible severities, with the rule's current severity in the current profile highlighted. To change a rule's severity, pick a different option in the drop-down list. Again, your changes are recorded immediately.

Don't think that dropped rules or changed severities will be immediately reflected in your projects' issues drilldowns or metrics, though. You'll need to re-analyze each affected project to see those changes in action. Depending on your issue counts, these edits can produce what look like wild swings in quality, as, say, 50 Criticals are "fixed" while 50 Majors are "added" because of a severity adjustment, and another 20 Minors are "fixed" because of a dropped rule. In reality, the code hasn't changed a keystroke—only the way you look at it has.

For the next couple of weeks, maybe the next few months, you'll remember what caused that wild quality swing. Next year? Maybe not. Fortunately, analysis with a new profile version (yes, rule profiles are versioned; more on that soon) is marked as an event in SonarQube. We talked about events in chapter 8; they're special flags that mark analyses where something memorable happened. So your next analysis, with your newly edited profile applied, will be flagged with a profile-change event, noting the profile name and version. Because analyses with events are exempt from automated database cleanup, you'll have a long-term record of both when and why that quality swing occurred.

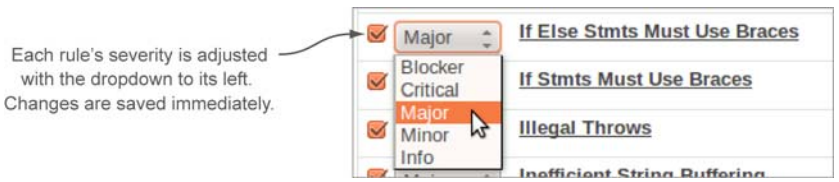


Figure 13.4 **Use the drop-down menu to the left of any rule to adjust its severity in that profile. Note that no other profile is affected by this change.**

13.1.3 Adding rules: how to find them and why you'd want to

The third kind of change you might want to make is adding rules, but first you have to find them. As you scroll up and down the page, you'll see that there are no unchecked (*addable*) rules available. Remember, that's because by default, this page shows all the rules in the current profile, and only those rules. Finding the others is easy, though.

Scroll back up to the search interface at the top of the page, as shown in figure 13.5. To find the rules that aren't currently in your profile, search by Status Inactive, and you should have plenty to choose from. Enabling rules in your profile is intuitive; just check the check box. But you may be wondering why, with over 500 rules currently in your profile (assuming you copied from the SonarQube way with FindBugs profile), you would want more.

Believe it or not, there are a number of reasons to turn on more rules. For instance, various plugins operate through additional rules, such as the Tag List plugin we discussed in chapter 10. It tracks instances of the `//TODO` comment for you, but only if you've enabled the relevant rule in your profile.

You may also want to investigate the rules that aren't on by default, such as those available for many languages to let you track some of the non-issue metrics: duplications (the Duplicated Block rule), test coverage (Insufficient Line Coverage by Unit Tests), complexity (Avoid Too Complex Method, Avoid Too Complex Class), and so on. Even beyond the Seven Axes of Quality, the non-default rules offer useful options, such as the Avoid Deprecated Method rule, which can help you hunt for and destroy these incompatibilities waiting to happen. There are also a lot of potentially valuable rules that don't fit neatly into a single category, such as Avoid Multiple Unary Operators, For Loop Should Be While Loop, and the rules available for most languages to record parsing failures so you don't have to pore over your analysis log every time to

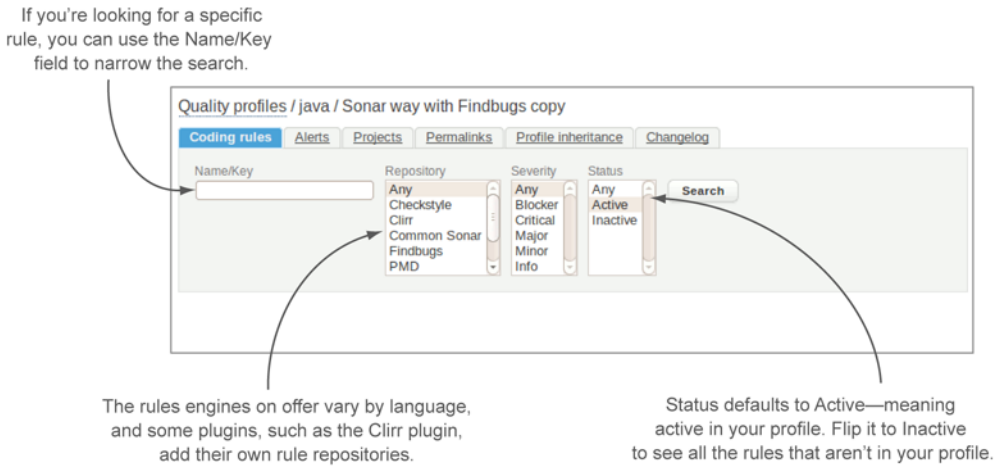


Figure 13.5 The rule search interface is straightforward. You can search by any combination of rule name, rule engine, severity, and status. It defaults to showing all active rules in the current profile.

make sure SonarQube was able to read all your files. Even if you don't think you're in the market for extra rules, they're well worth at least a browse.

Now that you've seen how easy it is to get started with your own profile and make changes to it, you may be ready to go to town. But before you go too far down the road, you should look at profile inheritance if you have multiple profiles to manage for a given language.

13.2 Profile inheritance

Remember, Russell was facing three different teams wanting three different rule sets: Red Team wanted a stripped-down set of just the "most important" rules; Green Team wanted to add rules; and Blue Team wanted to remove some rules, add others, and change some severities.

Of course, one option might be to have each team manage its own profile. But as you saw in chapter 12, only SonarQube administrators can edit rule profiles, and global admin permissions probably aren't something you want to hand out like candy.

Fortunately, you don't have to. Because Red Team and management concur on the minimum rule set (the Blockers and Criticals from the Sonar way with FindBugs profile), the Red Team profile is a good candidate to be the parent profile. In this section, we'll show you how to set up and manage inheritance and how to track the state of the relationship.

13.2.1 Establishing inheritance

If Red Team only wanted a handful of rules, creating it from scratch might be the way to go, especially because there's a Bulk Change feature that lets you toggle the status of every rule on the page at once. But the Sonar way with FindBugs profile includes hundreds of Blockers and Criticals: *nearly* every single one. Because a couple of Criticals don't belong in the profile, the easiest first step is to copy the Sonar way with FindBugs profile, search for the rules to drop from the copy, and then do a bulk change to deactivate them all in one fell swoop, as shown in figure 13.6.

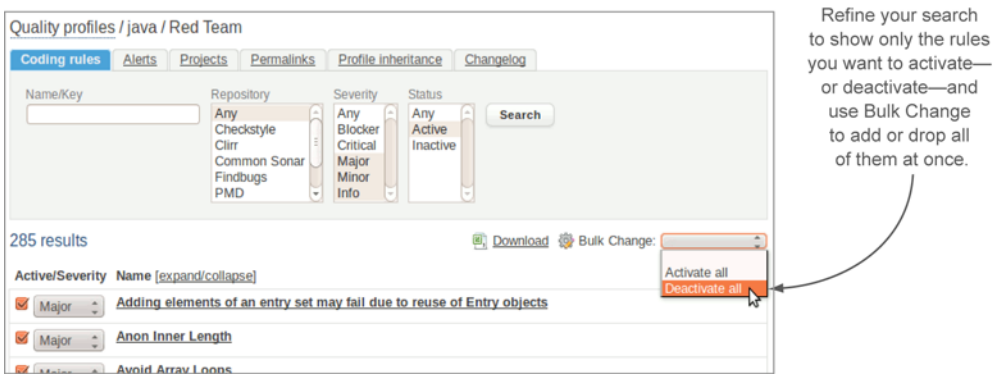


Figure 13.6 Each selection in the rule search accepts multiple options. Directly under the search form, a Bulk Change drop-down menu lets you activate or deactivate every rule on the page at once.

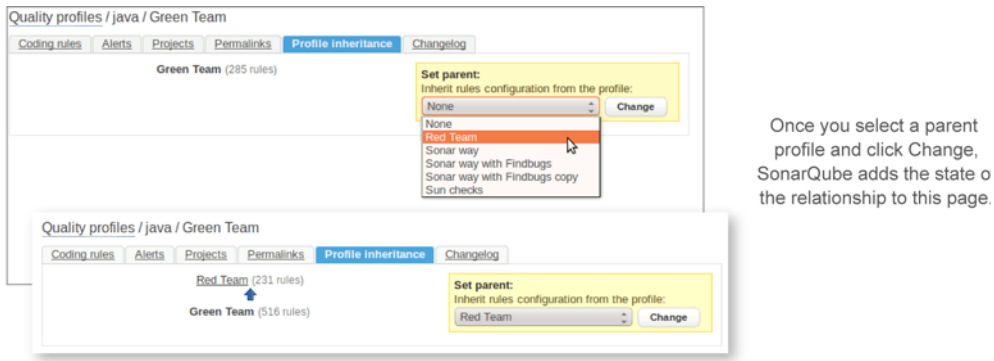


Figure 13.7 The Profile Inheritance tab lets you set a parent profile. It also reports on a profile's lineage, including the number of rules the current profile inherits.

Now that a parent profile is in place, you can follow a similar copy/modify strategy to create Green Team's profile. Start again with a copy of Sonar way with FindBugs, and then click-through to the list of rules and deactivate all the Blockers and Criticals: all those that will be inherited from the Red Team profile. Then click the Profile Inheritance tab. By default, this part of the interface is pretty spare, except for the yellow editing box at right where you can set a profile's parent (as shown in figure 13.7).

Once you've set a parent profile, the Inheritance tab shows how many rules the profile has in total and how many of those rules are inherited. From here, head back to the Coding Rules tab. The interface has changed a bit, as shown in figure 13.8.

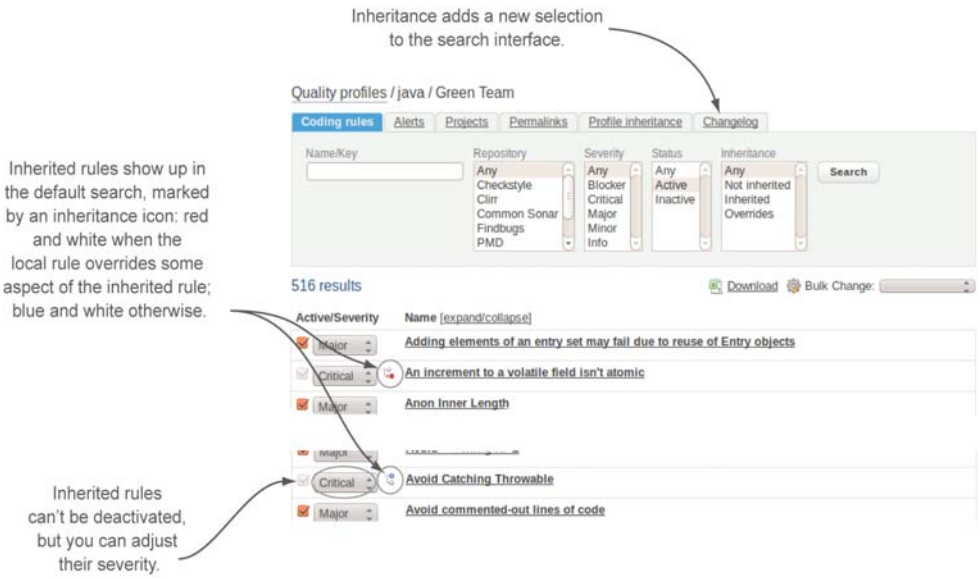


Figure 13.8 Inherited rules appear in a profile's rule search. You can't deactivate them, but you can change their severity. If you do change an inherited rule's severity, its inheritance icon changes to one that's partially red.

13.2.2 Managing the relationship

Starting from the top, you now have an extra selection in the search interface: Inheritance. Among the rules, you now see not only the rules explicitly included in the profile, but also the ones inherited from the parent profile. Those inherited rules are denoted with a blue and white inheritance icon to the left of the rule name, and they can't be removed from a child profile (the check box is greyed out), but you can still adjust their severities. If you do adjust the severity of an inherited rule, it's reported inline with a different inheritance icon: this one is red and white to show that the rule is out of synch.

The total number of overridden rules also appears on the Profile Inheritance tab, which displays not only a profile's parent but its children as well. Figure 13.9 demonstrates that with an updated shot of Red Team's profile inheritance. We've added a stubbed-in profile for Blue Team and given Red Team a parent: a Management Base Set profile, into which we've moved all the Blockers.

If you decide you'd like to revert an overridden rule to the parent profile configuration, click the rule name to expand the rule details section. (When you click a rule name in the issues drilldown, you get a pop-up, but here it's a different behavior.) We'll cover most of what you see in the rule details section in a moment. The thing to focus on here is the grey block, which you can see in figure 13.10.

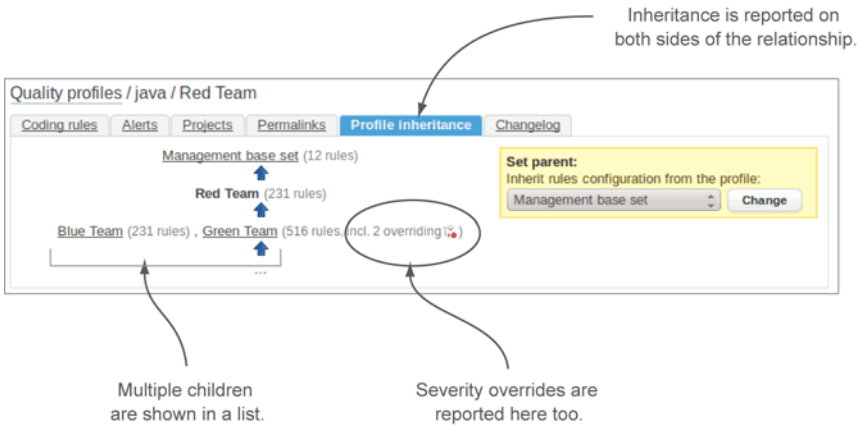


Figure 13.9 The Profile Inheritance tab reports not only on the inheritance relationship, but also on the state of it, showing the total number of overridden rules.

Figure 13.10 The rule details for an inherited rule contain a summary of that rule's inheritance status in the current profile. If you've overridden some aspect of the rule, such as severity, you can realign the rule with the parent profile using the Revert to Parent Definition button, which only appears conditionally.

13.3 Rule editing

Now that your profiles are set up, there are a couple other ways you may want to tweak their rules. In addition to adjusting a rule's severity, you can make up to three other types of edits: you can add a global extension to the rule description; add a profile-specific note; and, in some cases, adjust the rule's parameters.

To make those edits, start again from your profile's Coding Rules tab and click the rule name to expand the rule details section. At the top of the section, you see the same rule description that's in the pop-up you get when you click a rule name from the issues drilldown. Below that are a few links, and maybe some inputs.

The links are Extend Description and Add Note. The inputs are rule specific, and we'll look at them first.

13.3.1 Customizing individual rules: editing rule parameters

The rule sets for most languages offer a handful or so of rules that accept configurations. For Java, it's perhaps a double handful. Unfortunately, no reliable rule of thumb

Figure 13.11 Rules that cite specific numbers are often configurable, but they're not the only configurable rules.

lets you know ahead of time whether a rule is configurable, especially from the issues drilldown, which is where you're likely to spend the most time looking at rules.
If you see a rule that cites a number, such as the Cyclomatic Complexity rule shown in figure 13.11, it's highly possible that the number is configurable.

A rule doesn't have to cite a number to be configurable. The only way to know for certain is to look at the rule details section in the Coding Rules tab. If the rule is configurable, its details section includes a form. There is one shortcut to finding all the configurable rules in your profile: use the expand/collapse link at the top of the rule list to show all the rules' details at once and skim down the page, looking for form inputs. Figure 13.12 shows a couple of editable rules.

Once you find a configurable rule, how to make the configuration change should be pretty obvious from the context of the rule, the inline explanations next to the field, or both.

Rule parameters are one place you might thank profile inheritance (if you've chosen to use it). The parameter edits you make in a profile are unique to that profile; and

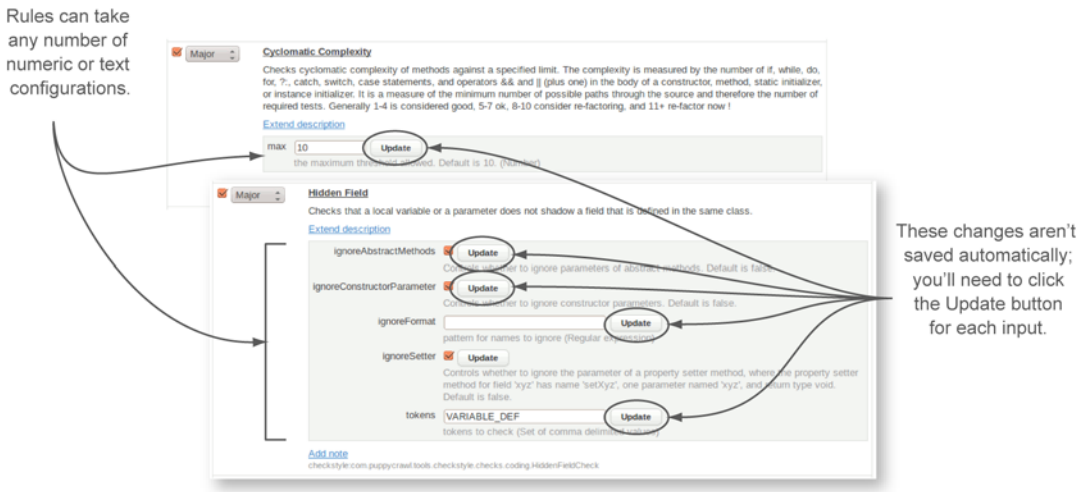


Figure 13.12 The variety of possible rule configurations is as wide as the rules they configure, from simple integers and Booleans to regular expressions, method names, and package names.

every other profile can have those parameters set to different values. But parameters set in a parent profile will be inherited by the child, just like the rule is. And just like the severity of an inherited rule, an inherited rule's parameters can be overridden at the child level, marking the rule and the profile as out of synch with the parent.

As you edit your rule parameters, there are only a couple of things you need to keep in mind. First, unlike every other edit you've seen so far on this page, these changes *are not* saved automatically. You must explicitly click the Update button for every single field you change. The second thing is something you can probably predict at this point: you need to run a new analysis to see your changes take effect.

13.3.2 Cookie-cutter rules: the ones you can duplicate

There's a subset of editable rules that you probably won't want to edit directly, although you can if you like. They're the *copyable* rules, and typically they're copyable because you'll want multiple variations on the configuration.

To try this, find a copyable rule. Again, there's no way of knowing whether a rule is copyable without looking at its details, but the trick of expanding the rule details works here too. Click the link, and then do a find on the page for *Copy*. Or you can use the rule we did, Comment Pattern Matcher, which is the rule you need if you want to use the Tag List plugin mentioned in earlier chapters. You should be able to find it pretty easily by entering `comment` in the Name/Key search input. Just be sure all the selects are set to Any.

When you've found the rule and expanded its details, a Copy Rule link appears at lower left. Click it, and you'll see that the process is pretty transparent, as shown in figure 13.13. Typically, you make copies of this rule to find `//TODO` and `//NOSONAR` comments. We've created a copy for a third comment, `//FixMe`, which would be relevant only if it was a something your team used regularly.

Note that creating a rule copy doesn't automatically enable it in the profile you were working in. That's because in making that copy, you haven't made a change that's local to that rule set. Instead you've created a new, globally available rule. You can enable it in your current rule set. Or not. You can also choose to enable it in every other rule set you have for that language. And if you decide not to enable it in any of them, that's fine too. It'll still be waiting for you in the inactive rule list when you're ready to use it.

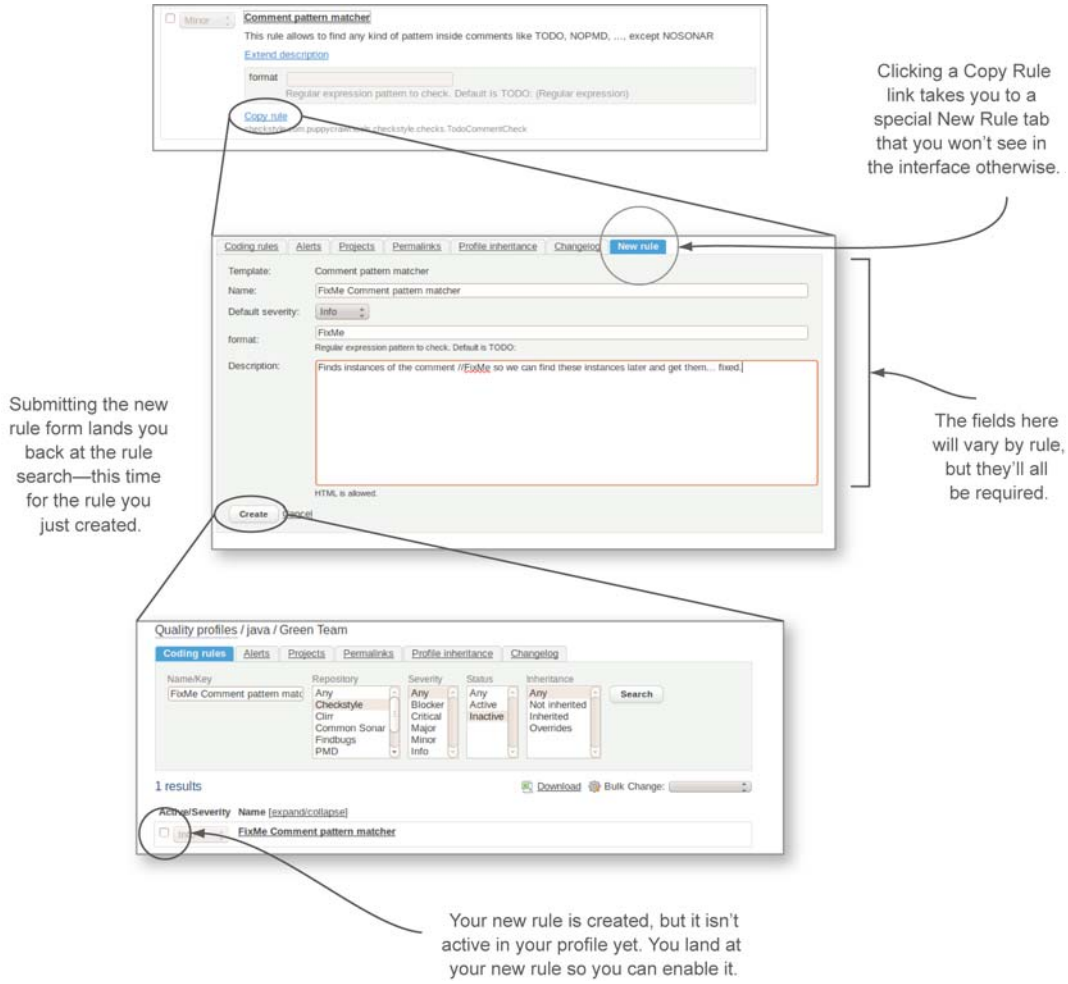


Figure 13.13 Copying a rule is a straightforward process, from the Copy Rule link to filling in the rule details (including the description that will be used in the rule's pop-up). After you've created the rule, you land back at a search result for it so you can enable it in your profile.

If you want to edit the parameters of your rule copy, the interface is the same as for any other editable rule. But when you expand the rule details, you'll notice an extra option on these rule clones: Edit Rule, as shown in figure 13.14.

In addition to having the option to edit all the fundamentals of a copied rule (including the name), you also have the option to delete it. What's that? You've already got issues racked up against your copy? No problem. Nothing blows up when you delete the rule, and the issues against it go away with the next analysis.

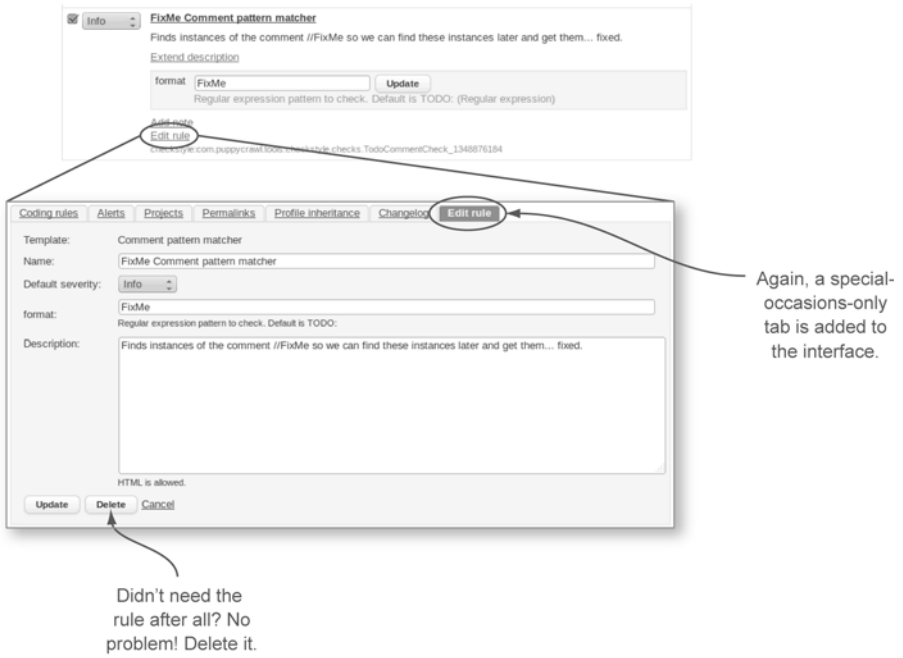


Figure 13.14 The process of creating new rules by copying is a forgiving one. Want to change the default severity? Use the Edit Rule link. Ditto the name and the description. If you decide you didn't need the rule after all, you can always delete it.

13.3.3 Extend Description: the rest of the story

Using the Extend Description link you'll find on every rule makes a global change, similar to duplicating rules. It affects all profiles, not just the one you're working in.

Why would you want to extend a rule description? Poke around in SonarQube long enough, and you'll find a rule with a description that's useless at best. For instance, the description of the Parameter Assignment rule is "Disallow assignment of parameters" with no indication of why that's a bad idea. Another reason to extend a description would be to add specifics about your environment.

To get started, click Extend Description. Doing so expands an editing form into the page, as shown in figure 13.15.

Again, these changes aren't auto-saved; you have to explicitly submit them. Once you do, they're reflected everywhere immediately, both when you browse to other profiles and when you click a rule name from the issues drilldown. You can also edit your extended description from any profile, not just the one where you originally extended it.

13.3.4 Notes: profile-specific records on individual rules

As you edit your profiles, adding rules and adjusting parameters, you may find yourself wishing you could annotate why you changed a certain parameter or why you

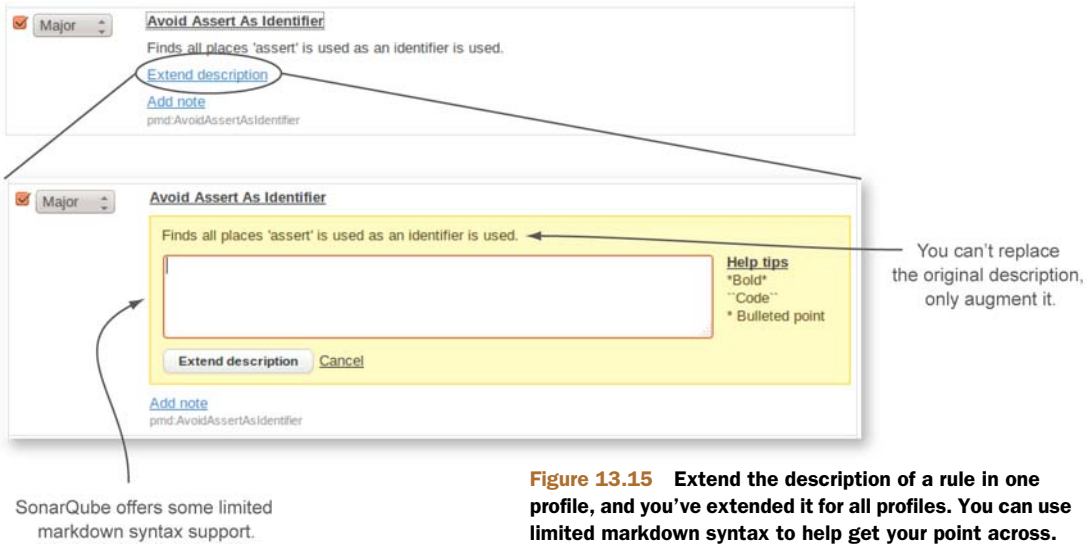


Figure 13.15 Extend the description of a rule in one profile, and you've extended it for all profiles. You can use limited markdown syntax to help get your point across.

included a given rule in the profile. That's what the Add Note link is for. Using it adds a profile-specific note that's displayed in the note details, as figure 13.16 shows.

Rule notes have a very limited life. They only appear in the profile administration interface; they aren't added to the rule's issues drilldown, because they're intended to be an admin's "note to self" sort of annotation. And because they're *profile-specific*, dropping a rule from your profile drops your notes on it as well. Further, notes made on a rule in a parent profile aren't inherited by child profiles, and adding a note to a child profile does *not* mark an inherited rule as overriding.

At this point, you know how to make all the rule-related changes that are possible in a profile. Next we'll look at alerts, which aren't related to rules at all.

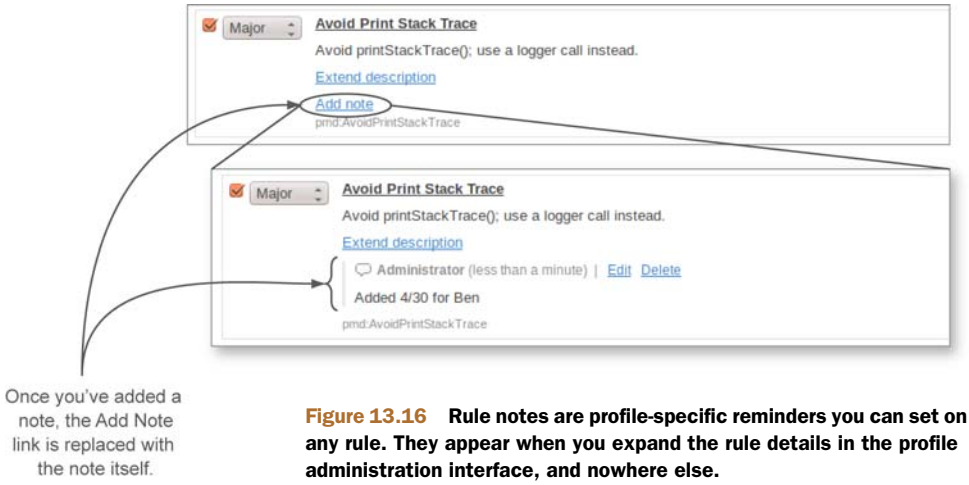


Figure 13.16 Rule notes are profile-specific reminders you can set on any rule. They appear when you expand the rule details in the profile administration interface, and nowhere else.

13.4 Alerts: knowing when your metrics have crossed the line

In addition to modifying the rules in a profile, you can also set alerts, which are metric-based warning and error thresholds. When you set an alert on a profile, any project analyzed with that profile will have an alert raised against it if it crosses one of those thresholds.

Even though a profile is primarily about rules (and therefore issues), you can set an alert against any metric: issues certainly, but also duplications, documentation, complexity, and so on. Alert setup and maintenance is found on the Alerts tab of the profile maintenance interface, as shown in figure 13.17.

You have the flexibility to set either warning or error thresholds, or both at once. To set one up, you choose the metric to monitor, the Boolean operator to use, and the threshold to test. If you do, you may be surprised to see a colorful new widget appear on your dashboard. The alerts widget isn't displayed on projects that are analyzed with alert-less profiles, but it will pop into view after your first analysis once you've set alerts. Its background color gives you at-a-glance insight into a project's alert status, and its contents provide the specifics, as shown in figure 13.18.

In addition to the colorful presentation of the alerts widget itself, other widgets may start to show additional color once you've set alerts. As the right side of figure 13.18 shows, a metric that has crossed an alert threshold is displayed on a colored background to draw your attention to the problem. But you only see red (error) and yellow (warning) backgrounds on these metrics; acceptable metric values aren't presented on green backgrounds.

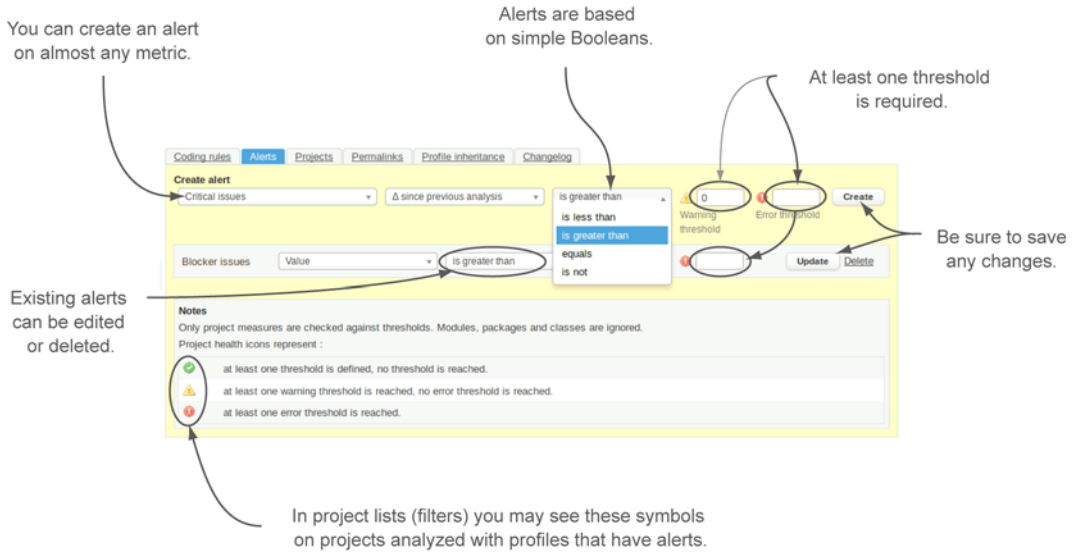


Figure 13.17 You can set as many alerts as you'd like. They're simple Boolean tests against metric thresholds.

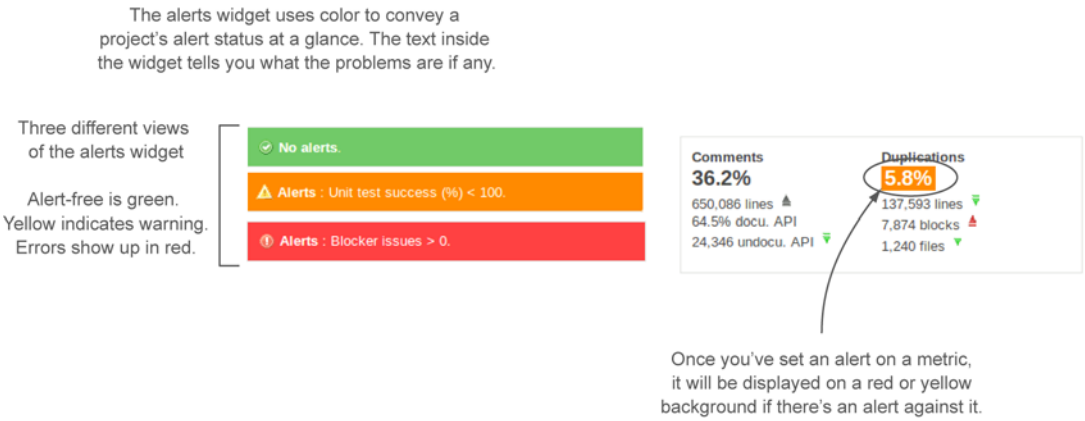


Figure 13.18 SonarQube gets more colorful once you've set up alerts. The alert widget, shown three times on the left, uses a stoplight scheme to convey alert status. Any other widget that displays an alerted metric will also use color if the value has crossed an alert threshold, as shown with duplications on the right.

Setting and seeing alerts is easy, but choosing alert metrics is a little more tricky. Some alerts are probably obvious: `Blocker issues > 0` is an attractive alert because Blockers need immediate attention. But what about Criticals? Clearly you don't want *any* issues in your project, but from a practical standpoint can you set up an alert on `Critical issues > 0`? For a brand-new project it may work; but set that alert on a profile that's used by legacy projects too, and you're likely to see your dashboards light up like Christmas trees. Sure, Criticals need attention too, but alerts are like salt: best used in moderation.

Fortunately, SonarQube offers the ability to set differential-based alerts, so that you can raise an alert when the number of Criticals *increases* versus any of your differential periods. Percentage metrics offer another handy option. In chapter 8, we advised you against using percentages for goal setting because they fluctuate based on project size, but they make good alert candidates because they can help you balance that brand-new, probably tiny project against your lumbering giants of legacy projects. Going back to Criticals, there's no Critical % metric to set an alert on, but there is a Rules Compliance Index metric, which factors the size of the code base against the number and severity of its issues.

Also looking backwards, you may remember that chapter 9 talked about using the Build Breaker plugin to mark your continuous integration build as failed if the SonarQube analysis raised any alerts. That way you can take advantage of the notification mechanisms available in most CI platforms to raise the alarm for new alerts, rather than having them wait passively (if colorfully) in SonarQube to be found.

Now that you know what alerts are, how to set them, and how to recognize a metric value that has raised an alert, we'll move on to how to track all the changes you've been making to your profile, including what is tracked and what *isn't*.

13.5 How to track profile changes

You've edited your rule set, edited the rules themselves, and probably set some alerts. At the beginning of this chapter, we advised you to start your profile journey by making a copy of your favorite profile and modifying it. At this point, your copy could be pretty far afield from where you started. Or maybe not. There are two ways you can keep up with those changes. The profile changelog gives you a granular record of changes to a profile, and the ability to compare two profiles will give you a summary of how far you've diverged from the source.

13.5.1 Changelog: who did what, when

Not surprisingly, you'll find the granular log of your changes under the Changelog tab in the profile editing interface. The changes recorded here are the ones that would affect an analysis. So setting an alert doesn't appear because it doesn't impact the metrics that come out of analysis; it just reports on them. Neither will adding a rule note or extending a rule description.

What does appear here are the times when you've added or removed a rule, and when you've changed a rule's severity or parameters—when you've made changes that will affect an analysis. Each of those changes is recorded granularly, so adding a new rule to your rule set and then adjusting it from the default severity appears as two distinct items in the changelog, as shown in figure 13.19.

Interestingly, profile inheritance changes are reflected in the changelog, but not in the manner you might expect. Recall that Russell has set up several profiles. The Red Team profile is a conservative set of only Blocker and Critical rules. The Green Team profile inherits from Red Team and adds rules of lower priorities as well. When that inheritance relationship is established (when he picked the Red Team profile as

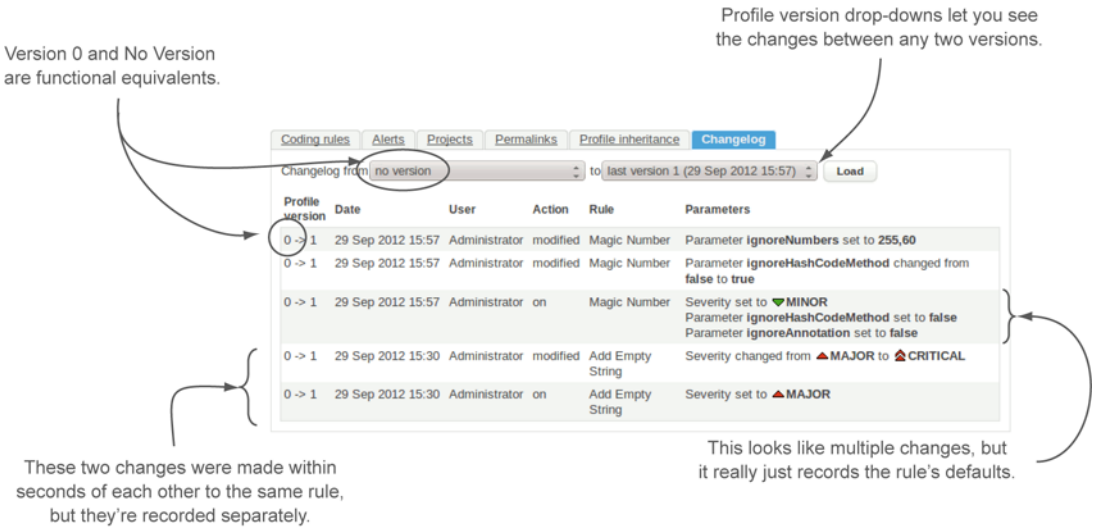


Figure 13.19 Each change you make to a rule set is recorded granularly.

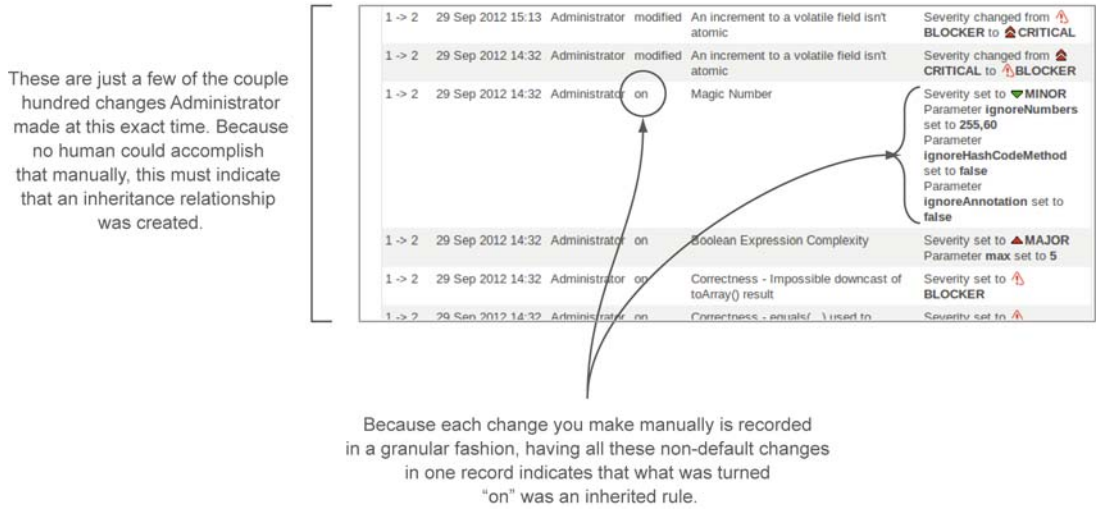


Figure 13.20 Setting or revoking an inheritance relationship between profiles isn't explicitly recorded in a changelog, but you can nonetheless find clues to a relationship. For instance, the addition or deletion of a couple hundred rules in the space of a minute is a strong indicator. An irrefutable one is the presence of multiple changes in a single record, because SonarQube doesn't record changes made manually in that way.

the Green Team's parent profile), what was recorded in the changelog was *not* the establishment of that relationship, as you can see in figure 13.20. Instead, the Green Team changelog shows (granularly) the addition of every single rule in the Red Team profile. In fact, the relationship between the two profiles isn't explicitly reflected in the changelog, because the relationship has no impact on an analysis. The rules that come with that relationship do, so they're what's recorded.

Similarly, if you override some aspect of an inherited rule or revert the rule back to the parent definition, the change is recorded, not the impact on the relationship. Even so, the changelog holds clues to the relationship, if you know what you're looking for, as figure 13.20 shows.

13.5.2 Profile versions: when changes go into production

Now that you've seen a couple of changelog excerpts and you understand what is recorded in a profile version, you may be wondering about the versions themselves. Profile creation is version 0, and your initial changes afterward go into version 1. But when does version 1 end and version 2 begin?

The version number changes when you run an analysis. That's because profile changelogs aren't about the profiles at all. Instead, they're about the changes you've made to a profile that will affect an analysis. Just as you'd change a program's version number when you put changes into production, a profile's version number is incremented when its changes go into production: when you perform the next analysis with it.

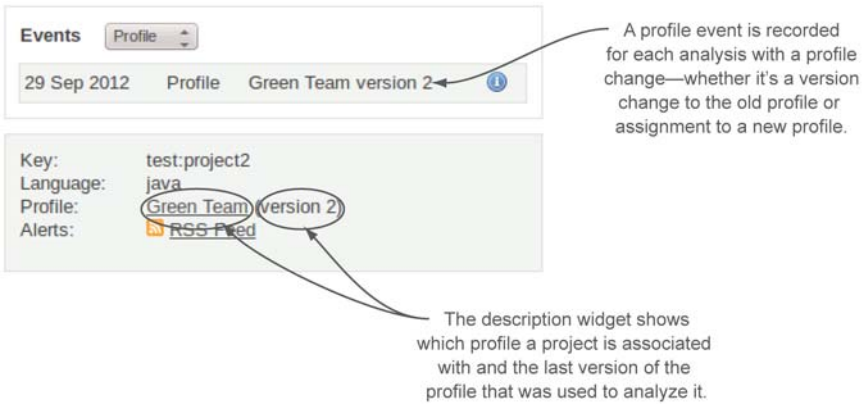


Figure 13.21 The description widget and the events widget both reflect profile version changes.

The Changelog tab is the only place in the profile editing interface that you see any reference to a profile's versions. But they're referenced in the projects that use the profile, as figure 13.21 shows.

Because everything in SonarQube revolves around the analysis, the description widget shows a project's assigned profile and the last version of the profile that was used to analyze the project, which isn't necessarily the most recent version of the profile. Profile version changes also appear in a project's events (an event is recorded for every analysis with a profile change) and history. Again, keep in mind that the history of an infrequently analyzed project may not reflect every single version of a profile.

13.5.3 Profile comparison

At the beginning of this chapter, we advised you to start your profile editing journey by making a copy of your favorite profile and editing it. At this point, your copy could be pretty far afield from where you started. Or not.

Fortunately, there is a way to tell. At the top of the list of quality profiles is an unobtrusive link: Compare Profiles. Click-through, and you'll find yourself at a simple interface showing two drop-down menus. Each one lists all available profiles. To find out how far you've strayed from the profile you copied, choose them in the drop-down menus and click the Compare button. Figure 13.22 shows the results for the Green Team.

With changelogs and profile comparisons under your belt, you know 95% of everything you need to fully and successfully administer your profiles. We'll cover the last 5% in the next section.

13.6 Administrative miscellany

At this point, we've discussed the meat of profile administration. But before we close the chapter, you need to know about a few more things: project assignment, backup and restore, and permalinks. Unfortunately, there's no good unifying theme among

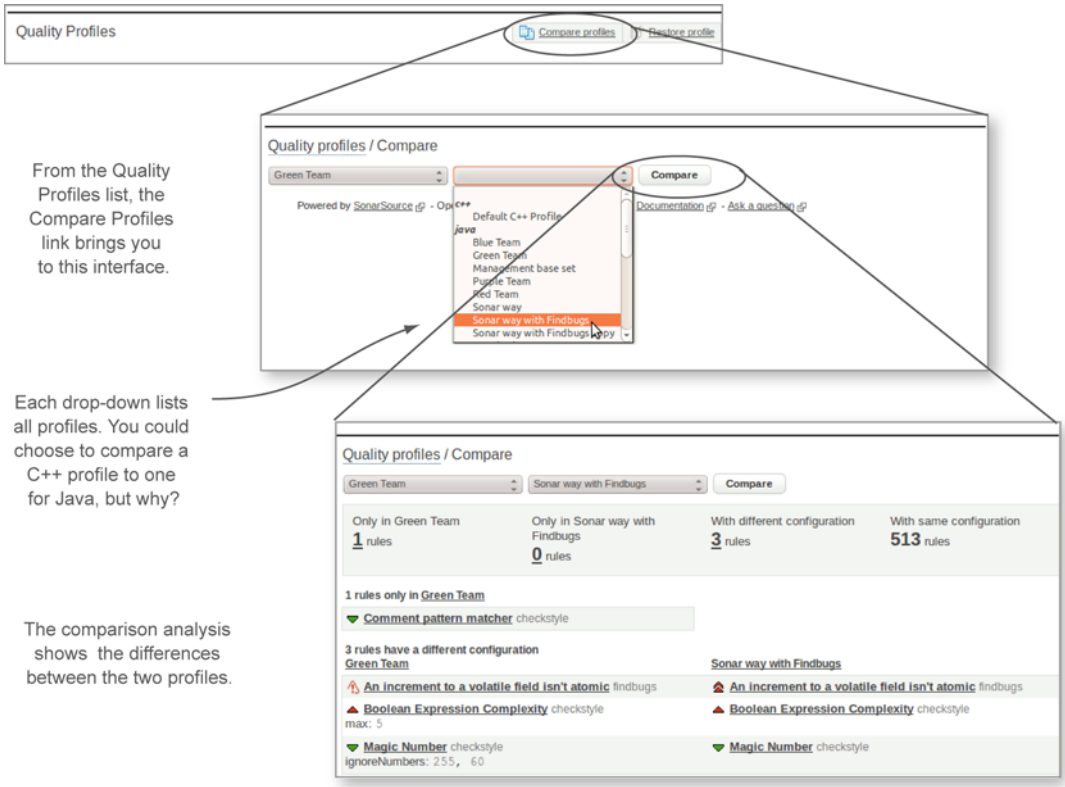


Figure 13.22 You access the profile comparison interface from the Compare Profiles link at the top of the list of profiles. It lets you compare any two profiles, regardless of language, and shows the differences in summary form at the top with a list of details at the bottom.

these topics, which is why we've punted and named this the miscellany section. We'll start with project assignment.

13.6.1 Project assignment: which project uses which profile

We covered the topic of the default profile and assigning a project to a profile in chapter 2, but it deserves at least a mention here. Every language SonarQube can analyze has a default profile: it's the one marked with a green check in the list of profiles. Changing the default is as easy as clicking the Set as Default button next to a different profile. When you do, the next analysis of each project that hasn't explicitly been assigned to a different profile will be run with the new default.

If you have projects that you don't want analyzed with the default profile, you can explicitly assign them to the profile you *do* want them analyzed with. To do that, click-through from the list of profiles on the target profile, and choose the Projects tab. Figure 13.23 shows the interface.

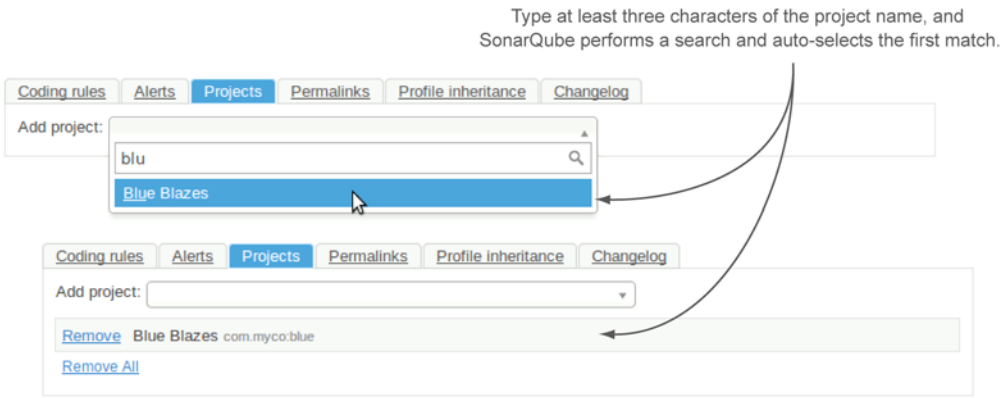


Figure 13.23 Use the interface on the Projects tab to assign projects to a specific profile for analysis. This works even on the uneditable profiles.

Once you've assigned a project to a profile, all its subsequent analyses take place with that profile—unless you override that choice by using the `sonar.profile` parameter at analysis time to specify a different profile, which isn't something you should do on a regular basis (if at all). One reason is that although a project's *assigned* profile appears on the project dashboard in the description widget, an analysis-time override of the assigned profile is *not* reflected there.

13.6.2 Profile backup and restoration

The mechanics of backing up a profile and then restoring it are fairly simple. The Quality Profiles page has a Backup button on each non-default profile. Click it, and SonarQube spits out an XML file. Save that to disk, and presto! You have a backup.

But a backup of what? Browse around in that file, and you'll find only two things: the profile's alerts and its rules. The backup includes your rule parameters, but it doesn't include any rule notes, the profile's changelog, or its assigned project list.

And although the backup includes every single rule in the profile, including the inherited ones, it doesn't include any indication of the inheritance relationship. So restoring that backup gives you back every rule, but you have to handle resetting inheritance manually.

But that raises the question of restoration. There's a Restore Profile link at upper right in the list of profiles. Click it, and a form pops up as shown in figure 13.24.

Restoring a backup is a simple matter of uploading it. Just be sure you're not trying to restore an existing profile.

13.6.3 Permalinks

A project's permalinks are a little like a backup. Each rule engine–specific link gives you a list of the profile's rules from that engine. Filed under the Permalinks tab in the profile administration interface, you'll find an All Rules link—which is another way to access the backup file—and a link for each rule engine available for the language.

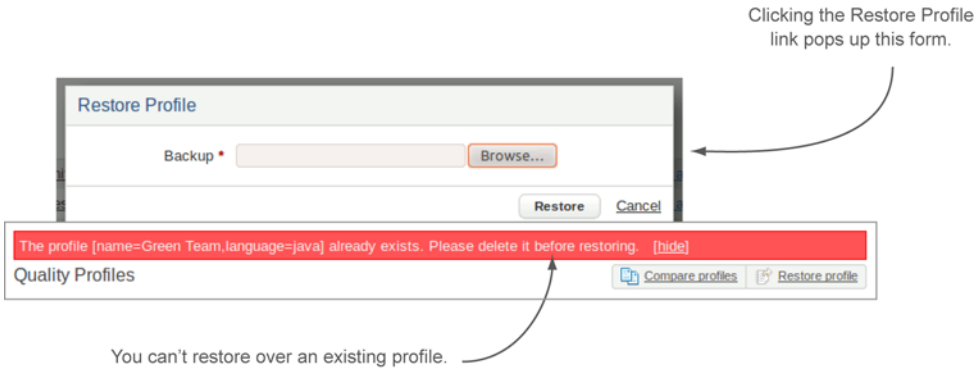


Figure 13.24 The Restore Profile link at the top of the Quality Profiles page adds a form that allows you to restore a backup. Just make sure you aren't trying to restore over an existing profile; you need to delete the current copy first.

At the beginning of this chapter, we looked at creating profiles from scratch. In that from-scratch interface, you have the opportunity to upload rule engine–specific rule files with which to seed the profile. At the time, we dodged the question of where those files would come from. Well, this is one source.

Like a profile backup, the engine-specific files contain your parameter settings. But because each of these files is generated in a format specific to the rule engine in question, don't look for your rule notes, because they're strictly a SonarQube-only thing.

With permalinks, backups, and project assignment under your belt, you now know everything you need to about how to administer SonarQube profiles.

13.7 Plugins

A number of plugins make new rules available, but most of them are better covered in other places. We'll cover only two plugins in this chapter: Switch Off Violations and our old favorite, Widget Lab.

13.7.1 Switch Off Violations

The Switch Off Violations plugin gives you the option to turn off rules in a fine-grained way. Why would that be useful? Let's say you have a profile that's being applied to both JEE and Swing applications. They're both flavors of Java development, but they're very different in focus and method. And the Swing developers are complaining about the "Unused formal parameter" rule.

They keep being dinged by it, but they say they can't help it because they didn't choose the method parameters; those are set by the Swing API. If they want their app to work, they have to use the method signature dictated by Swing, regardless of whether they need the parameter. They're lobbying to turn off the rule, but the JEE developers want to keep it on. You could split their rule sets, but if there are only a few

points of contention, it might be better to just turn off the use of that rule for the Swing trouble spots. That's what the Switch Off Violations plugin lets you do.

Using settings at a global or a project level, you can configure as many exclusions as you'd like. Each exclusion is one of the following:

- A pair of regular expressions to mark the beginning and ending of blocks that should be completely ignored by the rules. This is useful, for instance, when you're dealing with NetBeans' auto-generated code.
- A regular expression describing files or packages to be ignored.
- A combination of the file/package name regular expression, the key of the rule to be ignored, and a set of lines or line ranges, such as [4,23-50,72-81]. Leave the Lines input blank to have the exclusion cover all lines.

13.7.2 Widget Lab

The other plugin for this chapter is Widget Lab, which we created to fill what we saw as gaps in SonarQube's display of data. It's relevant here because it offers an augmented alerts plugin. In the earlier figures, you may have noticed that when there are multiple alerts, they're displayed together in paragraph form. Although that gets the point across, it doesn't grab you as hard as we thought it should. So we offer an alternative, shown side by side with the standard widget in figure 13.25.

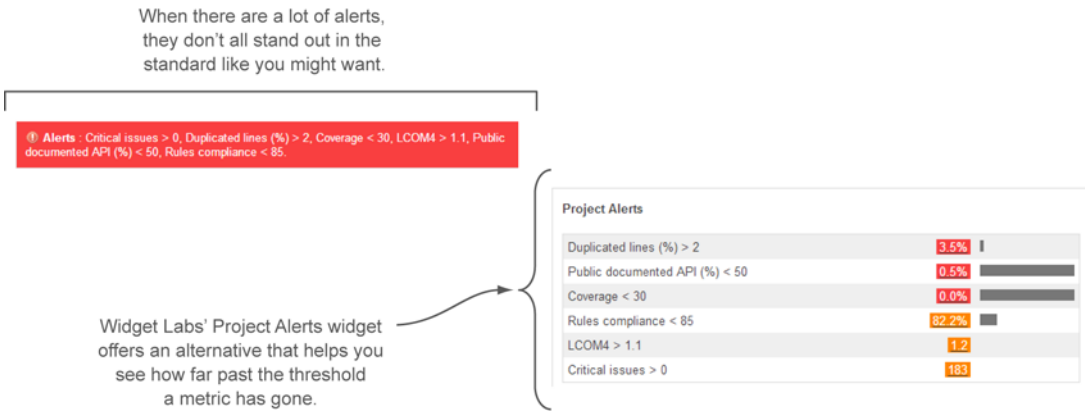


Figure 13.25 Whereas the standard alerts widget (left) presents all alerts in paragraph form, the project alerts widget (right) breaks out each alerting metric for a clearer picture of what's past its error threshold and what's at warning stage. The bar graphs to the right show you how far out of compliance the metric is.

13.8 Summary

This chapter has been a primer on profile management. You saw how to create a new profile (making a copy is usually much less work than starting from scratch) and how to set up profile inheritance. You learned how to include rules in and exclude them from your profiles and how to adjust their severities, and you learned to do lower-level edits of the rules themselves: their parameters, notes, and descriptions. We also showed you how to create alerts and how to pick good alert metrics.

You saw that you can track a profile's drift from the one you copied either by comparing the two profiles or by looking at your profile changelog. And you saw that profile changelogs and backups are rule-centric. Although a backup contains your alerts, the changelog doesn't; and neither reflects inheritance, rule notes, or rule-description extensions, because none of those things affect an analysis.

By learning to manage your profiles, you may feel you've mastered all the administration you need, but you're just scratching the surface. In the next chapter, you'll see how to configure filters and global dashboards to put the data that's most important for your organization front and center. You'll also learn how to set up notifications so that SonarQube notifies you (and your users) via email when there are new issues on one of your favorite projects.

14 Making SonarQube fit your needs

This chapter covers

- Working with filters
- SonarQube notifications
- The power of SonarQube dashboards
- Exploring the rest of global configuration

In most books that talk about software systems or tools, the parts that cover configuration or administration topics are considered boring and are usually skipped by readers. You're probably thinking the same about this chapter too, but don't pull the trigger yet.

SonarQube's default configurations are well thought out, but you can get much more out of them by adjusting some of the options we'll show you in this chapter. We aren't going to teach you only the "how"—instead, we'll focus on explaining the "why." You'll learn through real-world scenarios when and why you need to modify SonarQube configurations or use an administration feature.

We'll start by discussing filters. You might have already figured out that the default project list on SonarQube's home page is actually a filter. You can create as

many as you want, depending on your needs, and display their results in your dashboards. The first section of this chapter explains how to master SonarQube filters.

Then we'll look at global dashboards. They can show information on the highest level. You'll learn the purpose of default dashboards and how to share yours with other users (if you belong to the administrators group) or add theirs to your startup page. For normal users, dashboards are private, with no option to change that.

After that, we'll jump to SonarQube's embedded notification mechanism. You'll learn how you and your team can automatically receive important information, such as new issues or assigned issues. Next we'll cover custom metrics. These are metrics for which you manually enter the values because they can't be computed during a typical quality analysis.

Then we'll discuss some cases where you might want to modify the default SonarQube settings. For instance, how can you adjust SonarQube to appear in your language? Based on our experience, we'll present some best practices that we hope you'll find useful.

The rest of the chapter presents some low-level administration topics such as backup and restore, showing system info, and so on. Finally, although it may sound weird, we end with an overview of the related plugins. Before we start, make sure you're logged on in SonarQube as an administrator, because most of the stuff we'll show you is only available to registered users with admin rights.

14.1 Exploring filters

Tim's company started using SonarQube a couple of years ago. In the beginning, for evaluation and learning purposes, only a few Java projects were analyzed. Today, SonarQube hosts hundreds of projects developed in various programming languages.

Tim is the R&D line manager, and one of his key responsibilities is to track source code quality in collaboration with team leaders. Every day, he struggles to find, in SonarQube's first page, those projects for which the code-coverage metric is below 50%. He estimates that he spends around 15 minutes on searching activities. That's more than one hour per week!

In this section, we'll explain to Tim (and to you, of course) that SonarQube offers a flexible and powerful way to create filters based on a variety of attributes. Filters can be created by any logged-in user.

14.1.1 Adding a new filter

To create a new filter, click the Measures link, which can be found at upper left on your screen. When you do, the page shown in figure 14.1 opens. On the left are a link to manage your favorite filters and the search criteria form. The rest of the screen is currently blank. This is where you'll see the results when you perform your first search. Note that you can't use your new filter until you've added it to a dashboard, which we'll cover in the next section of this chapter.

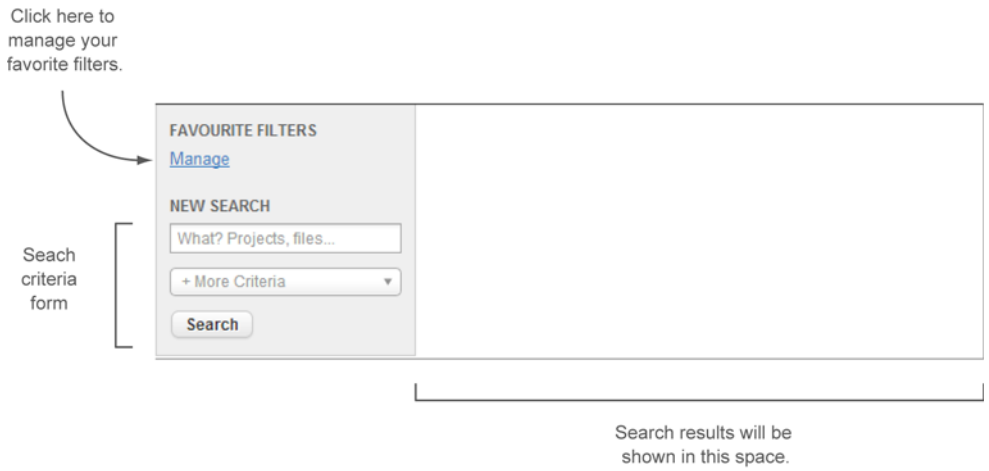


Figure 14.1 **Managing filters and performing a search**

Let's try to create your first search and save it as a filter. Assume that you want to search all projects with test coverage less than 50%. Click the input labeled What? Projects, Files in the search form to define what you want to search for, and select Projects. (From now on, we'll call this field the *resources input field*.) Then click the More Criteria button to add some more criteria. Select Metric from the criteria types drop-down list that appears. After that, click the Metric button, and pick Coverage (under the Tests category). Finally, fill in the rest of the fields so your screen looks like the left part of figure 14.2.

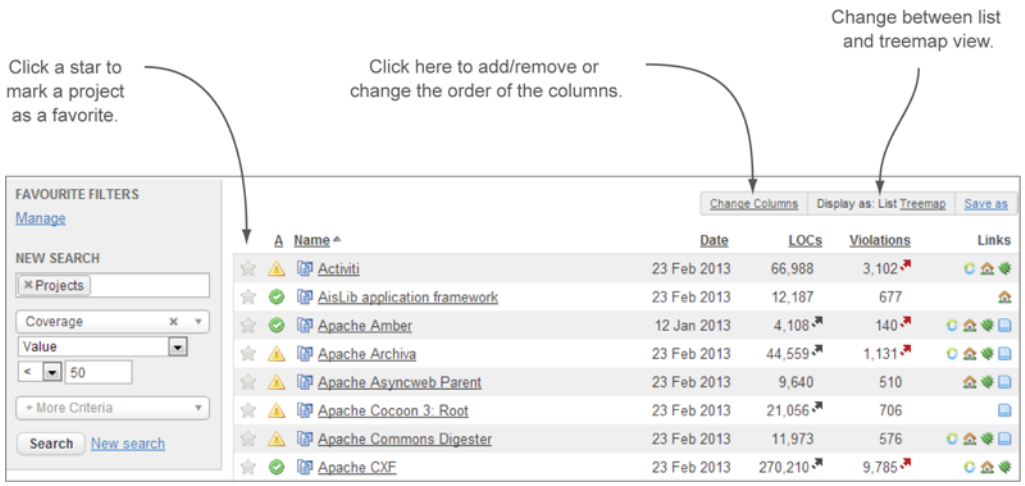


Figure 14.2 **SonarQube's flexible search form**

Click the Search button, and the results of your search appear as shown in figure 14.2. You now have several options, which we'll discuss in the next sections. For now, let's save the filter so you can use it in your dashboards.

Click the Save As link at upper right in the search results. In the pop-up window, pick a name that describes your filter, enter an optional description, and check the last option if you want to share your filter with other SonarQube users. When you click the Save button, you'll notice two changes on your screen: the Save As button is renamed Copy (clicking it opens the same window and makes a clone of the filter), and the filter you just created appears in the favorite filters section at left on your screen. From now on, whenever you browse the Measures page, you'll see this filter; if you click its name, SonarQube will display its results.

Before we explore more topics related to filtering, click the Manage link at the top of the search form. This link navigates to the filter administration page. As shown in figure 14.3, the upper part of this page displays the filters you've created, and the bottom lists filters created and shared by others. You're probably wondering where these shared filters came from. These are predefined filters included in the SonarQube installation. Only administrators can manage them, but by default they're available to all users.

14.1.2 Customizing the filter view

Now that we've covered the basics of filter administration, let's discuss how you can customize the look and feel of the search results. As shown in figure 14.2, two links appear at upper right on the page:

- *Change columns/Change Treemap*—Allows you to add/remove or change the order of the columns or change the values of the treemap view.
- *Display as List/Treemap*—Allows you to change the view of the current filter.

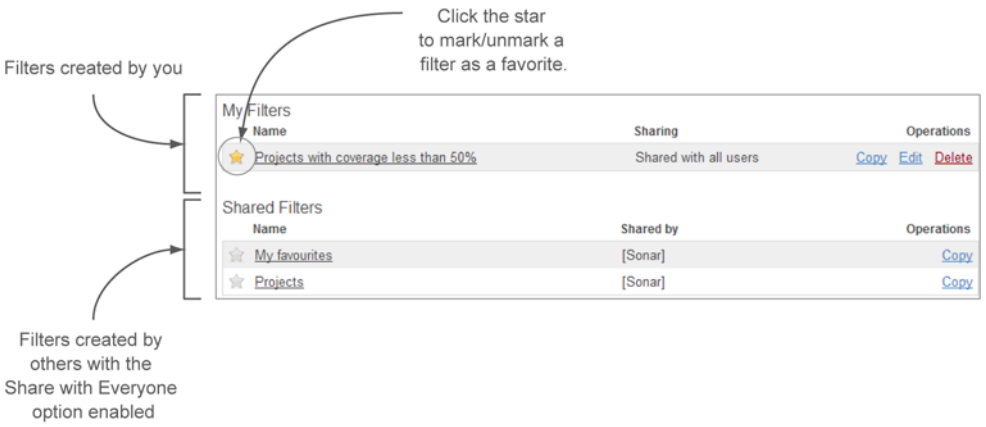


Figure 14.3 **Filter administration page**

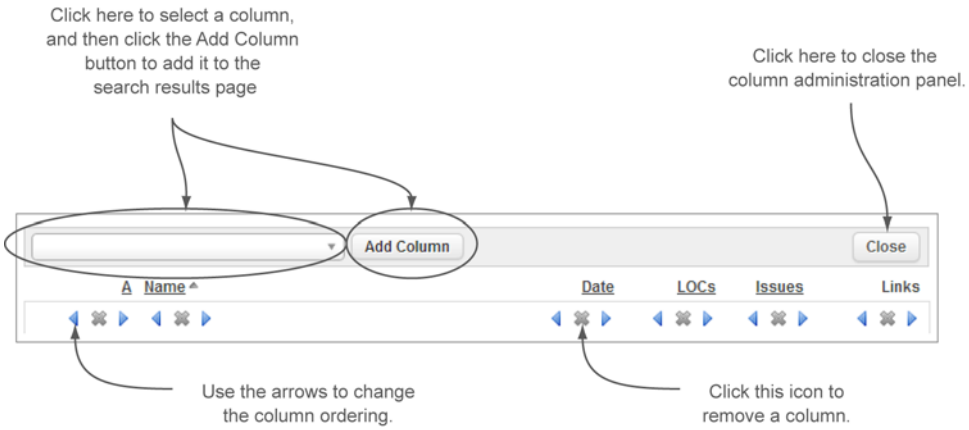


Figure 14.4 Change Columns panel at the top of the search results

When you're in List mode and you click the Change Columns button, a new panel is displayed at the top of the results, as shown in figure 14.4. You can add a new column by clicking the drop-down list box, remove an existing one, or change the order by using the arrows located over each column. Finally, you can sort the results by clicking a column's name.

Let's change the view mode to Treemap, which provides an alternative view of the filter results (see figure 14.5). Clicking the Change Treemap link displays a configuration panel similar to the one we showed you for the list view. There are only two attributes to set when using this display type: the selected Size metric is used to tell SonarQube how the project rectangles will be computed, and the selected Color metric is responsible for colorizing the rectangles.

Before we move on to more advanced cases of filter creation, as an exercise, try to remove the Date column from the filter you just created, and add a column about project complexity. Also move the Issues column so it's the first column, and then sort the results by the Lines of Code metric. Finally, add criteria so the filter displays projects with code coverage less than 50% and code duplication more than 1%.

14.1.3 Advanced filtering

Looking again at figure 14.2, by clicking the More Criteria button you can add criteria to your search. Notice that every possible SonarQube entity can be included in your search, which makes SonarQube filtering a powerful and flexible mechanism. Let's explore these options with some practical examples.

DIFFERENTIAL FILTERS

Imagine that Tim (the R&D line manager) wants to get a list of all the resources for which the code coverage got worse since the last analysis. He's in luck, because filters can take advantage of SonarQube's differential service. You might want to refresh your memory by going back for a minute to chapter 9, where we covered this topic in detail.

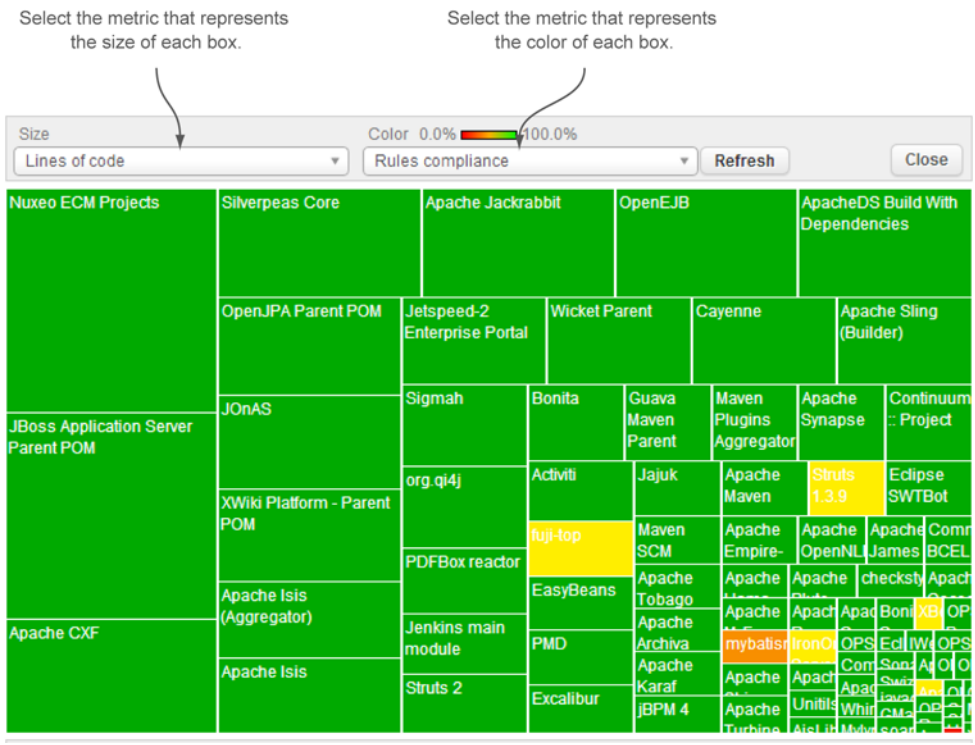


Figure 14.5 **Previewing and customizing how a filter looks as a treemap**

We'll guide you through the required steps to create such a filter in a couple of minutes. First, click the New Search button to reset the search form and the results. Now, in the first input, select all the available resources: Projects, Sub-projects, Directories, and Files. (We omitted the unit test files because SonarQube doesn't compute code coverage for this type of resource.) Then enter the search criteria: select the Coverage metric; and instead of Value, change the drop-down list to the differential period Since Previous Analysis. Choose your assertion criteria (<= in this example), and enter the value `0` in the last input field. Your screen should look like the one in figure 14.6.

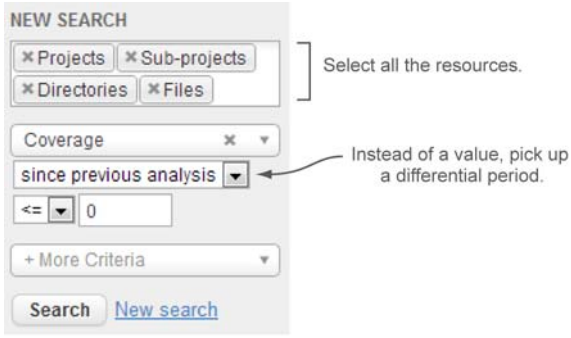


Figure 14.6 **Creating a differential filter**

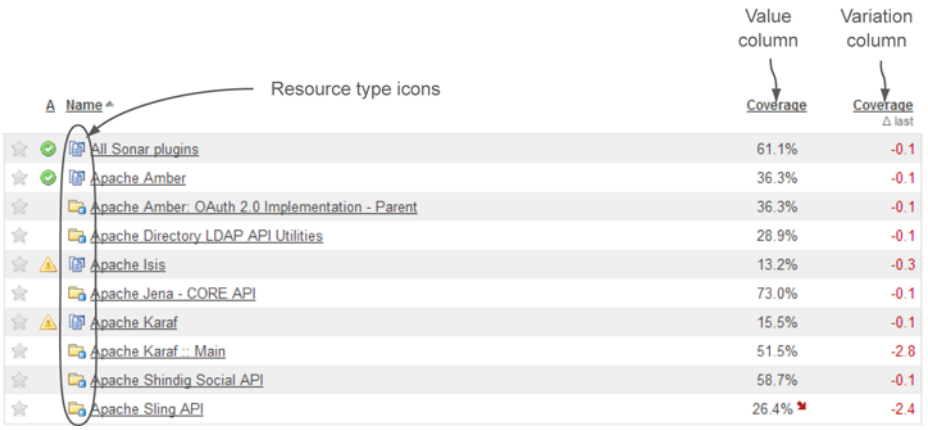


Figure 14.7 Results preview of a differential filter

The filter criteria you just entered might look weird. A change of zero (0) would mean no change at all, whereas a change of greater than zero would mean a change. What does this negative change criteria mean? Well, when you're dealing with differential periods, checking for negative change makes absolute sense: it means the metric got worse since the selected differential period.

Click the Search button to get a sneak peek at the results of your filter. Figure 14.7 displays a preview of the results. Notice that there are only two columns. We did that in the screenshot to call your attention to the difference between the value and differential period columns.

In the value column, in addition to the metric value, you also see the trending icon. Furthermore, we added a variation column for the coverage metric (there is a note under the column header to distinguish them) to show only the difference from the selected analysis for comparison. Note also that placing a variation column in a non-differential view is useless, because it doesn't show anything (it's blank). To familiarize yourself with columns and differential filters, try adding some more metrics as values or variations, or select another differential period to see how the filter results change.

Before the resource's name is an icon indicating its type. Table 14.1 summarizes the different icons and their meanings.

Table 14.1 Resource types and icon indicators

Icon Indicator	Resource type
	Project
	Subproject/module
	Package
	Directory
	File/Class

OTHER CASES

Filtering has more flexibility to show us. Let's assume that in your SonarQube installation, you host projects written in multiple programming languages. You'd like to examine the quality of all web development files containing the word *print*, excluding those developed in Java.

Due to the fact that there are plenty of legacy projects that have had no code modifications for a long time, you want to get results only for analyses triggered in the last three months.

> **NOTE** This example supposes that additional language plugins such as XML, JavaScript, and WEB have been installed in SonarQube. Chapter 1 provides an overview of the supported languages, and later in this chapter we'll discuss the update center, which is the central place to manage your plugins.

For all the requirements we just described, SonarQube filtering has something to tell you. Create a new search, and follow these steps:

1. Select only Files in the Resources input. Select the required languages by clicking the More Criteria button and selecting the Language option. Click again on the input that appears, and select all the required languages. For instance, if you deal with J2EE web development, you might need to check XML, JavaScript, and WEB. Don't select Java, because you don't want Java files in your results.
2. Add a new Name criteria, and enter the following value: `*print*`. The character * can be used as a wildcard to match more than one resource.
3. Add a new Age criteria, and enter the number `90` in the Inspected field to tell your search to include only files built during the last three months (90 days).
4. Click Search.

Before we move on to the next topic, there are a couple of criteria types we haven't discussed yet. In the last example, instead of using the resource name, you could have used its key. But be aware that for files, the key is composed of the project/module key plus the full package name.

For instance, let's say a project has the following key: `org.codehaus.sonar-plugins:sonar-widget-lab-plugin`. The key for the file AlertsWidget contained in the package `org.codehaus.sonar.plugins.widgetlab` is `org.codehaus.sonar-plugins:sonar-widget-lab-plugin.org.codehaus.sonar.plugins.widgetlab.AlertsWidget`.

There is one last thing we'd like to explain: the Components criteria type. In general, you can use it to limit the results of a filter for a given project or subproject/module. By selecting this type, you can pick a project or a component/module. To do so, type the name or part of the name you're looking for, and SonarQube will display a list of all projects and subprojects matching your input. Click Search to see the result of your filter and save it to your favorites.

14.1.4 SonarQube's default filters

Now that you're familiar with searching and creating filters, let's go back to the filter administration screen. To do so, click the Manage link. The filter you just added is shown in the My Filters list, and SonarQube's default filters appear below.

As we mentioned earlier, SonarQube comes with two ready-to-use-filters, which we'll discuss in this section. The bad news is that they're generic and, in large SonarQube installations, almost useless. The good news is that you copy them and modify their criteria.

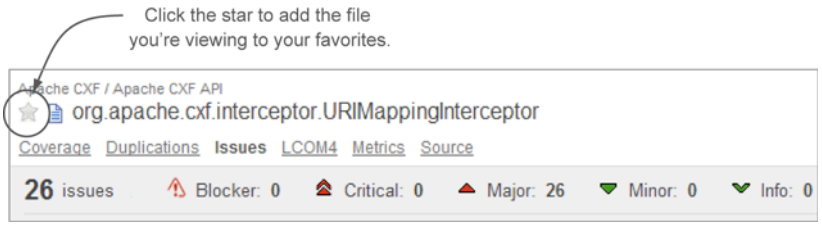


Figure 14.8 Adding a file to your favorites

MY FAVORITES

As you've probably figured out, this filter lists all your favorite resources: projects, components, or files. You can easily flag (or unflag) a resource as a favorite by clicking the star icon that appears on many SonarQube screens. For instance, in the source code tab viewer, the star icon is located just before the name of the file, as shown in figure 14.8. Similarly, when you're viewing a project's dashboard, you can find the Favorite star at upper right on the screen.

You can even flag/unflag resources when viewing a filter without needing to drill in to a project dashboard or browse resources. The Favorite star for each resource appears in the far-left column. Every resource that is flagged as a favorite is automatically displayed in the Favorites filter.

PROJECTS

In small SonarQube installations, or if you're making your first baby steps with SonarQube, this filter is useful because it shows all projects hosted in SonarQube. But as more and more projects are analyzed, such filters become obsolete or hard to use. Thus, we suggest you create filters that make sense to you and/or your team.

By now we hope that you've mastered SonarQube filters and can define search criteria based on your needs. The next section introduces dashboards and, among other things, teaches you how to use your mighty filters.

14.2 One size doesn't fit all: managing global dashboards

As you've already figured out, SonarQube computes hundreds of metrics associated with different aspects of quality. Its usefulness isn't limited to the developers' world; as we've mentioned many times throughout the book, testers, architects, team leaders, line or project managers, and even upper management can take advantage of what SonarQube offers.

The problem is that a line manager, for instance, needs completely different information to track and evaluate the quality of a project than a developer. A line manager is presumably satisfied with an overview of all active projects without too many details, whereas team members surely want to have access to source code coverage reports or complexity metrics for the projects they're currently involved in.

We've worked in all possible technical roles in a software project, and we've learned that each person needs a different point of view for the same data. SonarQube comes

to the rescue and provides a powerful and easy-to-use mechanism to manage your dashboards at a global or project level.

> **NOTE** If you're a SonarQube administrator or a SonarQube evangelist who introduces it to your team, then we advise you to empower users to create their own dashboards that show exactly what they want to see. Of course, SonarQube administrators should set up some generally useful dashboards and filters, but we're pretty sure those will never manage to make everyone happy. Instead of trying to fit all the different needs into a few global dashboards, focus on teaching users how to customize dashboards of their own.

This section discusses global dashboards, but most of the content also applies to project dashboards, which we'll cover in the next chapter. You'll learn how to create and customize the look of global dashboards by adding widgets or adjusting the way they look. We'll also show you the purpose of default global dashboards and how you, as an administrator, can manage them.

14.2.1 Creating your first global dashboard

To start managing your global dashboards, click the Home link in the upper-left corner of SonarQube's first page. Then click the Manage Dashboards link in the upper-right corner (see figure 14.9).

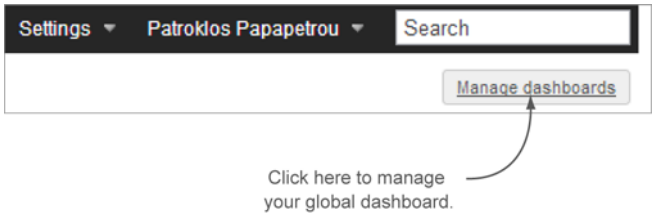


Figure 14.9 The Manage Dashboards link is available on your home page.

If you don't see the latter, ensure that you've already logged in, because global and project dashboard configuration isn't available to anonymous users. Figure 14.10 shows what you should see once you've clicked the Manage Dashboards link.

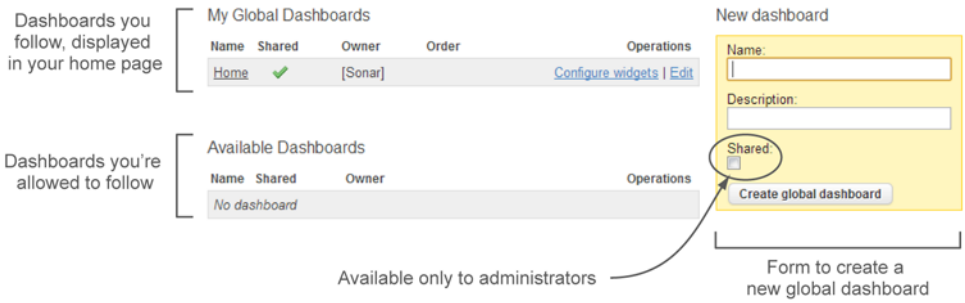


Figure 14.10 Global dashboard configuration page

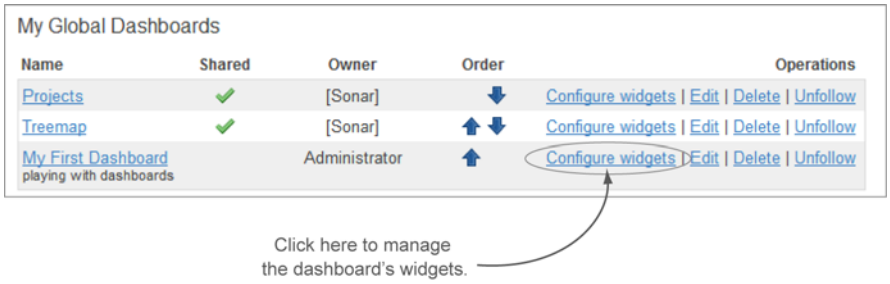


Figure 14.11 Every new global dashboard is added to the My Global Dashboards list.

Similar to filters, SonarQube comes with default dashboards that we'll explore later in this section. The Available Dashboards list includes all shared dashboards (either by administrators or by SonarQube). You can follow/unfollow any dashboard by clicking the relevant links. For clarification purposes, *following* a dashboard means the dashboard appears as a link in the left menu of your homepage.

Now that you're familiar with the basic concepts of global dashboards, let's create a new one. Enter a name and description in the New Dashboard panel, and click the Create Global Dashboard button. Keep in mind that only users with administrator privileges can share a dashboard, so if you can't find the Shared check box, you're probably logged in as a normal user.

The newly created dashboard now appears in the My Global Dashboards list, as shown in figure 14.11. If you want to edit its details (name, description, or sharing), click the Edit link. To see how it looks, although it's currently empty, click its name. If you want to completely remove it from SonarQube, click the Delete link.

14.2.2 Customizing your dashboards

To add some widgets and customize the dashboard's look, click the Configure Widgets link. As shown in figure 14.12, from the global dashboard customization page, you can do the following things:

- Change the layout
- Add/remove global or project widgets
- Preview your dashboard

SonarQube dashboards support several common page layouts. You can choose from among the following: single column, two columns, and three columns. The two-column layout comes in three flavors:

- *50%-50%*—Both columns are equally sized, and each gets half of the available page width.
- *30%-70%*—The left column gets 30% of the available page width, and the right column gets the rest (70%).
- *70%-30%*—The Left column gets 70% of the available page width, and the right column gets the rest (30%).

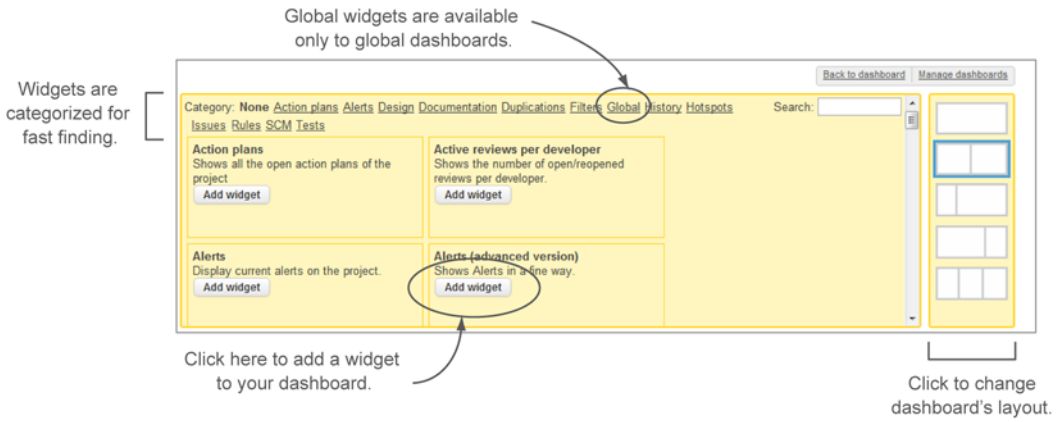


Figure 14.12 **Global dashboard customization page**

For this section's example, select the last option (70%-30%).

Adding a widget to your dashboard is easy. Search for the widget you wish to add by clicking the categories shown at the top of the screen. Once you find it, click the Add Widget button, and your widget is added in the upper-left position on your dashboard. But we need to clarify a couple of things.

You haven't met the Global widget category until now. This category contains widgets that can be placed *only* in global dashboards. Typical (project) widgets are available for both kinds of dashboards. Only three global widgets ship with SonarQube: Measure Filter as List, Measure Filter as Treemap, and Welcome, as shown in figure 14.13. Other plugins, such as the Widget Lab plugin, may offer additional widgets for your global dashboards.

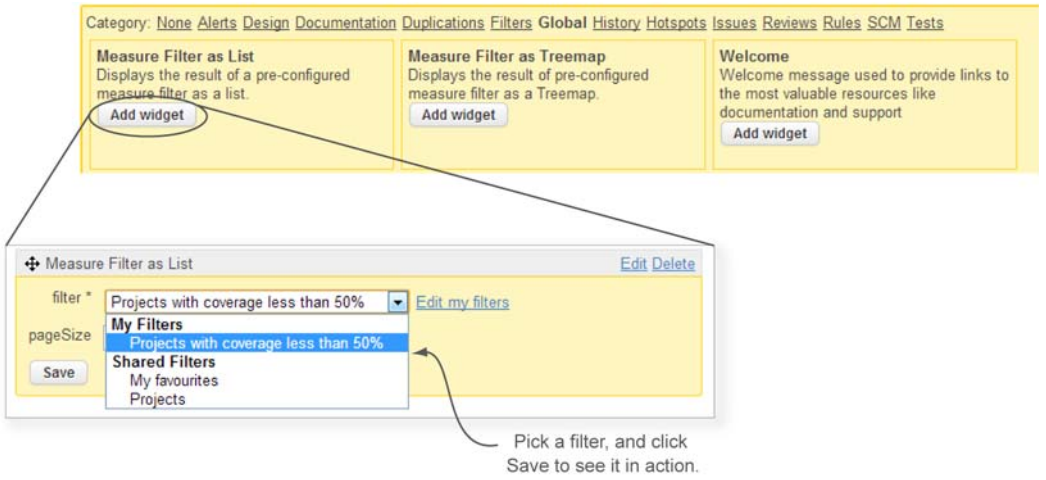


Figure 14.13 **Adding the global widget Measure Filter as List to a global dashboard**

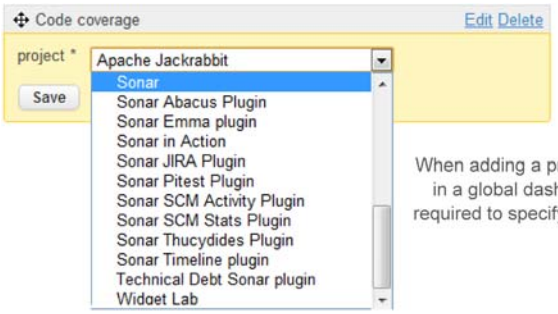


Figure 14.14 Adding a project widget to a global dashboard requires that you specify the project from which SonarQube will fetch widget data.

Once you've added a global widget to your dashboard, you need to select a filter from the drop-down list. You can pick either a filter you've created or a shared filter you follow. Click Save, and you'll see your dashboard in action. There is no limit on the number of filter widgets allowed in the same dashboard.

Now let's add a project widget in the same dashboard: code coverage, for instance, or any other one you're familiar with. As shown in figure 14.14, as soon as you add the widget, SonarQube asks you to specify the project for which you want to fetch data (code coverage in this case).

Click the Save button again, and the selected project's code-coverage metrics are shown in the widget. Notice that widgets in global dashboards have a descriptive title to remind you of the project or the filter you're viewing.

To remove a widget from the dashboard, click the Delete link in the upper-right corner of the widget. Finally, if you're dealing with a configurable widget (such as the filter widget we're looking at), you can edit its settings by clicking the Edit link, which is to the left of the Delete link.

Because new widgets are added to the left column by default, your dashboard probably isn't attractive right now, with two widgets on the left side and none on the right. To move a widget between dashboard columns, grab the widget's header to drag and drop it into a different column or into a different position in its current column (see figure 14.15).

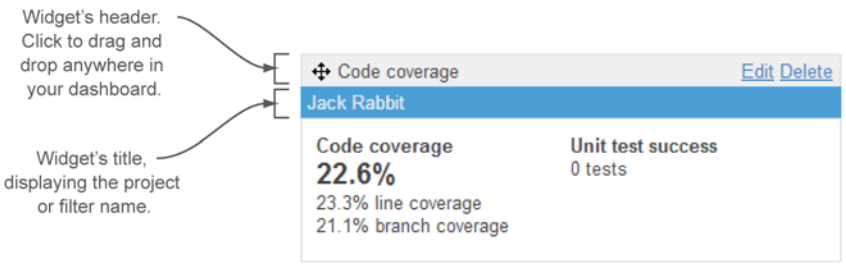


Figure 14.15 Project widgets in global dashboards have a header you can drag and drop between columns, and a title displaying the project or filter name.

When you're done rearranging your widgets, click the Back to Dashboard button, which is located in the upper-right corner of this page, and you'll go to your home page. In the left menu you'll see a new link that points to the dashboard you just created. If you want to make more modifications, you can always click the Configure Widgets button.

Figure 14.16 shows how the dashboard will look. Notice that in the right (30%) column, we've added the same project widget (coverage) for several projects.

14.2.3 Defining default global dashboards

In the beginning of this section, we scratched the surface of the topic of default global dashboards. Now it's time to expand on that topic. As you saw in figure 14.10, SonarQube ships with one preconfigured global dashboard named Home, which includes all the available global widgets. Recall from the section on SonarQube's default filters that, by default, you can use the following shared filters:

- Projects (displays all projects analyzed by SonarQube)
- My Favorites (displays all resources flagged as favorites)

Accordingly, on your home page you can add any global dashboard by clicking the Follow link in the dashboard administration page. Every user automatically follows the global dashboard that SonarQube ships with.

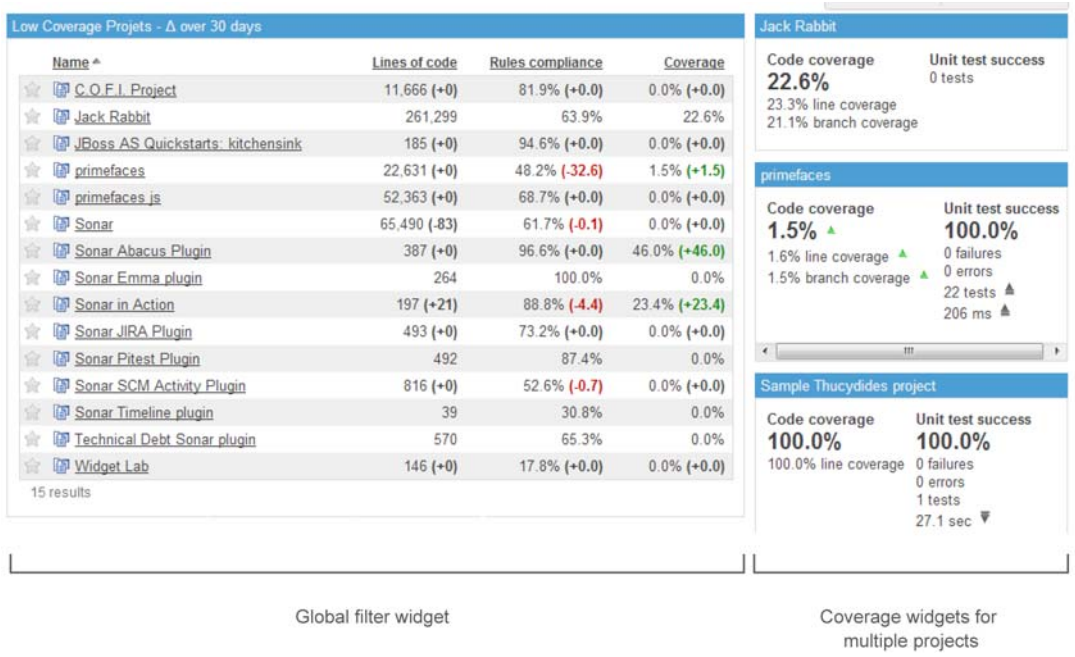


Figure 14.16 A complete global dashboard. The left column contains one global filter widget, and the right column contains three instances of the same project-coverage widget showing information for different projects.

The dashboards that are displayed by default for all users are called *global default dashboards*. Their usefulness isn't restricted to logged-in users; they're available for anonymous access as well. Otherwise, an anonymous user trying to access SonarQube's home page would probably see a blank page, which isn't convenient.

The beauty of default dashboards is that users with administrator privileges can add any global dashboards to or remove any global dashboards from them. Let's see how.

> **NOTE** Global dashboards are likely to contain one or more filter global widgets. You might be wondering what happens if a user follows a default global dashboard that contains a non-shared filter. You can try it at home and see how SonarQube handles it, but you need to take several steps to do so. The answer is that SonarQube displays the results of the filter even if the filter isn't shared.

First navigate to the global configuration page and click the Default Dashboards link from the left menu. As shown in figure 14.17, this administration page is split in three panels.

On the top are the default *global* dashboards. All of them are shown by default on SonarQube's home page. Remember that, as you learned in section 14.2.1, each user is free to choose which default dashboards to follow. Next, there's a list of default *project* dashboards. As you probably figured out, these dashboards appear on a project's home page. We're going to cover them in depth in chapter 15. The last panel lists all available (flagged as shared by administrators or SonarQube) dashboards that can be added to global or project default dashboards.

To add dashboards to or remove them from the default lists, click the link shown in the Operations column. You don't have to worry which list (global or project) to

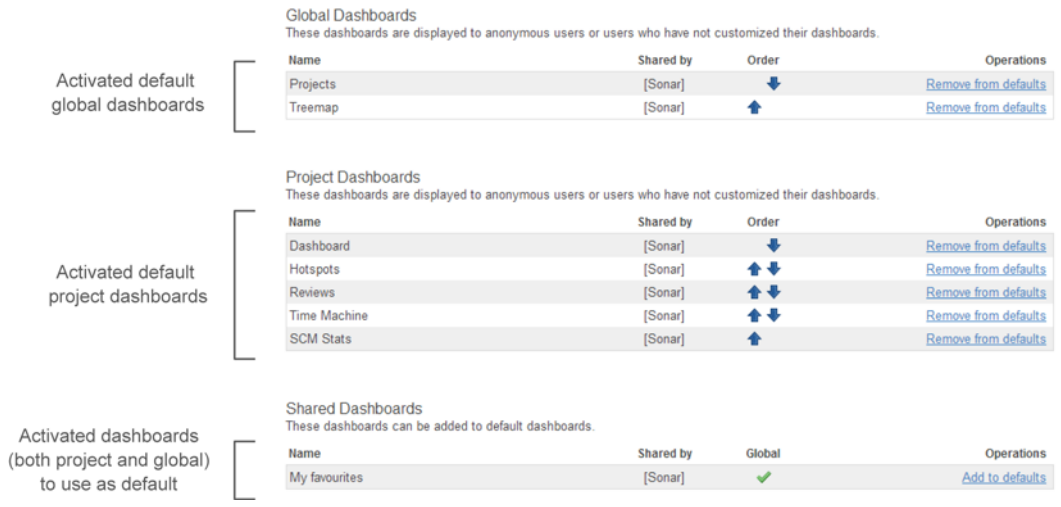


Figure 14.17 Default dashboard administration

move it into, because SonarQube automatically picks the right place for you, depending on the dashboard's type. You can change the order in which they appear by clicking the arrows shown in the Order column. Finally, the Shared By column indicates who has created (and shared) each dashboard.

For practice, play around with the default global dashboards. Try to remove all of them from the first list, and notice how SonarQube behaves.

So far, we've covered the two most important global configuration topics for SonarQube: filters and global dashboards. In the rest of the chapter, we'll explore some valuable features such as notifications and global settings.

14.3 Getting notified by SonarQube

In the world of automation, collaboration, and Continuous Inspection, processes and tools should help you keep your team in good shape. Interruptions or losing time in non-development activities should be eliminated as much as possible.

Assume that you've recently adopted the code review practice. Ivan, your team leader, has assigned an issue to you. You need to refactor a class to remove the complexity issues raised during the latest SonarQube analysis. It's a task that you'll need around two days to complete.

Unfortunately, you got sick and were absent for a few days. When you returned, you immediately started working on your task. Meanwhile, Ivan had assigned the same issue to Helen, because it was a top priority for the current iteration. At the end of the day, in a stand-up discussion, you realize that you were both working—*the whole day*—on the same task. Lack of communication, you probably think. Well, we'd say lack of notification!

14.3.1 Activating the notification mechanism

SonarQube comes with a notification mechanism that was making its first baby steps when this book was published. To activate it, you need to set up your email server settings. To do so, navigate to the global configuration page, click General Settings in the left menu's System section, and select the Email category.

As shown in figure 14.18, all attributes are straightforward, and SonarQube provides adequate information. When you're done, you can test if everything works by sending a test email using the Test Configuration section.

> **NOTE** Before we move on, double-check that you've entered a valid email address in your user profile. Otherwise you won't be able to receive email notifications for the events you'll subscribe to in the following sections. If you don't remember how, go back and take a look at chapter 12.

Now that you've activated the notification mechanism, let's see what kind of emails SonarQube can send you.

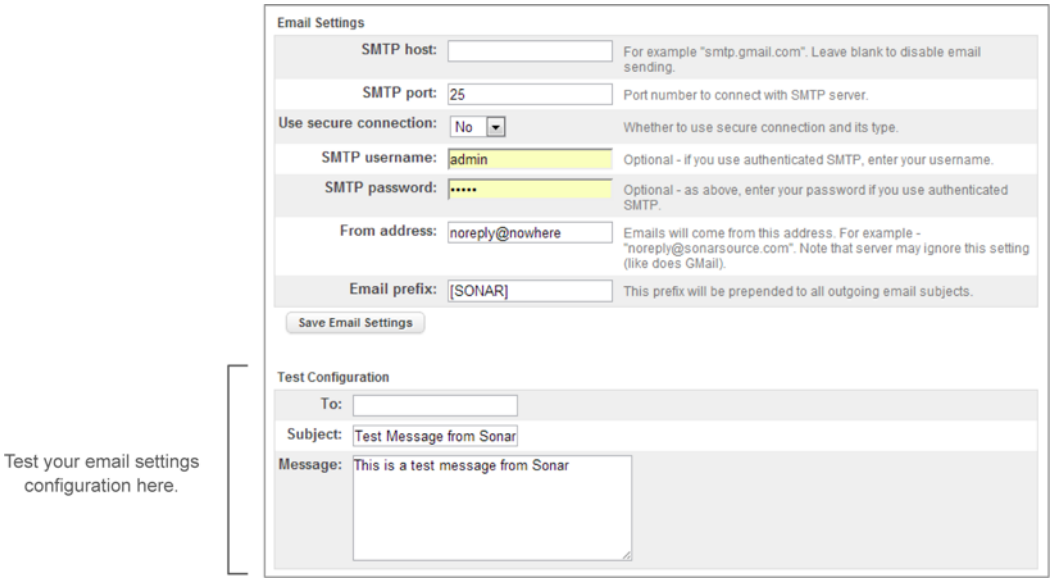


Figure 14.18 **Email settings: activation of SonarQube notifications**

14.3.2 Subscribing to event types

SonarQube can notify registered users about four kinds of events per project or globally:

- Changes in an issue assigned to you or created by you
- New alerts created by the assigned quality profiles
- New false positives
- New issues on your favorite projects introduced during the first differential view period

You've learned that, by default, the first differential period is Since Last Analysis. All notification events except the first one are triggered as a post-analysis step, which means you'll receive emails for new issues, alerts, and false positives on projects introduced during the last analysis. Of course, you can change the first differential period in the global configuration to something other than Since Last Analysis, in which case you'll be notified about issues, alerts, and false positives introduced since the period you've set.

By default, all notification types are disabled, so each individual user needs to tell SonarQube that they want to receive email messages. Navigate to your profile either by clicking your name in the upper-right corner or by clicking Configuration and then My Profile in the left menu.

Figure 14.19 shows the available options. Check the boxes to subscribe to the event types and start receiving emails for all projects. If you enable the first notification, SonarQube will send you a message when a change (comment, resolution, new assign-

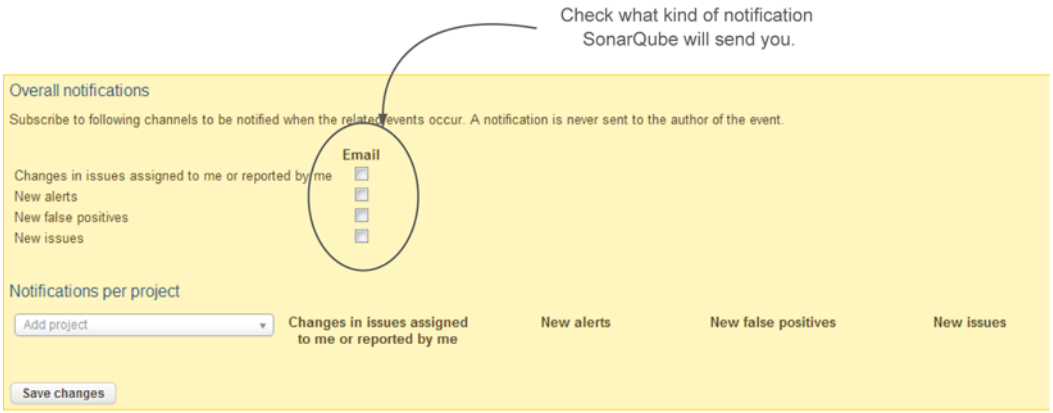


Figure 14.19 **Event subscription**

ment, and so on) occurs in a review assigned to you or created by you. The rest of the events are triggered for all projects. You receive an email when new issues, alerts, and false positives are introduced during the first differential period—that is, period 1 in the global differential service settings, as you saw in chapter 10.

> **TIP** SonarQube includes a URL for accessing the issue or project issues in all email messages. Make sure you've changed the server's base URL in the global settings, as we'll show you in the next section, so the URLs make sense to you and you can access SonarQube in your intranet or from the internet.

If you host several projects in your SonarQube installation, after a while you'll find that receiving emails for all of them isn't very productive or useful. The good news is that you can tell SonarQube the projects for which you want to receive these notifications. Click the Add Project list in the Notifications per Project section. Then, type the first three letters and select your favorite project. Finally, choose which notification types you want, and you're finished. Repeat the same steps for every project you're interested in.

In the last three sections, we've discussed the most remarkable topics related to global administration in SonarQube. You can use SonarQube without knowing about them, but we believe that by mastering these features, you'll boost the value of your SonarQube installation and, in some cases, the productivity of your team.

The rest of the chapter focuses on more specific configuration topics and ends by explaining the usability of the update center, as well as by describing a related plugin.

14.4 Adjusting global settings

We're pretty sure you're already familiar with the term *global settings*, because we've shown you many times how to edit global attributes for SonarQube core or other plugins. In this section we'll add some glue to what you've seen so far in the book and explain the remaining categories.

SonarQube ships with many system plugins and features with editable global properties. A newly installed plugin may also provide attributes that can be configured globally.

To see this default global configuration and edit it for your environment, start from the global configuration page and click the General Settings link in the left rail. On the left side of the content area is a list of available configuration categories. Looking at them, you'll realize that we've already covered most of them.

Configuring Checkstyle and FindBugs was introduced in chapter 2. Cobertura, code coverage, and JaCoCo were discussed in chapter 3, and duplications were explained in chapter 4. Email settings are fresh because you saw them earlier in section 14.3.1. Chapter 12 covered security settings; and finally, in chapter 9, we dealt with the differential view attributes and how you can use them to assist you in the Continuous Inspection process. Let's see how and when the rest of the categories may be useful.

14.4.1 Database cleaner

In large installations with hundreds of projects and many analyses per day, it's pointless to keep all snapshots stored in the database. SonarQube is enriched with a powerful database-cleanup mechanism that decides how many snapshots it will keep per day, week, or month, and how historical data of packages/directories should be handled.

We'll cover in detail the concept of a project snapshot in the next chapter, but for now bear in mind that a snapshot is an image of your project quality at a specific time. In that sense, keeping multiple snapshots for the same day won't make any difference when you're looking back a few weeks later, so the database cleaner is responsible for removing these obsolete snapshots from the database.

In the global settings, you can specify the attributes presented in table 14.2. (SonarQube's inline help is awesome in this category, so we don't need to explain further.)

The default values shouldn't cause you any problem because they've been carefully selected by the SonarQube team, but you can always adjust them if you feel that you need to increase or decrease the time before the snapshots will be removed.

Table 14.2 Database-cleaning configuration attributes

Property description	Default value
Number of hours before SonarQube starts keeping only one snapshot per day	24 (one day)
Number of weeks before SonarQube starts keeping only one snapshot per week	4
Number of weeks before SonarQube starts keeping only one snapshot per month	52 (one year)
Number of weeks before SonarQube starts removing all snapshots of a project	260 (five years)
Enable/disable the cleaning of historical data for directories/packages	True
Number of days before deleting closed issues	30

Just keep in mind that these values don't represent a schedule of when some sort of sweeper runs through the database, deleting snapshots. Snapshot cleanup is performed at the end of each analysis, and these values are the age thresholds beyond which any given snapshot (that's not marked with an event) is eligible to be deleted.

So if you've got a lot of hoary old snapshots from idle projects clogging your database, adjusting these settings won't free up any space. No analysis, no cleanup. You need to run a fresh analysis of each of those projects to have their snapshots cleaned out.

14.4.2 General

This category includes attributes that can't be grouped elsewhere. You can find some things to modify:

- If you're not in a Java house, you can change the default language of the source code to analyze.
- The next two attributes list the plugins that are accepted and excluded when running an analysis in DryRun mode. DryRun mode lets you get all data required to do a project analysis through a web service and dump the result of the analysis in a local file. So, DryRun mode is database-less. The first use case that takes advantage of this feature is Sonar Eclipse, but in the future it might be used in other cases such as pre-commit analysis to reject a file based on some criteria.
- Rules weight are related to issues and allow you to change the weight of each issue severity.
- The server base URL is the URL root SonarQube uses when it talks about itself.

As you saw in section 14.3.2, when SonarQube sends you an email notification, the message body includes a URL pointing to the issue page.

If you activate the notification mechanism, we strongly advise you to change the server base URL to something accessible at least within your intranet, because the default value (http://localhost:9000) doesn't make any sense unless you're reading the email messages on the same machine that hosts SonarQube.

Rules weight feeds into the formula SonarQube uses to calculate the Weighted Issues (WI) and Rules Compliance Index (RCI) metrics. Take a look at figure 14.20 to refresh your memory. If you edit the WI value, you'll have to run a new analysis for all projects in order to use the new weights.

$$100 - \left(\frac{(\text{Blockers} * 10) + (\text{Criticals} * 5) + (\text{Majors} * 3) + \text{Minors}}{\text{Lines of Code}} \right) * 100$$

Figure 14.20 The Rules Compliance Index is the Weighted Issues score divided by the number of lines of code in the project, turned into a percentage and subtracted from 100.

14.4.3 Localization

As we've already told you, localization of SonarQube messages relies on browser configuration. But if you want to display the rule engine messages in your language as well (if it's supported by the localization plugin), you need to change the value of the Localization attribute.

By default, it's set to `en`, which means all messages are displayed in English. Set it to the language you prefer, install the relevant localization pack, and restart SonarQube, and if the rule engine's message are translated, you'll see them in the language you've chosen.

14.4.4 Server ID

When you purchase a commercial plugin from SonarSource (the company behind SonarQube), you're asked for your server ID. To generate one, navigate to the Server ID category on the global settings page. You need to enter the name of your organization and the IP of the machine that hosts SonarQube.

If, for some reason, you change the IP address or move SonarQube to a new machine with a different IP address, you'll have to generate a new server ID to match the new configuration.

14.5 Housekeeping

SonarQube offers a couple of useful features for housekeeping activities, and we'll cover them in this section. We'll start by showing how you can create copies of your configuration and restore it in another SonarQube instance, and then we'll jump to the update center to discuss in detail how you can manage existing plugins, install new ones, and get upgrade information about SonarQube's latest version.

14.5.1 Backing up your SonarQube configuration

Imagine that you've been using SonarQube for several years, but only for one department in your organization. A week ago, you had a request from another IT manager who wants to analyze her projects in SonarQube as well. The problem is that the machine hosting your SonarQube installation is old, and adding more projects isn't the best idea. On the other hand, you've done a lot of work regarding quality profiles, general settings, and so on, and duplicating this configuration to a new machine is a time-consuming activity.

Your best bet is a full database backup. But if that's not an option, SonarQube provides some native backup functionality for the following:

- Global settings
- Custom metrics
- Quality profiles (coding rules and alerts)

To see the screen shown in figure 14.21, navigate to the global configuration page and click the Backup link in the left menu's General section. Both actions are simple. To

Figure 14.21 **Configuration backup and restore, made easy by SonarQube**

back up your configuration, click the Backup button, and an XML file containing the configuration data will be locally downloaded. To do the opposite, choose a backup file located on your hard disk, and click the Restore button. Keep in mind that the restore process permanently deletes any previous configuration, and there is no way to get it back, unless of course you've already created a copy.

Before you're tempted to play around with restoring a backup if you don't need to, be aware that it doesn't contain dashboards or filters, anything under Security (users, groups, and roles), or any details of your projects' configurations. Additionally, the profile backup it contains has the same limitations discussed in chapter 13.

Another useful SonarQube feature, especially when something goes wrong in the setup, running, or analysis phase, is the system info page found in the left rail a couple of links below the Backup option. This page gives you detailed information about the SonarQube installation, installed plugins, system environment, Java virtual machine statistics, and various system properties.

> **TIP** Take a look at this page if you're facing problems, to see if something isn't as expected. Also, if you ask for help in a user mailing list, this page might provide you with important information for a quicker and more accurate reply/solution.

14.5.2 Working with the update center

Many times throughout this book, especially in part 1, we discussed plugins that add extra value to SonarQube's core features. In chapter 1, we mentioned that SonarQube is suitable for more than just the Java community; it supports plenty of different programming languages. And as we hinted, SonarQube has been translated to several human languages.

The update center is the place where you perform the following activities:

- Install, remove, or update plugins to provide new features, localizations, or analysis of new languages.

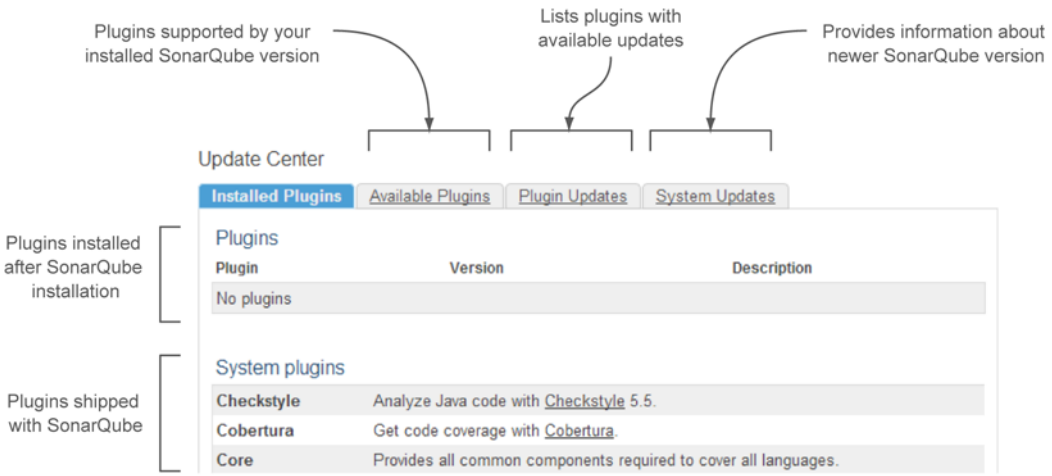


Figure 14.22 SonarQube's update center

- Get information about newer SonarQube versions. At the time this book was published, automated upgrades of SonarQube were unavailable (you have to upgrade manually). This is one reason we've included upgrade scripts in appendix A.

To access the update center, navigate to the global configuration page and click the link in the left menu's System section.

As shown in figure 14.22, there are four tabs to explore. By default, you see the Installed Plugins tab, which lists all the plugins found in your SonarQube installation. Near the top is a list of all the plugins you've manually installed after the initial installation. Below that are the system plugins: those shipped with SonarQube core.

The Available Plugins tab displays the available plugins that are compatible with your SonarQube version. To see plugins in this page, the machine on which SonarQube is installed must have access to the internet. The same applies to the next two tabs. Also keep in mind that not all plugins can be installed in all SonarQube versions. For a complete and updated compatibility matrix, browse SonarQube's online documentation at http://mng.bz/OOf9. The plugins are grouped according to their purpose. Table 14.3 summarizes the available plugin categories.

Table 14.3 Plugin categories

Category	Plugins that…
Additional Languages	Support the analysis of new programming languages.
Additional Metrics	Calculate new metrics during analysis.
Developer Tools	Facilitate developers' everyday life.

Table 14.3 Plugin categories *(continued)*

Category	Plugins that…
Governance	Use existing metrics to create an overview based on practices, indexes, or methodologies, such as SQALE and Technical Debt.
Integration	Integrate SonarQube with third-party systems, such as LDAP and Google Analytics.
Localization	Translate plugins into other human languages.
Visualization/Reporting	Use existing metrics to create reports, or offer new widgets to display quality data in different representations.

TIP If the update center doesn't show any available plugins or other information retrieved from the internet, then you're probably behind a firewall. To fix this, edit the sonar.properties file and set a couple of attributes (`http.proxyHost` and `http.proxyPort`) based on your intranet configuration.

To install a plugin, click its name on the Available Plugins tab and then click the Install button, as shown in figure 14.23. SonarQube downloads the plugin and installs it, but you need to restart SonarQube in order to activate your new plugin. An informative message that reminds you of that is shown at the top of the update center screen.

If you've changed your mind, you can click the Cancel Pending Installations button, and SonarQube will roll back all pending plugin installations. Note that SonarQube needs to be restarted not only when you add plugins, but also when you update or remove them.

The third tab, Plugin Updates, is simple because it looks like a lot like the previous tab. The only difference is that it lists the plugins for which a newer version than the one you're currently running (compatible with your SonarQube version) has been found. If no updates are found, then you should see the message "All of your plugins are up to date."

The last tab, System Updates, doesn't offer any actions. It displays messages about new SonarQube versions and detailed instructions on how to update your installation.

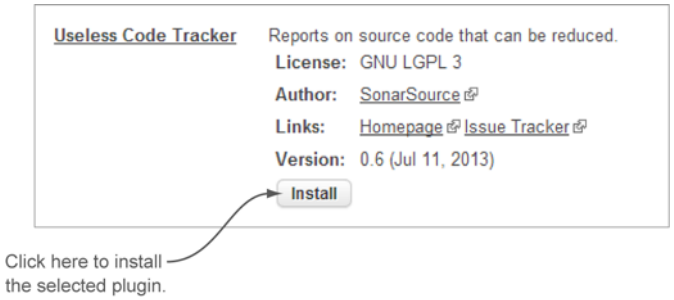


Figure 14.23 Installing a SonarQube plugin from the update center

14.6 Summary

Well done, SonarQube administrator! Nothing now prevents you from feeling comfortable with every detail of SonarQube's global configuration.

You're ready to manage and configure what anonymous users and registered users who haven't configured any dashboards yet will see when they access SonarQube. You've learned what global dashboards are, and that you can place both project widgets and global widgets in them. Furthermore, by now you should be able to do the following:

- Create filters and add them to your dashboards to show only the information you need. Design each filter by entering as many or few as you like of the following: criteria, differential periods, desired programming languages, and how results are displayed (treemap or table).
- Enable the notification mechanism to receive alerts about assigned issues or issues created by you. SonarQube also sends you messages whenever new issues are created for the projects you've flagged as favorites.
- Define your own custom metrics or use the preconfigured metrics that SonarQube ships with, and use them in filters or widgets to display their manually entered measures.
- Adjust a variety of global settings and make SonarQube fit in your development lifecycle instead of fighting it.
- Use the update center to install, update, and remove plugins, and to read the update instructions for the latest SonarQube version.

The next chapter is dedicated to project administration. It will guide you through the different ways you can customize a project analysis and manage its history.

15 Managing your projects

This chapter covers

- Differences between project and global dashboards
- Manual metrics
- Multiple quality profiles
- Understanding the history of a project
- Exploring the rest of project configuration

Welcome to the last chapter of the administration part of the book. Although it may be the last, it's not the least, because it will teach you how to tune your projects and customize them to fit in the Continuous Inspection process you started adopting after reading chapter 9 (right?).

We'll start by explaining the differences between global and project dashboards. What you saw in the previous chapter about managing global dashboards works almost the same for project dashboards. We'll focus only on the details that vary.

Have you ever wondered how SonarQube handles projects with source files from multiple programming languages? If every module contains code from a different language, then you can take advantage of a powerful SonarQube feature that lets you run a single project analysis for all languages. Without that, you would have to instead run several analyses (one for each language), and you could end up

with multiple SonarQube projects. In this chapter, we'll show you how to assign various quality profiles (one for each language) in the same project.

SonarQube also offers a plethora of quality metrics. In addition, you've seen throughout the book that plugins extend SonarQube by feeding it new metrics. But plenty of measures (budget, team size, and so on) are outside pure code quality, so SonarQube can't compute them based on the source code. In such cases you can use the manual measures feature: embedded reporting functionality that displays information in a unified dashboard.

Next we'll explore how to manage your project history. We'll explain basic terms such as snapshots, versions, and events, and we'll present some cases when you need to manage them in SonarQube.

The last part of this chapter covers the rest of the available project configuration. We'll discuss the various settings, similar to what we covered for global settings; and you'll learn how to define links, delete project(s), and modify a project's key.

You'll notice that the order of the topics we'll discuss doesn't follow the order of the links that appear in SonarQube's menu. The reason for that inconsistency is that we decided to present first the features that are most used or most significant, and leave the less important stuff for the end.

Enough with the introduction. It's time to see what's different in project dashboards.

15.1 Working with project dashboards

In chapter 14, we covered a lot about managing global dashboards. You've learned about customizing the look and feel of your dashboards, sharing them, and configuring widgets. Project dashboards are more or less the same.

Because most of the material in this chapter is about projects, navigate to SonarQube's start page and click your favorite project. You're redirected to the project dashboard, where you see various widgets we've covered in previous chapters.

Let's start by exploring the page where you can manage your project dashboards. It's identical to what you saw in our discussion of global dashboards. The only difference is that you can access this page only when viewing an existing dashboard. Don't worry if you haven't created any yet. Remember that SonarQube comes with four default dashboards that appear in the upper-left menu (see figure 15.1). That means every new SonarQube user automatically follows all default dashboards—unless you edit them as you saw in chapter 14. Don't get confused by the fact the first project dashboard is called…Dashboard. All of them are dashboards, and table 15.1 summarizes them.

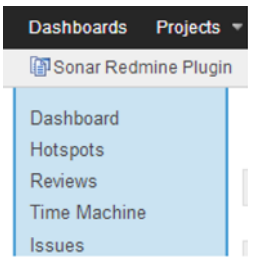


Figure 15.1 SonarQube ships with five default project dashboards. They're available in the upper-left menu when you're viewing a project.

Table 15.1 Default global dashboards shipped with SonarQube

Name	Widgets included
Dashboard	Rules compliance, test coverage, comments and duplications, size metrics, complexity, and package design. This is the dashboard you land on by default when you click-through from the global level to the project level.
Hotspots	Most violated resource, most violated rules, and several instances of the metric hotspot widget (highest untested lines, highest complexity, highest duplications, and so on).
Reviews	All widgets related to reviews and action plans (review activity, unplanned reviews, reviews by developers, my active reviews, and so on).
Time Machine	A timeline widget about coverage, rules, and complexity, and several instances of the history table widget for the most important metrics.
Issues	Unresolved issues by status, unresolved issues by assignee, action plans, false positive issues, and unresolved issues.

Now that you're viewing a (default) project dashboard, you can see in the upper-right corner the links—the same ones you see for global dashboards—for managing dashboards and configuring widgets. We won't discuss that further because what you already know from the previous chapter applies here as well. Don't forget that you can share project dashboards only when you're granted global system administration rights.

But keep in mind a small difference about widgets in project dashboards. You don't need to specify the project (as you do in global dashboards), because it's assumed that the widget will display data from the current project.

Even though it might be considered weird, you can use global widgets in project dashboards exactly as you do with global dashboards.

Figure 15.2 shows the same widget (rules compliance) in a global dashboard and a project dashboard. You can see the extra widget header in the global dashboard for the project description.

As a last reminder, you don't have to create every dashboard your users want to see. They can create their own private global and/or project-level dashboards and follow/unfollow default dashboards created by global administrators.

Now we're leaving dashboards behind and moving on to a more sophisticated topic. In the next section, we'll discuss the available project settings that assist you in applying Continuous Inspection practices in your development process.

15.2 Adopting Continuous Inspection more quickly

In chapter 9, we explained in detail the idea of Continuous Inspection and covered the steps to achieve it. We even dedicated a whole chapter (10) to code reviews and how SonarQube makes them a piece of cake, to underline the importance of this practice when adopting Continuous Inspection.

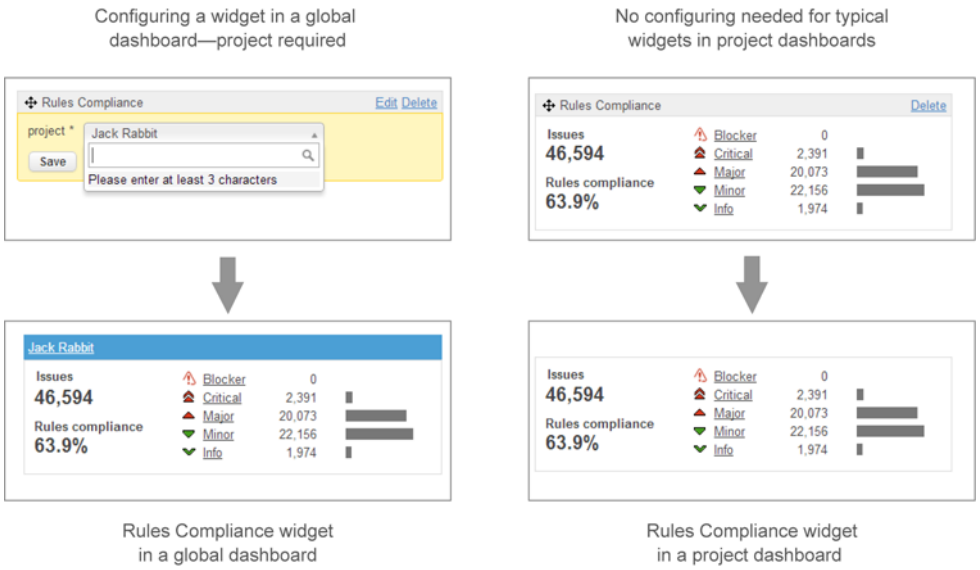


Figure 15.2 Adding a widget in a project dashboard is simple because you don't need to specify the project, as you probably expect from chapter 14.

This section elaborates on this topic and discusses some project-dependent features and configuration that will help smooth the adoption of the Continuous Inspection practice. We'll start by showing how you can assign multiple quality profiles in a project and the reasoning behind doing so. Then we'll explain manual metrics, and finally you'll learn when and how you might want to exclude part of your source code from analysis.

15.2.1 Assigning quality profiles

In modern layered systems, a common and best practice is to separate source code in different libraries (called *modules*) based on the programming language used. For instance, imagine a project with two modules. The first module contains Java source code files, and the other one contains files related to the web interface (JSP/JSF/XHTML pages). SonarQube can analyze this language-based multi-module project during one analysis, which is convenient.

In chapter 1, we talked about running multiple analyses for multiple languages in the same project, which is different from what we're discussing here (multi-module language-based project organization). Both approaches are possible, both are common, and they're supported by all available analyzers (clients). If you don't know/remember how to trigger such an analysis, refer to appendix B.

As you learned in chapter 13, SonarQube ships with several default quality profiles. Accordingly, every language plugin adds at least one default quality profile. But as we explained, it's a good idea to create your own profiles and associate them with projects. This can be done two ways. On the quality profiles administration page, you can

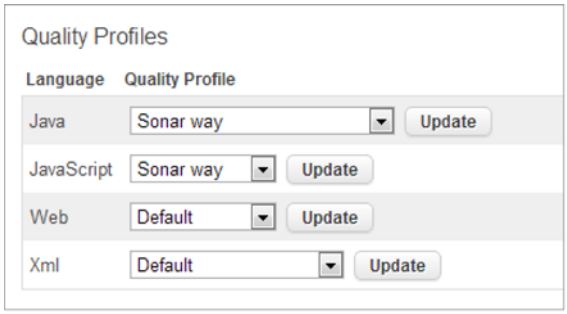


Figure 15.3 Assigning project quality profiles for supported languages

associate a project with the selected profile, but this requires that the logged-in user have system administration rights.

You also learned in chapter 12 that it's a good practice to create an admin group for each project and assign them administrator rights only for that project.

By following this practice, you can have project admins manage the selected quality profiles. They can't create new profiles, but they can assign for each language which profile SonarQube should use to analyze the source code.

Figure 15.3 shows the project Quality Profiles page. Yours might be different, because this page is dynamically created. A separated option is displayed for each programming language supported by the current SonarQube installation. Your project might contain only Java and JavaScript source files, but SonarQube doesn't know that, so it displays all available languages. You can navigate to this page (when viewing a project) by clicking the Configuration link at upper right on your screen. In the drop-down menu that appears, select Quality Profiles. If you don't see such an option, it means you haven't yet created a quality profile, as you saw in chapter 13.

After that, all you have to do is select your favorite profile for your project languages and click the Update button for each selection. For instance, clicking Update for the Java language doesn't save a selection made for XML. Run a new analysis, and you'll see language-specific metrics calculated by SonarQube for each module.

Speaking of metrics, have you ever wondered if SonarQube supports measures that can't be computed from source code analysis but that are important to track? The next section answers that question.

15.2.2 Defining your own metrics

We've seen many posts in the SonarQube user mailing list from users asking how to group projects with attributes that don't exist in computed SonarQube metrics or in project properties. Assume that you work in a large organization, and SonarQube hosts hundreds of projects assigned to different development teams. Team members would prefer to see in their global dashboard only the projects they're responsible for,

but there's no common characteristic across the project set on which to base a filter. SonarQube doesn't have any built-in feature to distinguish projects based on a condition that isn't related to computed metrics. So what now? If only you could add a property (such as team ID or name) to your project and set its value! Then you could create a filter based on this property and show only the correct projects.

ADDING A NEW MANUAL METRIC

The answer is simple: it's called *manual measures*. SonarQube allows you to create as many manual metrics as you want and specify their measures for each project. In this section, we'll show you how to manage manual metrics at a global level and how to set their values and include them in dashboards and filters.

> **NOTE** Metrics and measures are two different concepts in SonarQube. A *metric* is a definition/specification, and a *measure* is the result of a metric computation by an analysis.

To add a new manual metric, click the Settings link at upper right on your screen and then select Configuration from the drop-down menu. This takes you to the global administration page we covered in chapter 14. Click Manual Metrics, and you see the page shown in figure 15.4. Keep in mind that although project-level rights are sufficient for most of the configuration discussed in this chapter, to add manual metrics you must have global administration permissions. See chapter 12 to refresh your memory about permission levels.

The Manual Metrics page is similar to the one we discussed for global dashboards. It lists all the manual metrics and, at right, provides a form you can use to create a new

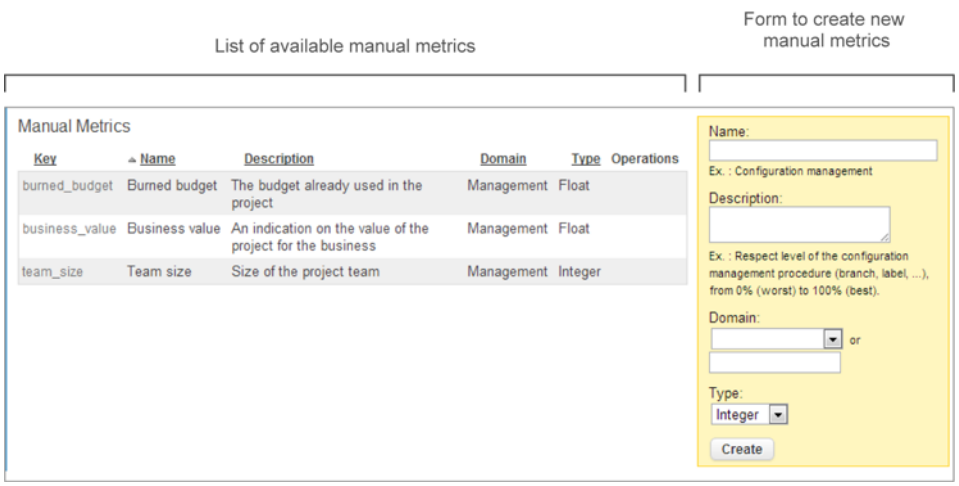


Figure 15.4 SonarQube ships with three manual metrics. The administration page is simple: it lists all available metrics and provides a form to add new or edit existing ones.

one. SonarQube comes with three preconfigured manual metrics you can use in your projects.

- *Burned Budget*—The budget used so far by the project
- *Business Value*—How important the project is to the business
- *Team Size*—Project team size

Let's create a new metric to indicate the project's development team. Enter the name of the metric (Team Name/ID) and a description that reminds you of its purpose, and select or enter a domain. Finally, choose the metric type. The available options are as follows:

- Integer
- Float
- Percent
- Level
- Text
- Yes/No

In the example, you only want to record the name of the in-charge team, so the Text type is what you're looking for. Click the Create button, and the metric is automatically added to the table. In addition, the Operations links are activated so you can edit or delete the metric, as shown in figure 15.5. (Preconfigured metrics can't be modified or removed.)

Now you can use the metric to create a new filter that lists all team projects, as discussed in chapter 14. Of course, you aren't finished. You need to set each project's metric values, as we'll discuss next.

Setting Measures of Manual Metrics

Select a project to view, and click the Configuration drop-down option Manual Measures. Figure 15.6 shows the page for managing measures of manual metrics. Because you haven't set any values yet, your page probably looks identical to the figure. To add a new measure, click the Add Measure link at upper right.

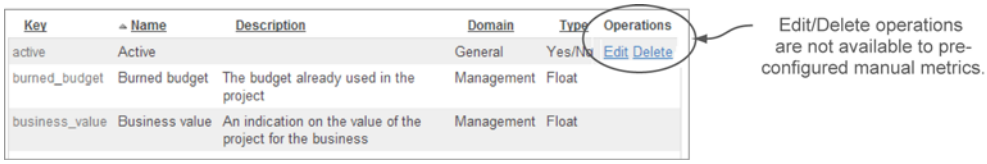


Figure 15.5 The Edit and Delete links are shown only in custom manual metrics.

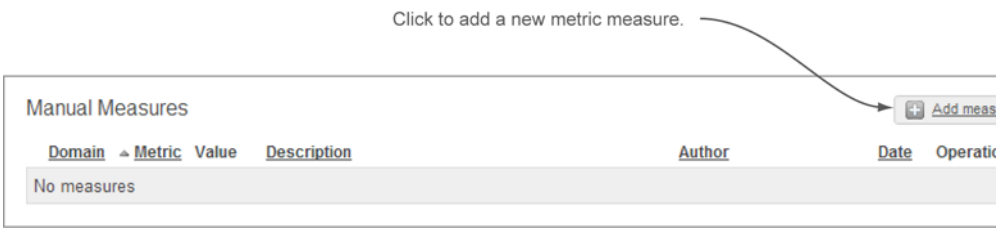


Figure 15.6 Manual measures (empty) administration page

As you can see in figure 15.7, the Add Manual Measure page is pretty simple. Just choose a manual metric from the corresponding list, and two new attributes (Value and Description) appear below it. Fill in the information, and save your settings. Your new manual measure is (almost) ready for use like a typical measure (filters, widgets, and so on), as we'll show you in a minute. There's one last step.

As the upper part of figure 15.8 shows, in the custom measures list view, a golden marker appears to the left of the measure. This indicates that the measure is still pending, meaning you need to run a new project analysis so its value will be integrated with the other SonarQube metrics. The lower part of the figure shows that after the analysis, the golden marker and the corresponding message disappear.

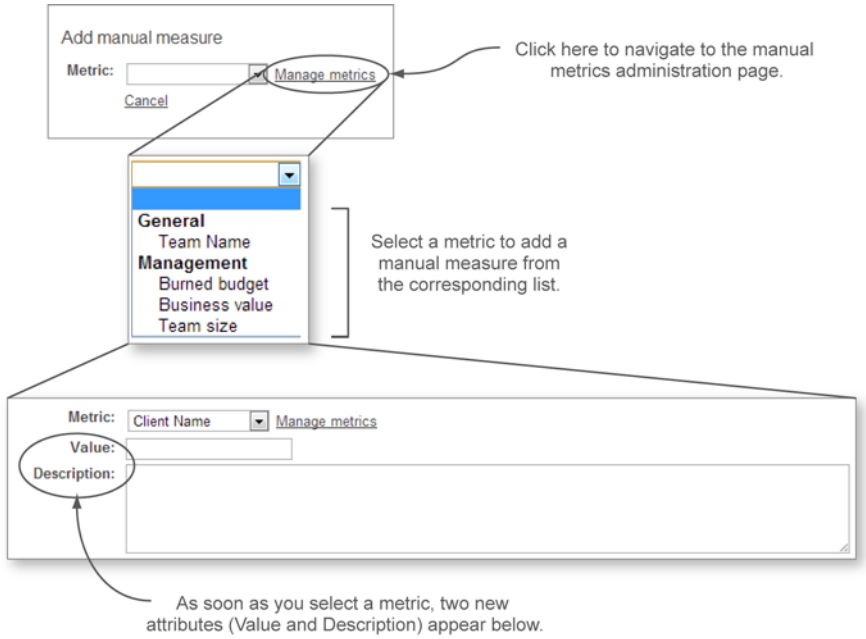


Figure 15.7 Adding a new manual measure

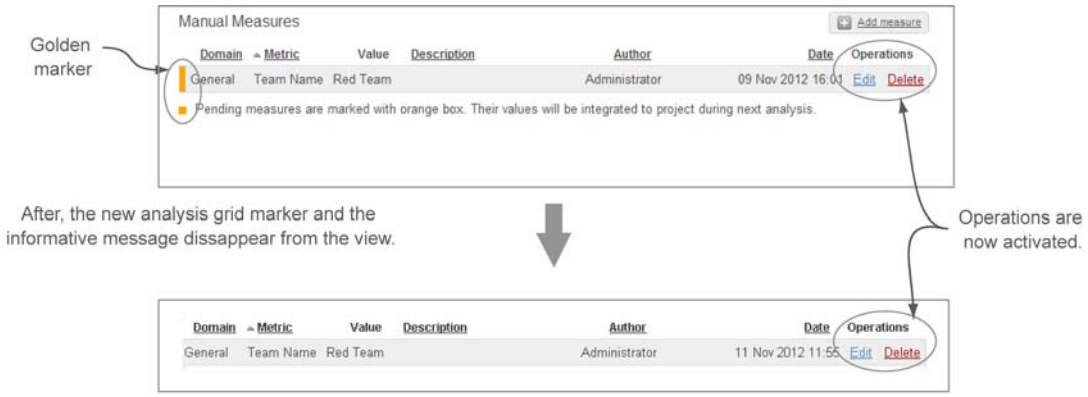


Figure 15.8 A golden marker points out which measures have not yet been integrated with the rest of the SonarQube metrics. Running a new project analysis removes these markers.

Displaying Custom Measures in Dashboards

Showing custom measures in a dashboard is easy. If you don't remember how to add widgets to a global or project dashboard, go back to chapter 14 to refresh your memory.

First, as step 1 in figures 15.9 shows, locate the custom measures widget and add it to your favorite dashboard. (Don't forget that you can add a widget in both project and global dashboards.) Then select the custom measure(s) you want the widget to display. When you're done, click the Save button at the bottom of the widget. That's it. The widget now shows your custom metric(s).

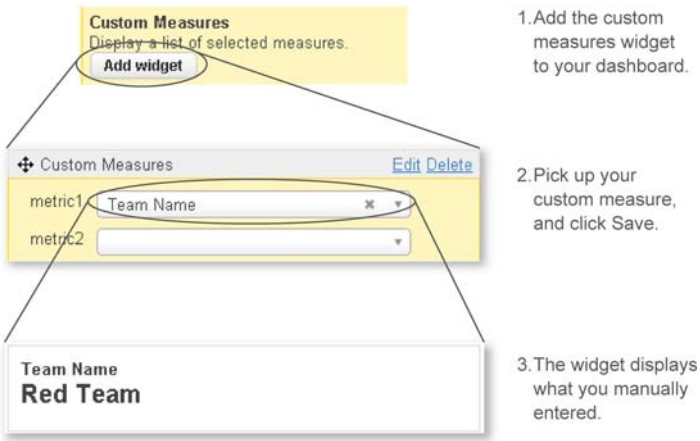


Figure 15.9 Displaying custom measures in a dashboard takes two steps: add the custom measures widget, and select the measures you want to show.

TIP The custom measures widget shows only the newest metric values integrated with the rest of the SonarQube metrics during the last project analysis. That means you may enter multiple values for the same measure, but only the last one entered is displayed in the corresponding widget. Custom measures behave just like other metrics calculated by SonarQube: you can add them to a timeline widget and see their historical evolution.

Well done! Now that you've mastered SonarQube's custom metrics, it's time to change the subject. The next section will introduce the concept of omitting some files from SonarQube analysis.

15.2.3 Excluding source code from analysis

Assume that you recently integrated your system with a promising open source framework. After a while, you discovered some bugs that seriously affect your system but that aren't scheduled for resolution in any future release of the framework. So you decide to fix them locally.

You download the source files, make copies of the five classes involved, fix the issues, and place the corrected files with the rest of your project. So far, so good. The next morning, you open SonarQube's web page, and—what a mess! Most of the metrics (coverage, issue compliance, documentation, and so on) have decreased. You quickly drill down to see what happened, and you realize that the classes you copied from the open source framework appear at the top of the drilldown lists. No tests, no documentation, and many broken rules for your favorite quality profile. These files destroy the quality of your project.

Don't panic! Remember that you're in the world of SonarQube, where (almost) everything is possible one way or another. Once again, in this case, there's an easy-to-implement solution. Click the Settings option in the Configuration drop-down menu, and then select the Exclusions category.

(We'll explore most of the available settings in the next section. We're mentioning exclusions here because of the importance of this feature and the flexibility it adds to SonarQube analysis.)

Figure 15.10 shows the exclusions settings page. You can add several values both for sources and tests based on a pattern. The instructions and examples provided by SonarQube on this page are awesome, and it's redundant to repeat them here.

We prefer to give you a comparison of all available exclusions discussed in this book. You saw the Switch Off Violations plugin in chapter 13 and the Cutoff plugin in chapter 9. Both of them exclude files from SonarQube analysis. Table 15.2 summarizes their basic features along with what we explored in this section.

Which one you should use? It depends on what you want to achieve. You can even combine them. In general, SonarQube's core exclusions feature is used when you want to ignore source files during analysis. Switch Off Violations fits when you want to bypass issue creation without modifying the quality profile. And finally, keep in mind that the Cutoff plugin is convenient to exclude legacy parts of the code.

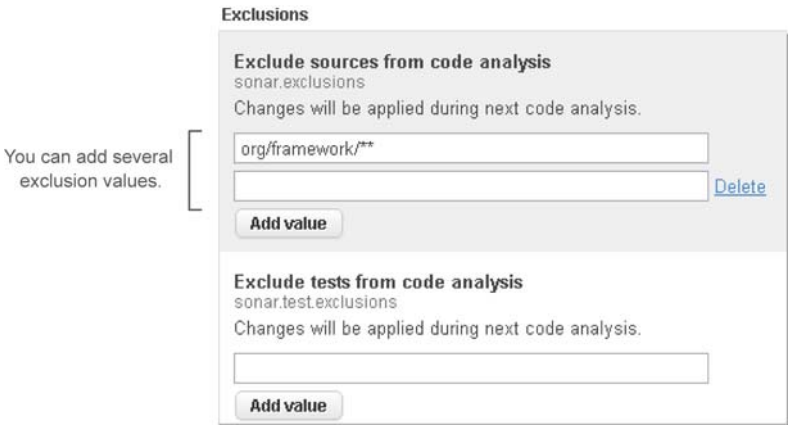


Figure 15.10 By modifying exclusions settings, you can make SonarQube exclude source files based on patterns.

The last part of this section is dedicated to the project's history. You'll learn how to manage it and how it interacts with the differential views we explored in chapter 10.

Table 15.2 SonarQube's exclusion options

Name	Description	Plugin	Metrics affected	Project / Global
SonarQube's exclusions settings	Exclude files from being analyzed by SonarQube based on patterns	No	All	Project
Switch off Violations plugin	Exclude issues in a fine-grained way by rule, name/path, line or range of lines, and file content	Yes	Only issues. Complexity, duplications, and so on are still calculated.	Both
Cutoff plugin	Exclude files from being analyzed by SonarQube based on a predefined date or period threshold	Yes	All	Both

15.2.4 Understanding versions, snapshots, and events

Before we start explaining some terms, let's browse the project's history. To do so, click the History link, found in the project's Configuration drop-down menu. SonarQube has a clever, flexible mechanism to maintain historical information for a project. This information isn't limited to previous analyses but is expanded to include versions and events.

In figure 15.11, notice that all the details you could want about your project's history are consolidated on just one page. Starting from the left column, you get precise information about the date and time of analysis execution. Year and Month column data are omitted if they're the same as the previous analysis; this makes the table even

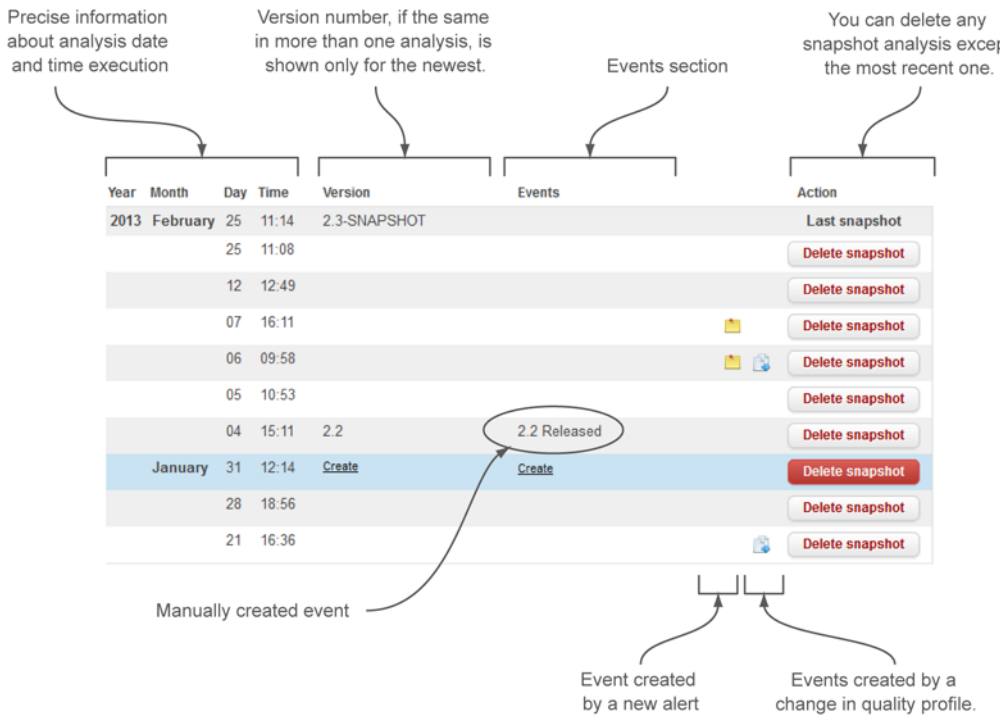


Figure 15.11 Every detail of your project's history on one page

more readable. Next, the Version column shows the `sonar.projectVersion` property provided in SonarQube Runner or in an Ant task. In Maven projects, it can be automatically retrieved by the pom.xml file (that is, the `<project><version>` property). Next is the Events section, which is composed of three columns: the first displays manually created events, and the other two indicate events created by alerts or quality profile changes. The last column provides a button to delete a snapshot analysis. Note also that SonarQube doesn't let you delete the most recent snapshot analysis.

Before we move on, let's elaborate on three terms we used in the previous paragraph: *snapshot*, *event*, and *version*. Table 15.3 explains them in detail.

Table 15.3 History terms

Term	Description	Comments
Snapshot	Every time SonarQube runs a new project analysis, its results are stored in the database, and a new snapshot of the project is created.	Snapshots aren't forever. Recall what we discussed in chapter 14 about the database cleaning mechanism and how you can modify the default settings.
Version	The project's version number is passed as parameter in SonarQube analysis.	The latest snapshot of a version is always kept in the database, even if it breaks the database cleaner rules.

Table 15.3 History terms *(continued)*

Term	Description	Comments
Event	Events either are noteworthy facts that occurred during project analysis or can be manually added.	A snapshot may be associated with more than one event. Snapshots with events are never deleted automatically from the database, although someone with project administration rights can do that manually.

Now that we've clarified the basic terms, let's go back to figure 15.11. You might be thinking right now that this is a static page, and the only interaction is through the Delete Snapshot button. Well, move your mouse over a row.

As figure 15.12 shows, the row is highlighted; and depending on its content, various actions appear. For instance, if there's no information about the version or custom event, you see a Create link in the corresponding places. On the other hand, if the row (that is, a snapshot analysis) holds data about the version, you see links to rename or remove it. For the Events column, in addition to the Rename/Remove links, you always see a Create link, because you can have more than one custom event associated with a snapshot analysis.

As you might expect, you can rename an event or a version. To do so, click the Rename link and enter the new description in the resulting input. Removing data is just as easy: click the Remove link.

> **TIP** You're probably wondering why you need a Delete Snapshot button, because SonarQube takes care of older snapshots. In some cases, you need to delete a snapshot such as wrong rules in a quality profile or any other wrong parameter passed to the analysis. Instead of waiting for the database cleaner, it's much better to do it yourself.

We've now covered the most significant project administration topics that you need to know. The rest of the chapter discusses subjects that are less important but still useful to know.

15.3 Exploring the rest of the project configuration

This section explores in more depth some low-level project administration options and features covered in previous chapters. Although you might never use them, it's

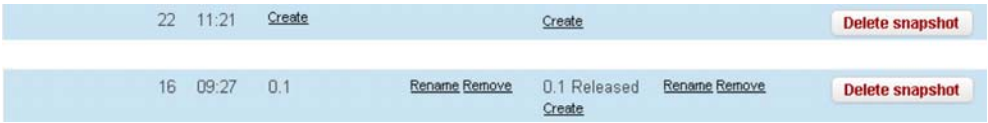


Figure 15.12 Moving your mouse over a row displays various actions depending on the row's content. You can create a new version, add multiple custom events, rename them, or remove them from the selected snapshot.

good to know they exist, in case some day you need them. We'll discuss assigning project roles, setting various project links, modifying the project's key, and deleting project(s). Finally, we'll show you some useful miscellaneous project settings.

15.3.1 Changing permissions

In chapter 12 we talked about user, group, and role management. You learned that it's a good idea to add users with administrator privileges for each project and let them manage project security without bothering the SonarQube administrator.

For that purpose, there's a Roles link in the project's configuration drop-down menu. The page shown in figure 15.13 looks very similar to the Roles pages you saw in chapter 12. The only difference is that here, you can manage only the permissions of the current project you're viewing. You can use the Select links to add/remove individual users or groups.

15.3.2 Setting project links

SonarQube lets you define up to 10 external project links by clicking the Links link on the project configuration menu (see figure 15.14). These links can be categorized as follows:

- *Standard links*—Can be passed as SonarQube execution parameters or discovered automatically in Maven projects. These links can't be modified using SonarQube's UI.
- *Custom links* (up to five)—Can be added/edited only by using SonarQube's UI.

Table 15.4 lists the available standard links and compares them with a generic custom link.

Figure 15.13 Project Roles page, which is similar to the global roles administration page

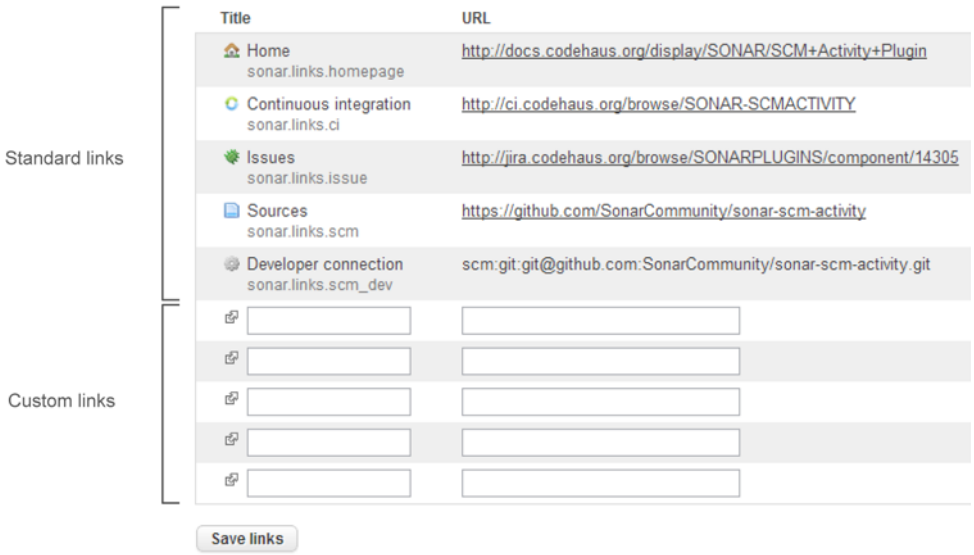


Figure 15.14 **You can define up 10 ten links for every project, but only half of them can be generic and managed though SonarQube's UI.**

Table 15.4 **Standard links**

Link	Description	Property Name	Maven attribute	Usage in SonarQube
Home	Links to the project's home page	`sonar.links.homepage`	`<url>`	Dashboards (description widget) Filters
Continuous Integration	Links to the job page of a CI engine (Jenkins/ Hudson, Bamboo)	`sonar.links.ci`	`<ciManagement>` `<url>`	Dashboards (description widget) Filters
Issue Manage-ment System	Links to the issue manage-ment system (JIRA, Mantis, Redmine, Trac, and so on)	`sonar.links.issue`	`<issueManagement>` `<url>`	Dashboards (description widget) Filters
Sources	Links to the project sources	`sonar.links.scm`	`<scm>` `<url>`	Dashboards (description widget) Filters

Table 15.4 Standard links *(continued)*

Link	Description	Property Name	Maven attribute	Usage in SonarQube
Developer Connection	URL for developers to get a copy of the source code	`sonar.links.scm_dev`	`<scm>` `<developerConnection>`	Dashboards (description widget) Filters
Custom	Anything useful (Javadocs, other documentation, and so on)	N/A	N/A	Dashboards (description widget)

TIP In Maven projects, you can override the pom.xml attributes by setting the corresponding `sonar.links.*` property during SonarQube analysis.

15.3.3 Modifying the project key

If you think modifying a project's key in SonarQube might be a useless feature, imagine the following (real-world) scenario. You've been using SonarQube for the last year for your project. That means you have a lot of historical data. For some reason (policies, wrong key, and so on), you need to modify the project's key, which is a trivial activity. Next, you run a new SonarQube analysis, and BOOM!

Why does SonarQube now display two projects with the same name (one with the previous key and one with the new)? Where is the historical information in the project with the new key? It's a complete mess.

If you're wondering why these evil things happened, the answer is simple. SonarQube identifies projects by their keys. If a key isn't found in the database, it's considered new, and the project is created from scratch. That's why you see a replica (in terms of description) of the initial project.

To solve this issue, SonarQube allows you to modify the project's key by choosing Update Key in the Configuration drop-down menu. The resulting page is simple (see figure 15.15): all you have to do is enter the new project key and click the Rename button. Keep in mind that you need to modify the project key in SonarQube's UI before triggering a new analysis of the project with the new key.

In other words, before you run a new analysis, make sure both keys (project key in source files/analysis properties and project key in SonarQube) are identical. Otherwise you end up with two different projects in SonarQube.

Figure 15.15 SonarQube lets you edit a project's key, but you should do this before triggering a new analysis of the project with the new key.

15.3.4 Deleting projects

SonarQube offers two ways to delete a project:

- Single-project deletion
- Bulk deletion

Although the second option is more like a global administration feature, we decided to list both of them in this section to make it easier for you to use this book as a reference and not a one-time read.

If you want, for any reason, to delete a project from SonarQube's database, choose Project Deletion in the project's Configuration drop-down menu, and then confirm the action. Keep in mind that there's no way to undo this operation: double-check that you really want to delete the project and its history, and/or back up your database first!

Bulk deletion is available by choosing Bulk Deletion from the global configuration menu. You see a list of all available projects with pagination (see figure 15.16). At the top of the screen is an input where you can type some text and filter the projects. Select all the projects you want to delete, or choose the Select All check box to quickly select all projects on the current page.

> **TIP** When you navigate between pages, previous selections aren't remembered. For instance, if you select project X from page 1, navigate to page 2, select project Y, and click Delete, only project Y will be deleted.

Be aware that if your projects span several pages and you want to select all of them, you must use the link that appears to the right of Select All, as shown in figure 15.17. Finally, SonarQube, before executing the deletion, requests for a final confirmation.

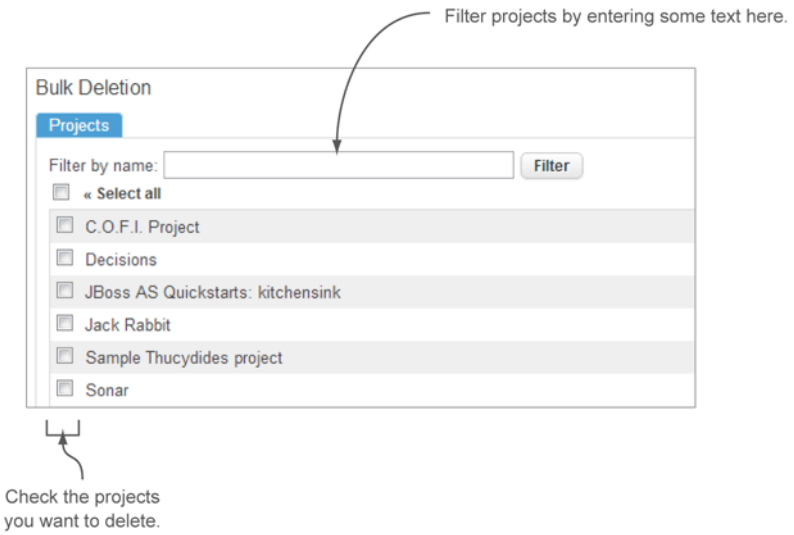


Figure 15.16 You can delete several projects at the same time by using the Bulk Deletion feature.

Figure 15.17 Checking the Select All (in current page) option activates a new link to select all projects.

As with single-project deletion, there's no way to undo this action unless you have a database backup in place.

15.3.5 Miscellaneous settings

The last option in the project configuration menu that we haven't discussed yet is the Settings link. The settings page is similar to the global settings feature you saw in chapter 14. We've already covered all the available categories in chapter 14 or elsewhere; for instance, the Code Coverage category was discussed in chapter 3, and the Differential Views category in chapter 9.

Keep in mind that many plugins provide not only global settings but also settings at a project level. Every time you install a new plugin, it's a good idea to visit this page to explore the available project settings.

15.4 Summary

Well done, SonarQube Master! You can be proud that you know everything you need to work with SonarQube in action. This chapter covered the topics of project administration.

You've learned what's different in project dashboards and how you can improve your Continuous Inspection process by doing the following:

- Assigning multiple quality profiles for multi-module, multi-language projects.
- Creating manual metrics, setting their measures, and using them in filter criteria and widgets as you do the rest of the metrics calculated by SonarQube.
- Excluding parts of your source code from analysis. We also presented a comparison of the three ways you can do this (core SonarQube, the Switch Off Violations, and the Cutoff plugins).
- Managing your project's history by creating custom events or new versions, or deleting useless snapshots.

Finally, we covered some low-level administration features such as assigning project roles, changing the project key, and deleting one or many projects at the same time. We also discussed setting standard or custom links (project-related) and where you can use them.

The final chapter of this book is dedicated to developers and will give you a step-by-step guide to create your own plugin using the SonarQube API. To make this fun and useful at the same time, we've chosen to implement a plugin for Redmine (www.redmine.org), an alternative to the JIRA issue-tracking system.

16 Writing your own plugin

This chapter covers

- Understanding SonarQube architecture
- Writing code for a SonarQube plugin
- Creating your own widgets
- Supporting new languages

Welcome to the last chapter of *SonarQube in Action*, which will teach you how to implement your own SonarQube plugins. If you wonder why you might want to write your own plugin, here's a non-exhaustive list of possible needs:

- Integrate SonarQube with external tools such as the example we'll show you in this chapter.
- Create customized reports based on the metrics computed by SonarQube.
- Translate SonarQube in a new language (localization).
- Add support for a new programming language.

We've told you that SonarQube isn't Java-centric and that even though our examples are Java-based, the same ideas apply to other languages. From an analysis standpoint, that's true; but in this chapter, that changes. What we're going to show you requires you to be familiar with Java, jRuby, and Ruby on Rails, because these

are the languages currently supported for plugin writing in SonarQube. You'll also be using Maven to build your plugin, but we'll feed you the commands, so only a passing familiarity with Maven is required.

We'll start by briefly explaining how SonarQube works internally when you launch an analysis. It's important to understand what's going on behind the scenes when SonarQube runs a plugin. The first thing you'll learn is the difference between *decorators* and *sensors* and when they're executed during an analysis.

Then we'll give you a step-by-step guide for implementing your own plugin. Together we'll integrate SonarQube with Redmine (www.redmine.org), an alternative to JIRA. Redmine is more than a ticket system, but for the purpose of this chapter we'll focus only on ticket-related features. It's recommended, although not required, that you install Redmine (http://bitnami.org/stack/redmine) so you can run and see the plugin in action. Plugin writing can include many tasks, and we'll cover most of them in this chapter: adding metrics, class loading and Dependency Injection, widget creation, and internationalization, just to name a few.

Finally, we'll look at how you can make SonarQube support new languages. You won't implement the entire plugin, but you'll get the basic idea; and you'll discover the SonarSource Language Recognizer (SSLR), a library that simplifies the development process of a language plugin in SonarQube. Let's start by exploring SonarQube's architecture.

16.1 Understanding SonarQube's architecture

Now that you've gotten this far in *SonarQube in Action*, we're betting you're wondering how to implement your own plugin to integrate your favorite tool or external application with SonarQube. If so, this is the chapter you've been waiting for. But before you start writing code, you need to familiarize yourself with SonarQube's architecture and the analysis process, as shown in figure 16.1.

In general you can run a SonarQube analysis with three different tools: Maven, Ant, and SonarQube Runner (recommended by SonarSource). Let's call them *clients*. Appendix B discusses all of them in detail, but no matter what your preferred method is, what happens in SonarQube during an analysis is the same.

As soon as an analysis is triggered by a client, the first thing SonarQube does is import the source code and (optionally) test files. This is done by the appropriate *language plugin*. As you know, SonarQube ships with Java support but you can add support for other languages by installing additional plugins. Once SonarQube has the source files loaded, it's time to analyze them with a variety of analyzers that can create or update measures. These analyzers are either embedded in SonarQube core, included in language plugins, or provided by non-language-specific plugins.

The analyzers can be grouped in two categories. Analyzers that are able to create new measures, compute their metrics for each resource (method, file, or package), and store them in SonarQube's database are called *sensors*. After all the sensors have done their jobs, other analyzers, called *decorators*, are triggered. They can aggregate

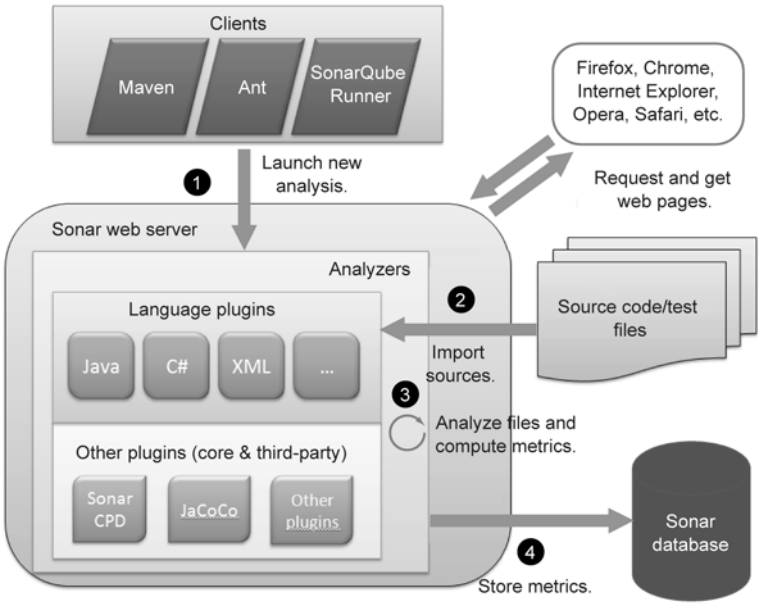


Figure 16.1 SonarQube's internal architecture

the metrics that have already been computed at low levels into new, higher-level metrics. We'll come back to the details of these two flavors of analyzers in the next section.

When the analysis is over, meaning that all sensors and decorators have completed their tasks, a new snapshot is created in SonarQube's database, and all the newly computed metrics are connected to it. At this point, you can see the data by using your favorite browser to send requests to the SonarQube web server.

Now that you have a general idea of how SonarQube works, it's time to move to the chapter's core topic: plugin writing. We decided to implement a SonarQube plugin for the Redmine project management tool as a way to demonstrate most of the concepts of plugin writing in a real-world and easy-to-understand scenario. We'll focus on Redmine's issue-tracking features to provide functionality similar to the SonarQube JIRA plugin (http://docs.codehaus.org/display/SONAR/JIRA+Plugin). The purpose of the example is to retrieve from Redmine—using its Rest API—a project's unresolved issues and display them by priority in a SonarQube widget.

> **NOTE** The source code shipped with the book includes only the first version of the plugin. In the meantime, the plugin might have been updated; to be sure you have the latest version, you can get the code directly from the GitHub public repository (https://github.com/SonarCommunity/sonar-redmine).

16.2 Implementing the Redmine plugin

This section will show you how to create a SonarQube plugin, step by step. The purpose of the example plugin is to collect open project issues from a Redmine installation and present them in a dashboard widget, grouped by priority. We'll try to avoid

getting bogged down in the details of the integration of SonarQube with Redmine and focus on the important stuff—plugin-related code and configuration.

You'll start by setting up your development environment and creating the outline of the plugin. This is something you can do in many ways, but we'll show you the easiest, especially if you're not an experienced plugin developer. Then you'll learn how you can specify the configuration and settings exposed by your plugin. After that, we'll look at how new metrics can be described and configured. Then we'll dig in to the difference between sensors and decorators and discuss how Dependency Injection works in SonarQube. You'll learn how to compute measures using sensors. Then we'll show you how to create a new widget to display the metrics you calculated on SonarQube dashboards. Finally, we'll discuss a simple decorator example and show you how to support internationalization.

16.2.1 Creating the plugin Maven project

Let's begin by ensuring that you have all the necessary tools available in your machine. You need Java 6 or above. You also need to install Maven (version 2 or later) to build and package the plugin (http://maven.apache.org).

> **NOTE** If you install a version of Java higher than 6, such as Java 7, your plugin must be compiled for Java 6 compatibility.

Now it's time to create the plugin project. There are a couple of ways to do it. You could create the project from scratch, but then you'd have to manually edit the pom.xml file to enter some required property values. Alternately, you can copy an existing SonarQube plugin or the SonarQube reference plugin (http://mng.bz/FIZC). Although this approach sounds like a good idea, it adds extra work such as renaming classes, modifying headers, changing pom attributes, cleaning out unneeded code, and so on. Furthermore, inexperienced developers may get confused about what they actually want to implement. But you might use this technique later, if you're familiar enough with plugin writing or if you're going to implement a plugin that's similar to an existing one.

To make it easy, we've implemented a simple Maven archetype that automatically generates a plain SonarQube plugin that includes only some basic classes you'll need. The tutorial that follows in this chapter is based on this Maven archetype. Download the archetype's source code compressed file from http://mng.bz/8hm3, unzip it, and install it in your local Maven repository by running the `mvn install` command. Then you're ready to create a new SonarQube plugin based on this archetype by running the following Maven command: `mvn archetype:generate -DarchetypeCatalog=local`. You'll be prompted for a series of values. Table 16.1 details them.

Table 16.1 SonarQube plugin Maven archetype input parameters

Parameter	Explanation	Default value	Value you should enter
`groupId`	Maven group ID. It must be the same for all plugins submitted back to the SonarQube Community, so a default value is provided, and it's highly recommended that you not change it.	Yes	
`artifactId`	Maven artifact ID. Used in pom.xml and for creating the project directory.	N/A	`sonar-redmine-plugin`
`Version`	Plugin's version. Used in pom.xml. Usually, the first release of a SonarQube plugin is version 0.1.	Yes	
`package`	Full package name where plugin classes will be placed. For all plugins submitted back to the community, the packages should start with `org.sonar.plugins` followed by the plugin's key. The actual value is composed of the default value and plugin's key. If you don't intend to submit the plugin to the community, you can change this value by editing your pom.xml file.	Yes	
`inceptionYear`	Used in pom.xml and headers license.	No	The current year
`organizationName`	Used in pom.xml and headers license. Also displayed in SonarQube's update center information.	No	Your company's name or your full name
`pluginDescription`	Used in pom.xml to describe the purpose of the plugin, and displayed in SonarQube's update center information.	No	`SonarQube Redmine Plugin: Integrates SonarQube with Redmine in various ways`
`pluginKey`	Very important parameter. Used by the archetype to build the names of packages, auto-generated plugin classes, project folders, and so on. First letter should always be in uppercase.	No	`Redmine`
`pluginName`	Used in pom.xml and license headers. Also displayed in SonarQube's update center information.	No	`SonarQube Redmine Plugin`
`sonarVersion`	Used in pom.xml to specify the minimum SonarQube version supported by the plugin. Keeps SonarQube from loading incompatible plugins.	No	3.0 (if you're not sure) or something greater if you're going to use features introduced after version 3.0

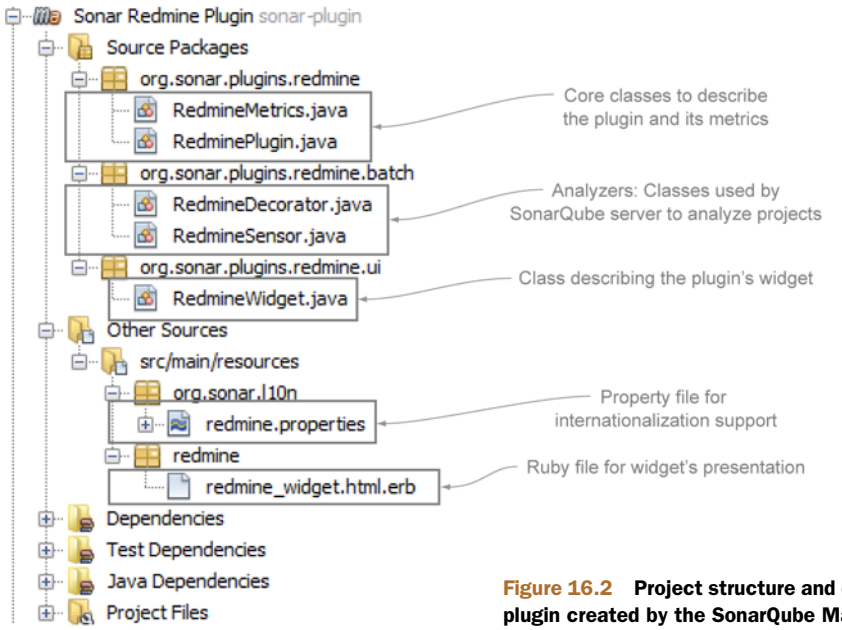


Figure 16.2 Project structure and contents of the plugin created by the SonarQube Maven archetype

When you're done with all these parameters, the archetype creates your SonarQube plugin project in a directory named sonar-redmine-plugin. Open the project with your favorite IDE, and you'll see the structure and contents shown in figure 16.2.

The Maven archetype creates a generic plugin outline that includes the following:

- Two classes (`RedmineMetrics` and `RedminePlugin`) describing the plugin configuration and its metrics
- Two analyzer classes: a sensor (`RedmineSensor`) and a decorator (`RedmineDecorator`)
- Two files describing the dashboard widget: a Java class (`RedmineWidget`) and a Ruby file for the actual representation (redmine_widget.html.erb)
- One property file to support internationalization (redmine.properties)

The plugin is fully usable at this point, because there's demo code in the auto-generated files that explains the purpose of each class. You can build and deploy it in your SonarQube instance immediately if you like. But right now, it doesn't add any meaningful functionality, so the next step is to make it valuable.

16.2.2 Defining the plugin's available configuration

The first thing you should consider is the configuration settings that will be exposed to the end user. You want to collect issues from a Redmine installation for a specific project, so you need to know the Redmine host URL and project. Furthermore, according to the Redmine API documentation (http://mng.bz/AC2P), you need an API access key, which a logged-in user can find on the Redmine account page in the right pane of the default layout.

The `RedminePlugin` class is where you define these properties. Take a close look at the following listing, which shows most of the actual `RedminePlugin` class (import statements and header definition are omitted for space).

Listing 16.1 `RedminePlugin` class: properties definition and plugin extensions

```
@Properties({                                          <-- Annotation for Properties definition
  @Property(key = RedminePlugin.HOST,                  <-- Required attribute for every property
         name = "Redmine Host URL",
         description = "Example : http://demo.redmine.org/",
         defaultValue = "",
         global = true,                                <-- Available in global settings
         project = true,
         module = false),
  @Property(key = RedminePlugin.API_ACCESS_KEY,
         name = "API Access Key",
         description = "You can find your API key on your
                        account page ( /my/account ) when logged in,
                        on the right-hand
                        pane of the default layout.",
         type = org.sonar.api.PropertyType.PASSWORD,   <-- Type definition
         global = true,
         project = true,
         module = false),
  @Property(key = RedminePlugin.PROJECT_KEY,
         name = "Project Key",
         global = false,
         project = true,
         module = false)
})

public final class RedminePlugin extends SonarPlugin {   <-- Extends base class

public static final String HOST = "sonar.redmine.host";
public static final String API_ACCESS_KEY = "sonar.redmine.api-access-key";
public static final String PROJECT_KEY = "sonar.redmine.project-key";

public List getExtensions() {                          <-- Returns all plugin extensions
   return ImmutableList.of(RedmineMetrics.class,RedmineSensor.class,
                RedmineWidget.class);
}
}
```

There's a lot to take in here—more than can be adequately covered in a few code annotations, so we'll walk through it field by field. First, two annotations in the SonarQube API can be used to specify properties (`@Properties` and `@Property`). The former is used to define an array of properties (you can skip it if you only have one), and the latter provides attributes to fully describe a property. The required attributes for the `@Property` annotation are `key` and `name`. We'll look at the rest of its attributes shortly.

Each property should have a unique key, so it's a good idea to follow some naming conventions: we started with the word *sonar* followed by the plugin key and then the

property name. The `name` and `description` attributes are used by the SonarQube UI. Use `name` to briefly describe the property, and add some explanation—if necessary—in the `description` attribute. If there's no user input, you have the option of setting a default value with the `defaultValue` attribute.

By default, all properties are `String`s, but you can change a property's type with the `type` attribute, which expects a value from the `PropertyType Enum`. Several options are available, such as `Integer`, `Boolean`, `Regular Expression`, `Multi-line text`, or even `password` to mask the input value with asterisks. For a complete (and up-to-date) list of the available property types, check the relevant API documentation.

> **NOTE** In the following pages we'll talk a lot about SonarQube's API. We recommend that you always look at the latest online documentation (http://docs.sonarsource.org/latest/apidocs/) before you use a provided class or interface.

Two of the properties in listing 16.1 use the default `type`, whereas the `API access key` property is set to `PASSWORD` because you don't want its value to be visible in the SonarQube UI.

Finally, you need to tell SonarQube at which levels the property can be set, using the `global`, `project`, and `module` attributes, which accept Boolean values. If you don't specify any of these attributes, then by default the property will be available only at a global level, because `global` is true by default and the other two attributes default to `false`. In listing 16.1, the `host` and `API access key` properties are settable at the global level and overridable at the project level. Nothing is set at the module level, and the project key can only be set at the project level, not globally.

Now that we've covered the properties, we can move to the code of the `RedminePlugin` class. Each plugin should have one class that extends the base `SonarPlugin` class so that SonarQube will recognize and load your plugin. In this case, it's the `RedminePlugin` class, and for that reason it implements the abstract method `getExtensions`. This method should return a list (preferably immutable) of all extension points implemented by the plugin.

Extension points

We didn't mention this earlier, but now it's the right time: a SonarQube plugin is a set of classes that extend other classes or implement interfaces provided by the SonarQube API. We refer to them as *extension points*, and you can get a complete list of the available extension points by visiting the relevant API documentation.

Back to the `RedminePlugin` class, notice that the implementation of the `getExtensions` method returns a list of all classes found in the plugin (except of course the `RedminePlugin` class itself). This enables SonarQube to register these extension points and execute their code when needed (during analysis, when showing the widget, and so on).

At this point, the configuration and plugin description class is complete, so you can move to the next step: specifying the metrics the plugin will compute and store in SonarQube's database.

16.2.3 Describing the metrics: what you'll calculate and store

According to the requirements, the plugin should collect a count of all open issues for a given project and display the counts by priority. So you need a couple of metrics: one to keep the total number of open issues, and one to keep the number of open issues for each priority. Although these metrics can be combined as one, for clarity the responsible `RedmineMetrics` class in the next listing has two definitions (one for each metric).

Listing 16.2 `RedmineMetrics` class: metrics definition

```
public final class RedmineMetrics implements Metrics {

public static final String ISSUES_DOMAIN = "Issues";
public static final String ISSUES_KEY = "redmine-issues";
public static final String ISSUES_BY_PRIORITY_KEY =
   "redmine-issues-by-priority";

public static final Metric ISSUES =
   new Metric.Builder(ISSUES_KEY, "Redmine Issues", Metric.ValueType.INT)
     .setDescription("Number of Redmine Issues")
     .setDirection(Metric.DIRECTION_NONE)
     .setQualitative(false)
     .setDomain(ISSUES_DOMAIN).create();

public static final Metric ISSUES_BY_PRIORITY =
   new Metric.Builder(ISSUES_BY_PRIORITY_KEY, "Redmine Issues by priority",
       Metric.ValueType.DATA)
      .setDescription("Number of Redmine issues by priority")
      .setQualitative(false)
      .setDomain(ISSUES_DOMAIN).create();

public List<Metric> getMetrics() { #Implementation of abstract method
  return ImmutableList.of(ISSUES, ISSUES_BY_PRIORITY);
}
}
```

Implements abstract Metrics class

Describes the "total number of issues" metric

Each metric has three required fields (key, name, value type)

Describes the "issues by priority" metric

It's a good practice to categorize metrics in domains

Looking at the class definition, you see that it implements the abstract `Metrics` class, which is another extension point provided by SonarQube. Then, each metric is described as a static final variable of type `Metric` using the Builder design pattern (www.oodesign.com/builder-pattern.html). You're required to provide three attributes for every metric: a key, which should be unique (consider including the plugin key as part of the metric key to ensure uniqueness); a short name; and the value type.

The value type is critical because it tells SonarQube what kind of data is stored in the database. The possible choices are straightforward and can be found in SonarQube's API documentation. But keep in mind that if you want to store large amounts of data, such as the example's distribution of issues, you should use the `Data` value

type. Wondering why you don't define a metric per priority? Although this approach would work, it would make your code much more complex and make it difficult to handle and display your metrics in a widget. The `Data` value type was created by SonarQube for that reason: to avoid creating several metrics just to describe a distribution of them.

For the complete list of the available metric attributes, you can browse the latest online API documentation, but we'd like to give you some useful tips:

- A metric's *domain* determines how it's grouped in drop-downs in the SonarQube UI. You can make up your own value to put your metrics together in their own group, or use an existing value such as `Complexity`, `Documentation`, or `Rules` to add your metrics to one of those groupings.
- When you're dealing with metrics where trending is important, be sure to set the `direction` attribute so SonarQube knows if an increase in this value is good or bad. Then you can use the `quantitative` attribute (if set to `true`) to tell SonarQube to colorize (green/red) the metric's trending icons in the UI.
- If you want to hide resources from the drilldown view when they've reached the best possible value for a metric, set the metric's `bestValue` attribute.

The last thing you need to do in a `Metrics` class is implement the `getMetrics` method to return a list of the plugin's metrics.

As you'll see in next section, there are two kind of metrics: those that hold a single value and those that hold a collection of values, which is called a *distribution*. We'll show you how to compute and store both of these types in the database.

The basic configuration of the plugin is now ready. The next step is to compute the measures. In other words, you need to write the code that will perform the core functionality of your plugin.

16.2.4 Implementing your analyzer with a sensor

As we already told you, the hard work is done by analyzers, which are responsible for computing measures. There are two types: *sensors* and *decorators*. The logical difference is that sensors are executed only *once* during the analysis phase. They're responsible for producing measures by parsing files (for instance, code-coverage result files), communicating with external services such as the Redmine server in the example, or even invoking a Maven plugin. On the other hand, decorators are triggered when all sensors have finished their tasks. The `decorate` method is executed for every resource of a given level following a bottom-up approach. Resources in SonarQube are considered the following: block units (methods, functions, and so on), files, directories (packages), program units (classes, interfaces) or projects (projects, modules, and so forth).

Because decorators are executed after sensors, they have access to the metrics calculated by sensors and can also create new metrics based on them. Normally, a decorator is used to aggregate at higher levels the measures computed by sensors at lower levels. Table 16.2 summarizes the differences between sensors and decorators.

Table 16.2 Differences between sensors and decorators

Characteristic	Sensor	Decorator
Analysis	Triggered only once at the project or submodule level	Triggered several times, once for every resource.
Timing	Can be executed in random order	Executed only when all sensors have finished. You can specify the order of execution between decorators.
Metrics / Usage	Used to feed SonarQube with new measures	Used to aggregate already-computed measures on a higher level.

You know the difference, and it's time to decide. What kind of analyzer do you need: sensor or decorator? A complicated plugin can be composed of several sensors and decorators, so you might say, "Both." Fortunately, in this example things are simpler. You want a component to access a Redmine server's API, retrieve issues, and store them, and you want that to happen only once per analysis, not once per file (for example). Thus a sensor is what you're looking for.

Before we continue, we recommended that you delete the `RedmineDecorator` class from the auto-generated project, because you won't need it. Later we'll discuss an example of a decorator out of the Redmine plugin context.

Listing 16.3 contains the basic code of the Redmine plugin. This is the code triggered by SonarQube during the analysis phase. Once again, the `RedmineSensor` class implements the appropriate base class (sensor) and two abstract methods (`shouldExecuteOnProject` and `analyze`).

Listing 16.3 `RedmineSensor` class: analyzer implementation

```
public class RedmineSensor implements Sensor {

private final Settings settings;
private final RedmineAdapter redmineAdapter;

public RedmineSensor (Settings settings,
                      RedmineAdapter redmineAdapter) {
  this.settings = settings;
  this.redmineAdapter = redmineAdapter;
}

public boolean shouldExecuteOnProject
                   (Project project) {
  if (missingMandatoryParameters()) {
    LOG.info("Redmine issues sensor will not run as some parameters are
     missing.");
  }
  return project.isRoot() && !missingMandatoryParameters();
}
```

Implements abstract sensor

Local variables to hold injected components

Declares needed components in constructor. Components will be injected by SonarQube's IoC container.

Implements abstract method that checks if sensor should be executed in current project

```
public void analyse(Project project, SensorContext context) {
  try {
    Map<String, Integer> issuesByPriority =
            redmineAdapter.collectProjectIssuesByPriority(
            getHost(), getApiAccessKey(), getProjectKey());
    double totalIssues = 0;
    PropertiesBuilder<String, Integer> distribution = new
                                  PropertiesBuilder<String, Integer>();
    for (Map.Entry<String, Integer> entry : issuesByPriority.entrySet()) {
      totalIssues += entry.getValue();
      distribution.add(entry.getKey(),
                       entry.getValue());
    }

    saveMeasures(context, totalIssues, distribution.buildData());
  } catch (RedmineException ex) {
    LOG.error("Redmine issues sensor failed to get project issues.", ex);
  }
}

protected void saveMeasures(
                 SensorContext context, double
                 totalPrioritiesCount,
                 String priorityDistribution) {

  context.saveMeasure(new Measure(
                      RedmineMetrics.ISSUES,

                      totalPrioritiesCount));
  context.saveMeasure(new Measure(RedmineMetrics.ISSUES_BY_PRIORITY,
                      priorityDistribution));
} }
}
```

Distribution metric

Basic (double) metric

Implements abstract method that does the hard work

Transforms Map to PropertiesBuilder datatype provided by SonarQube and used to store data value types to database

Method to persist computed measures to database

SensorContext, responsible among others to handle measures

The first method (`shouldExecuteOnProject`) is responsible for allowing the plugin to run or not based on its output. In the example, the sensor analyzes the project (which is passed as a method parameter) only if it's a root project (and not a submodule) and if all required parameters have been set (see the `missingMandatoryParameters` protected method). Keep in mind that a *project* in SonarQube is either the root project or any of its submodules. If you don't check for the project level, then the sensor will run for the root project and all submodules. Let's consider a couple more examples to help you understand the purpose of this method. If you want to trigger the sensor for all analyses and then return just a `true` value—or, for instance, if you want to enable it only for Java projects—you should consider writing something like `return Java.INSTANCE.equals(project.getLanguage());`.

In general, keep in mind, that the `shouldExecuteOnProject` method should check for the existence of required settings and/or other project attributes (level, language) to turn on the green light for execution.

The `analyze` method is where all the great things happen. Here, you write the meat of your plugin: you get information from external resources, compute metrics, and save them in SonarQube. You have available as input parameters the project currently being

analyzed and the context of the sensor. This context is used to store computed metrics in the database, add issues on source/test files, and so on.

As we told you, there are two kinds of metrics (single values and distributions), and the example deals with both of them. Before you compute the metrics, you need to get access to the data, which consists of the Redmine issues. To do so, by invoking the `collectProjectIssuesByPriority` method of the `RedmineAdapter` class, you get a reference to a map (key of `String` type, value of `Integer` type) that contains the number of open issues for each priority. You might wonder who created the `RedmineAdapter` object. Hold you breath; we'll get back to that in a minute.

Having access to this map, you can iterate it and calculate the total number of open issues by simple addition. You may think that calculating and storing the distribution metric of open issues by priority would be done the same way. Well, that's almost true. The difference is that you need to transform this map to `String` or `Binary` type (for a large amount of data) before you store it in the database. You can't store the map that's returned from the `RedmineAdapter` class. The `PropertiesBuilder` class is the gift of the SonarQube API for that transformation. You just need to fill it accordingly when you iterate through the map. Then you can use the `buildData` method to transform it to a readable `String`—for instance, `"Normal=2;Urgent=1"`—or use the `build` method to transform it to a `Measure` object that contains the same data in binary format.

To see how you can store these metrics in the database, look at the `saveMeasures` method. It uses the `SensorContext` reference by invoking the `saveMeasure` method. You can either pass as a parameter the object produced by the `PropertiesBuilder#build` method or create a new `Measure` object. The `Measure` constructor takes two arguments: the unique key of the metric and the computed value. The rest is automatically done by the SonarQube API.

With about 20 lines of code, you've written the most important part of the Redmine plugin, which computes and stores two metrics in SonarQube! The next step is to create a widget to show these metrics in a dashboard. But before we show you how to do that, we'll explain a couple of dark spots we haven't covered yet in the `RedmineSensor` class.

Dependency Injection in SonarQube plugins

In several places, the code uses the `settings` and `RedmineAdapter` variables, which are passed to the class as constructor parameters. But you may be wondering how those parameters are created.

Here comes the beauty of Dependency Injection in SonarQube. We told you before that it's a good practice for every class you create in a plugin to extend an extension point and be included in the list of extensions returned by the `getExtensions` method of the class in the plugin that extends the `SonarPlugin` class. All classes that do so are automatically registered with SonarQube's Inversion of Control (IoC) container, for constructor injection. That means you can pass any component you want to use in the `class` constructor, and SonarQube will automatically inject it (create it for you).

NOTE If you're not familiar with Inversion of Control and Dependency Injection, check out Martin Fowler's excellent article on the subject at http://martinfowler.com/articles/injection.html, or refer to the book *Dependency Injection* by Dhanji R. Prasanna (Manning, 2009: www.manning.com/prasanna/).

Now you can specify any registered extension as a constructor parameter and use it as if you had manually initialized it. The great thing is, as shown in listing 16.3, you can pass objects not only from your own plugin (`RedmineAdapter`), but from SonarQube core (`Settings`) as well.

Dependency Injection in SonarQube

A final note about Dependency Injection in SonarQube: it's a very bad technique, although programmatically allowed, to manually instantiate classes that implement (directly or indirectly) the `BatchComponent` or `ServerComponent` class, because those classes should be automatically instantiated via IoC. If you do, your plugin might work today, but nobody guarantees that it will work under future versions of SonarQube. To avoid this kind of potential problem, and to make your life easier, let the IoC container handle the creation/destruction of the classes you need.

You're almost finished. You've created a plugin, declared its properties and metrics, and written the code to create and store those metrics. If you only wanted to show your metric values in filter columns, you'd be done. But you probably also want to make your metrics available on dashboards, so there's one last step: the design of a dashboard widget. In the next section, we'll teach you how to do it.

16.2.5 Creating your first widget

A widget is composed of two source files: a Java class that—what a surprise—extends another basic class and needs to be included in the plugin's extension points, and a Ruby file that is the actual representation of the widget. Widgets are typically used to show in a nice way the metrics computed by the plugin and allow interaction with end users. The following listing shows the code for the `RedmineWidget` class.

Listing 16.4 `RedmineWidget` class: describing the widget

```
@WidgetCategory("Redmine")
public class RedmineWidget extends AbstractRubyTemplate implements
     RubyRailsWidget {

public String getId() {
  return "redmine";
}

public String getTitle() {
  return "Redmine Widget";
}

@Override
```

```
protected String getTemplatePath() {
  return "/redmine/redmine_widget.html.erb";
}
}
```

As you probably notice, `RedmineWidget` implements `RubyRailsWidget` to override the `getTemplatePath` method and provide its own Ruby file reference. The `getTitle` method is used in the SonarQube Web UI when adding the widget to a dashboard; but as you'll see in the next section, you can change this when localizing the plugin. Finally, the `getId` method provides a unique widget ID, so it's a good practice to include in the ID the name of the plugin to avoid duplications with other widgets.

The @WidgetCategory annotation

Before we move on to the Ruby code, let's explain the `@WidgetCategory` annotation. It's used to group the plugins in the specified category during dashboard editing. It's optional, but if you don't use the annotation, then the widget will only be found under the None category (or by browsing the entire, uncategorized list).

Now it's time to design the widget, so refresh your HTML and Ruby skills. The widget you'll create will be split in two columns. On the left it'll show the total number of open issues, and on the right will be a list of open issues by priority. This layout might be familiar, because it's the general layout of the issues widget we discussed in chapter 2.

Listing 16.5 redmine_widget.html.erb

```
<%
  issues_measure = measure('redmine-issues')
  issues_by_priority_measure = measure('redmine-issues-by-priority')
  if issues_measure
%>
<table width="100%">
  <tbody>
    <tr>
      <td align="left">
        <div class="dashbox" >
          <p class="title"><%= message('widget.redmine.name') -%></p>
          <p><span class="big">
        <%= format_measure(issues_measure) -%></span></p>
        </div>
      </td>
      <td valign="top" align="left">
        <div class="dashbox">
          <p class="title">
            <%= message('widget.redmine.by_priority') -%></p>
            <table>
            <%
              priorities =

    issues_by_priority_measure.text_value.split(";")
              priorities.each do |priority_with_issues|
```

Gets measure by its key

Localization 1

Displays formatted value 2

1 Localization

3 Splits values

```
                priority_with_issues_array =
                    priority_with_issues.split("=")
            %>
                <tr>
                  <td align="left">
                    <%= priority_with_issues_array[0] -%>
                  </td>
                  <td align="right" style="padding-left: 10px;">
                    <%= priority_with_issues_array[1] -%>
                  </td>
                </tr>
              <% end %>
            </table>
          </div>
        </td>
      </tr>
    </tbody>
  </table>
<% end %>
```

Displays cell 4

3 Splits values

Listing 16.5 is a mixture of HTML with Ruby templating (ERB) included between the `<%` and `%>` symbols. We won't explain the HTML formatting of the plugin but will instead focus on some Ruby functions provided by SonarQube that you can use in your widgets.

The first thing you want to do in a widget is get the value of a metric, which is easy thanks to the SonarQube Ruby function `measure`. It takes as a parameter the metric ID discussed in section 16.2.3 and returns its value. To display the value in a formatted way, you can use the `format_measure` function 2. You'll probably want to use this function because it automatically formats the measure based on its value, and you don't have to worry about adding symbols such as percent signs or commas in the widget. That's all you need to get and display metrics from SonarQube's database.

But what about distribution metrics like the issues by priority you implemented in the Redmine plugin? You need to introduce another step before you show them—one that splits the metric. Remember that it's stored as a string in the following format: `"Normal=2;Urgent=1"`. You don't want end users to see something like that in your widget!

To split the values 3, you get the `text_value` of the metric and then call the `split` function by passing as a parameter the semicolon character (`;`). This creates an array of `Strings`: `[Normal=2][Urgent=1]`. You could display the content of each element; but you can also use the `split` function, this time passing an equal sign (`=`) as a parameter, to get an array of split values for each element. For example, splitting the first element, you get the following table: `[Normal][2]`. Splitting the second, you get `[Urgent][1]`. From this point on, you can display the appropriate cell 4.

And there you have it. Your widget is ready! But wait—we promised that your plugin would support localization. Where is it? 1: another Ruby function (`message`) that accepts a key as a parameter, finds its value (where?—we'll show you in a minute), and displays it at runtime. In the next section, we'll discuss how SonarQube supports localization and how you should name the keys in your localized bundles.

16.2.6 Supporting internationalization

SonarQube uses the i18n mechanism for platform and plugin localization. We're not going to cover how to create a language pack (a plugin that translates SonarQube core and other open source plugins), but if you're interested in creating one you can visit SonarQube's online documentation. In this section, we'll explain how to allow the localization of the Redmine plugin you just finished.

The first thing you need to do is to create a properties file like the one shown in listing 16.6. It's a simple text file containing keys and their values. This file must be placed under the src/main/resources project directory in a package named `org.sonar.i18n`, and its name should follow the pattern <*pluginKey*>.properties. In the case of the Redmine plugin, because its key is `redmine` (if you don't remember when you gave this name, go back to section 16.2.1, where you initialized your plugin's project), you should save the file as redmine.properties.

Listing 16.6 redmine.properties: localization file for the Redmine plugin

```
widget.redmine.name=Redmine issues
widget.redmine.description=Displays the number
                           of open issues of a given redmine project.
widget.redmine.redmine_issues=Redmine Issues
widget.redmine.by_priority=By Priority

metric.redmine-issues.name=Redmine Open Issues
metric.redmine-issues.description=Number of Open Redmine Issues
metric.redmine-issues-by-priority.name=Redmine Open Issues by Priority
metric.redmine-issues-by-priority.description=
                       Number of Open Redmine Issues grouped by Priority
```

You can use the localized messages in widgets or other Ruby on Rails pages by invoking the overloaded `message` Ruby function. The simplest form of execution requires only the message key. So, to display in your widget the message "By priority", you could write

```
message('widget.redmine.by_priority')
```

Or you could provide a default value, in case the given property key isn't found in the file:

```
message('widget.redmine.by_priority',:default=>'By Priority')
```

What we've shown you so far is all you need to know about i18n for widgets. But on the server side—during analysis—there's a restriction: you can only take advantage of the localized messages in components that extend the `ServerExtension` base class. Classes that directly or indirectly extend from `BatchExtension`, such as the `RedmineSensor` class, can't use these messages for logging or in the rare case where the metric may need to be internationalized before being stored in the database. To get the value of a property key in a server component, you need to pass the `org.sonar.api.i18n.I18n` component in its constructor to be injected by the IoC container discussed in

section 16.2.4. Then you can write something like the following in your code (for instance, to get the value of the `redmine.issue_subject` property):

```
: i18n.message(Locale.getDefault(),"redmine.issue_subject",null,
  new Object[0])
```

The latest version of the Redmine plugin hosted in GitHub (https://github.com/SonarCommunity/sonar-redmine) adds integration with SonarQube issues. A class named `RedmineLinkFunction` extends `ServerExtension` and uses what we discussed in the previous paragraph, so you can see a real-world scenario of using localized messages in Java.

There's one more thing to note about localizing messages. You probably noticed that the property keys follow some conventions. For instance, to localize the name/description of the metrics defined in section 16.2.3, you need to follow the conventions `metric.<key>.name` and `metric.<key>.description` and include in your properties file pairs of keys for each metric, as shown in listing 16.6. The same or similar idea applies to the widget key/description, property key/description, and other SonarQube concepts such as issues and rules. For a complete and updated list of these conventions, we recommend that you visit SonarQube's wiki at http://mng.bz/Ufcv.

This completes our discussion of writing a SonarQube plugin. We covered most of the stuff you need to be aware of to start implementing your own brilliant ideas for extending SonarQube's functionality. But before we close, there's a little more to see. We promised we'd give you an example of a decorator; and after that we'll discuss some concepts related to supporting new languages.

16.2.7 A decorator example

We mentioned earlier in this chapter that a decorator, in SonarQube terms, is a component that applies a `decorate` function to all resources. It's your responsibility to check the level of resources in this method and perform tasks. In this section, we'll show you an example of such a decorator.

The source code, shown in listing 16.7, is taken from the open source Abacus plugin (http://docs.codehaus.org/display/SONAR/Abacus+Plugin), which reads the complexity value of the given resource (as calculated by SonarQube), computes the so-called Abacus Complexity, and stores it in the database as a new metric.

Listing 16.7 `AbacusDecorator`: example of simple decorator

```
public class AbacusDecorator implements Decorator {

  @DependsUpon
  public List<Metric> dependsOn() {
    return Arrays.asList(CoreMetrics.FILE_COMPLEXITY);
  }

  public boolean shouldExecuteOnProject(Project project) {
    return true;
  }
```

```
  public void decorate(Resource rsrc, DecoratorContext dc) {
    computeAbacusComplexity(rsrc, dc);
  }

  private void computeAbacusComplexity(Resource rsrc, DecoratorContext dc) {
    if (ResourceUtils.isFile(rsrc) ||
        ResourceUtils.isPackage(rsrc) ||
        ResourceUtils.isDirectory(rsrc) ||
        ResourceUtils.isRootProject(rsrc)          ||
        ResourceUtils.isModuleProject(rsrc)) {
      Double fileComplexity =
      MeasureUtils.getValue(dc.getMeasure(CoreMetrics.FILE_COMPLEXITY),
                Double.NaN);
      if (!Double.isNaN(fileComplexity)) {
        dc.saveMeasure(new Measure(AbacusMetrics.ABACUS_COMPLEXITY,
        ComplexityThresholdsUtils.
         convertCyclomaticComplexityToAbacusComplexity(
               fileComplexity, complexityThresholds)));
      }
    }
  }

}
```

The class implements the decorator interface and its two required methods. The first method, `shouldExecuteOnProject`, is just like the sensor method with the same name. The second, `decorate`, is to a decorator what the `analyze` method is to a sensor. That means it's where the real code goes.

The *huge* difference between `decorate` and `analyze` is that the former takes a `Resource` parameter, and not the `Project` itself; so as we told you, you need to do some additional work to check whether you want to run the decorator's code for the given resource. To make it even clearer, if the analyzed project has a total of 120 resources, then SonarQube will trigger the `decorate` method 120 times: once for each resource with that resource passed as an incoming parameter.

Also notice the annotated method at the beginning of the class. When you want to use another metric calculated by SonarQube core or other plugins, you should declare it in an annotated (`@DependsUpon`) method. Don't worry about the method name—you can choose anything you like. The signature of this method may return a list of `Metric`s instead of a single one. For instance, you can write something like the following:

```
@DependsUpon
public List<Metric> dependsOnCoreMetrics() {
  return ImmutableList.of(CoreMetrics.COMPLEXITY,CoreMetrics.COVERAGE);
}
```

This means the decorator depends on the `Complexity` and `Coverage` metrics already computed by other sensor(s).

Now let's examine the core of the decorator: the `decorate` method. The `context` parameter, just as in sensor classes, is responsible for handling measures. If you want to refer to the value of a metric listed in any of the `@DependsUpon` annotated methods

(you may have more than one methods annotated with `@DependsUpon`), invoke the `DecoratorContext.getMeasure` method.

What we've discussed about saving measures in sensors applies to decorators as well. Keep in mind that the context changes every time the `decorate` method is triggered. So, expect to get a different measure value each time—the value that corresponds to the *current* resource. If the metric isn't calculated for the resource, such as when the metric you're calculating is the average complexity per package but the resource you're on is a file, the measure is `null`.

Finally let's look at how you can check the resource's level to control the flow of your decorator's code. SonarQube provides a `ResourceUtils` class with static methods that test a resource against a level and return a Boolean. We suggest that you visit the latest SonarQube online Javadoc to read more about the method definitions and how you can use the `ResourceUtils` class.

So far, we've covered most of the basic and some advanced concepts related to SonarQube plugin development. As we've mentioned, plugins can be also used to add support for new languages. Next we'll give you an overview of how you can start writing your own language plugin.

16.3 Adding support for new programming languages

Most of the basic concepts we've discussed apply to language plugins. To quickly begin writing, you can clone an existing plugin or create a new one from the Maven archetype we presented earlier. But this time, you need to make several modifications in the generated code.

Before we show you the basics, let's consider the reasons you might want to write such a plugin and what it offers. At the time this book was written, SonarQube supported about 20 programming languages of the several hundred available. We know that most of them are known by very few people, but plenty of them are popular though not covered by SonarQube. Imagine that you're using one of those languages. Too bad your projects can't take advantage of all the brilliant SonarQube features discussed in this book. So why don't you try to write your own language plugin? As you'll discover in this section, a significant part of the work you need to do has already been implemented by SonarSource.

But you'll still need to write some serious code for the plugin. Not all plugins do the same things: for instance, the open source C++ plugin doesn't compute measures about LCOM4 and RFC metrics. So, your first and most important decision should be which metrics your plugin will support. Then you may consider adding coding rules grouped in a default profile. After that, you might include other features such as unit-test execution, duplication recognition, and so on. Let's discuss the steps to start implementing such a plugin.

First you need a way to parse your target language: you need a "language recognizer." Open source libraries are available for the job, but the SonarQube toolkit offers the SonarSource Language Recognizer (SSLR: http://docs.codehaus.org/display/

Figure 16.3 SSLR-based language plugin (JavaScript) project structure

SONAR/SSLR), a lightweight library for analyzing source code. It offers a stable integration with SonarQube core and facilitates the development of language plugins. Many of the most popular language plugins (JavaScript, COBOL, C#, Python, PL/SQL, C, C++, Flex, and so on) already use SSLR, so we strongly recommend it and will give you some insights in this section.

Typically, a language plugin based on SSLR is a multi-module Maven project. Figure 16.3 shows the structure of the JavaScript plugin, which is composed of four modules:

- `JavaScript :: Sonar Plugin`—The plugin module as discussed in previous sections. You need to provide code, apart from defining the extensions, for the language quality profile, the rule repository, the source importer, and basic metrics such as coverage, duplications, and so on.
- `JavaScript :: Checks`—Library that includes code for checking the code (using the SSLR, of course) against rules.
- `JavaScript :: Squid`—The core of the plugin. Here you can find the code for parsing and analyzing source code. Language grammar, as well as keywords, metrics, and punctuation definitions, are also placed in this module.
- `JavaScript :: DevKit` : A development kit for testing and displaying all the available measures on your project

You can integrate the plugin with external tools (such as coverage tools to parse result files) just as you'd do with simpler plugins. You can write as many sensors or decorators as you need, and you can create as many widgets as you like to display the metrics you compute during analysis. Certainly the most time-consuming and difficult task is to describe the language using the SSLR and to parse the source code files. Unfortunately, that's beyond what we can cover in this book.

But we believe that if you know the basics—and you have surely learned more than the basics by reading this chapter—you'll be able to quickly start implementing a SonarQube plugin for your favorite programming language.

16.4 Summary

Well done, developers! You've learned many great things about extending SonarQube and building your own plugins. We started this chapter by illustrating how SonarQube works behind the scenes. We discussed its architecture and briefly explained the core components. Then you began to implement a fully working plugin for SonarQube

that integrates it with Redmine, a popular issue-tracking system. You've learned how to do the following:

- Quickly start a Maven SonarQube plugin project by using a Maven archetype.
- Implement a complete plugin that supports localization with basic and some advanced features.
- Choose between a sensor and a decorator. By now you should be able to identify their differences and when you should use each analyzer type.
- Write the analyzing code to compute the values of your metrics and store them in SonarQube's database.

Finally, we scratched the surface of writing a plugin to support a new programming language with an overview of the SSLR and the factors you need to keep in mind when implementing such a plugin. The truth is that fully explaining how to write a language plugin would probably take several chapters; maybe we'll include more content in the next edition of this book or—who knows?—in another book.

We hope you've enjoyed reading this book as much as we enjoyed writing it. We'd like to thank you for following us to the end. Best of luck with your first or advanced steps with SonarQube!

appendix A
Installation and setup

The purpose of this appendix is to get you up and running on the SonarQube platform. We'll assume that if you're here, it's because you're ready to give SonarQube a try, so we'll focus on practical advice and skip the persuasive writing.

Before you can get SonarQube up and running, there are a few prerequisites. You'll need to have Java installed where you want to run SonarQube, at least version 6, as well as a database.

Don't let the fact that SonarQube bundles-in the H2 database tempt you to skip this step. H2 is included as a courtesy for initial testing only and *is not* for long-term, production use. If you try to use H2 as your production database, we guarantee you'll be disappointed—not least because you *cannot upgrade* a SonarQube database that's stored in H2. We use MySQL because, much like SonarQube, it's free and open source; it's dependable; and if you decide down the road that you want paid support, it's available. But you can use any one of the supported databases: Oracle, Postgres, SQL Server, and of course MySQL.

Once you've checked off the basic requirements, you can move on to installing SonarQube itself. We'll start by walking you through verifying your Java installation and setting up the SonarQube database, then move on to installing SonarQube itself, and then finish with some advice on upgrades. For the SonarQube installation and upgrade steps, we'll look at Windows 7 and Ubuntu Linux.

A.1 Preparing for installation

First, let's deal with the prerequisites: Java and a database. We won't show you how to install either of these things—that's what the internet is for—but we can show you how to make sure you're ready to install SonarQube.

A.1.1 Verifying Java

To verify that Java is installed and ready, go to a command prompt and type the following command:

```
java -version
```

Depending on the version of Java you're using, the response will be something along these lines:

```
java version "1.7.0_09"
Java(TM) SE Runtime Environment (build 1.7.0_09-b04)
Java HotSpot(TM) Server VM (build 20.6-b01, mixed mode)
```

A.1.2 Database setup

Once you've installed a database, you need to create the schema and user that SonarQube will need. After these steps, SonarQube will handle all its own database management for you, but you need to get it bootstrapped first. The following instructions assume you're using MySQL.

Go to a command prompt (Windows 7 users: run `cmd.exe` to get to the command line), and type the following command:

```
mysql -u root -p
```

This command starts a session with the MySQL database. The `-u` means that the next thing coming up is the user name to log in as. The `-p` says you want to specify a password. After you press Enter, you're prompted for it. Enter the database's root password. You'll see a MySQL prompt, like so:

```
mysql>
```

You're nearly ready to create the SonarQube user and database, but first you need to pick a password for the SonarQube user you're about to make. It's a good idea to use something different than the root password. You don't want to use *sonar* as the password, either.

It's time to set up the SonarQube schema and user. The good folks at SonarSource make this particularly easy by providing the commands you need in a script format. We've reproduced it in listing A.1 for your convenience. As you enter these commands, make sure you end each one with a semicolon, and be sure each instance of the `sonar` user's password is enclosed in single quotes.

Listing A.1 Schema and user creation

Create schema

```
CREATE DATABASE sonar CHARACTER SET utf8 COLLATE utf8_general_ci;
CREATE USER 'sonar' IDENTIFIED BY '<password you picked>';
```

Create user

Grant permissions

```
GRANT ALL ON sonar.* TO 'sonar'@'%' IDENTIFIED BY '<password you picked>';
GRANT ALL ON sonar.* TO 'sonar'@'localhost' IDENTIFIED BY '<password you
    picked>';
FLUSH PRIVILEGES;
```

Finalize

After each line, MySQL should respond like this:

```
Query OK, 0 rows affected (0.00 sec)
```

When you get `Query OK` after the last line, type `exit` to quit the session. You've just created SonarQube's database, named `sonar`, and SonarQube's user, also named `sonar`. The last three lines give the `sonar` user permission to do whatever it wants in the `sonar` database.

Now you're ready to move on to the installation of SonarQube itself.

A.2 Installing SonarQube

Ideally, you'll run SonarQube as a service, whatever your platform. That way, it starts back up automatically after a reboot. We prefer running SonarQube standalone, so that's what we'll show you how to set up. The exact steps for setting up a service vary by platform, but the download is the same regardless of platform (which is the beauty of Java applications).

The following URL goes to the download page: www.sonarsource.org/downloads/. The versions' zip files are listed with the newest first—and the newest is always the one you want. The SonarSource folks release fairly frequently (several times a year), and almost every time the new version contains compelling new features. If you're running Windows 7, skip ahead to section A.2.2 for specific installation instructions. For Ubuntu, continue with the next section.

A.2.1 Ubuntu

The first thing you need to do is decide where to install the SonarQube server. We use /usr/share, and the scripts that follow reflect that. If you prefer a different parent directory, adjust accordingly.

Listing A.2 is an install script. We'll tell you what you're about to do and then give you the commands to do it:

1. Expand the SonarQube zip file into /usr/share in a subdirectory named for the zip file, including the version number.
2. Create a symbolic link to your expanded directory, named sonar (without the version number), in /usr/bin.
3. Create a link in /etc/init.d to the service startup script provided with SonarQube. (For this step, you need to know whether you're running a 32-bit or 64-bit version of the OS.)
4. Register your startup script with the OS.

Listing A.2 Ubuntu SonarQube installation

```
sudo unzip -d /usr/share sonar-<version>.zip
sudo ln -s /usr/share/sonar-<version> /usr/bin/sonar
sudo ln -s /usr/bin/sonar/bin/linux-x86-
  <32 / 64 depending on your OS>/sonar.sh /etc/init.d/sonar
sudo update-rc.d sonar defaults
```

Expand zip file

Create first link

Link to startup script

Register startup with OS

Once you've completed these steps, SonarQube is installed, but you still need to configure it. Skip ahead to section A.2.3 for those steps.

A.2.2 Windows 7

The first thing you need to do is decide where to install the SonarQube server. We typically put it right into the root of whatever drive is chosen, but that's more a religious decision than a technical one. Whatever you decide, it's best to create a parent directory named SonarQube and then expand the zip file you downloaded into a subdirectory with the version in the name. This will make life simpler at upgrade time, which comes fairly frequently.

Once you've expanded the zip, you can set up SonarQube as a startup service. The good folks at SonarSource have made this easy for you with an installer. Look in the bin directory of the expanded SonarQube zip, and choose the subdirectory that's correct for your operating system: windows-x86-32 for a 32-bit operating system or windows-x86-64 for a 64-bit version. In the subdirectory, right-click the InstallNTService.bat file and choose Run as Administrator to make SonarQube a startup service on the box.

Depending on how permissions are set up on your machine, you may get an error when you try to run the service setup script. If you do, look in sonar-<version>/logs at sonar.log. You'll probably see something like this toward the bottom of the file (although not at the very bottom):

```
Unable to create file in temporary directory, please check existence
of it and permissions:
     C:\Windows\system32\config\systemprofile\AppData\Local\Temp\
```

If so, open Explorer and try to navigate down the cited path. As you drill in, you'll eventually be told that you don't have permissions to access the folder you just tried to open and given an option to grant yourself permissions. Accept the option in the dialog, and keep drilling until you get as far down the path as the directories exist (when we tried it, we got the dialog twice). The last directory in the path (Temp) probably isn't there. Don't worry about creating it—now that the permissions are taken care of, SonarQube can handle that for you. Try Run as Administrator again on the install script provided with SonarQube, and you should be good to go.

Next you need to make a few configuration changes.

A.2.3 Configuring SonarQube: Windows 7 and Ubuntu

You're nearly ready to turn on SonarQube. There's just one more step: telling it where its database is.

Look in SonarQube's conf directory for the sonar.properties file, and open it in your favorite text editor. (If you're on Linux, you'll need to use sudo to open the file; because you used sudo to unzip the archive originally, root owns all the files.) You'll find that it's well documented internally, with clearly named properties and helpful comment blocks above each section. In case you're not used to the conventions used here, lines that start with a pound sign (#) are comments (or commented-out properties).

There are two formats for properties in this file: one where the key and the value are separated by a colon and whitespace doesn't seem to matter, and ones where the

key and value are separated by an equal sign. Whitespace does matter for these, so don't insert any spaces around an equal sign, like so:

```
sonar.sample.property1:  spaces_don't_matter
sonar.sample.property2=leaveSpacesOut
```

Spaces after the colon are OK

Remove whitespace before and after the property value

These two formats aren't interchangeable, so properties you see with colons initially should retain those colons.

To point SonarQube to the database you set up for it, start by looking for the line that begins with `sonar.jdbc.password`. The default value is `sonar`. Change it to the password you set up for the `sonar` database user earlier:

```
sonar.jdbc.password:       <password you picked>
```

Next, look for these two lines:

```
# Comment the following line to deactivate the default embedded database.
sonar.jdbc.url:          jdbc:h2:tcp://localhost:9092/sonar
```

Comment-out the second one by adding a pound sign to the beginning of the line.

Scroll down a little until you see this set of lines, and uncomment each one that starts with sonar.jdbc by removing the pound sign from the beginning of the line:

```
#----- MySQL 5.x/6.x
# Comment the embedded database and uncomment the following line to use MySQL
#sonar.jdbc.url:
jdbc:mysql://localhost:3306/sonar?useUnicode=true&characterEncoding=
utf8&rewriteBatchedStatements=true

# Optional properties
#sonar.jdbc.driverClassName:                com.mysql.jdbc.Driver
```

Be sure to correct the server name in the database access URL if it's not on the same box as SonarQube. At this point, you could save the file and turn on SonarQube if you wanted, but there are a few other configurations you might like to look at.

If you're behind a proxy, be sure to uncomment and configure your authentication properties in the Update Center section so you can use SonarQube's built-in plug-in installation and updating features:

```
#--------------------------------------------------------
# UPDATE CENTER
#--------------------------------------------------------

# The Update Center requires an internet connection to request http://
    update.sonarsource.org

# It is activated by default:
#sonar.updatecenter.activate=true

# HTTP proxy (default none)
#http.proxyHost=
#http.proxyPort=
```

```
# NT domain name if NTLM proxy is used
#http.auth.ntlm.domain=

# SOCKS proxy (default none)
#socksProxyHost=
#socksProxyPort=

# proxy authentication. The 2 following properties are used for HTTP and
     SOCKS proxies.
#http.proxyUser=
#http.proxyPassword=
```

Once you've finished your configurations, you're ready to start SonarQube.

A.2.4 Turning it on

It's time to start the SonarQube service. If you're not familiar with how to start a service on your platform, we'll tell you how.

This first time, it will take SonarQube a few minutes to start up—in addition to all the normal startup activities, it's also creating the tables and indexes it needs in the database. After a few minutes, you should be able to point your browser to http://localhost:9000 and see something like what's shown in figure A.1.

WINDOWS 7

You can start the service from the Services control panel (which you may have to Run as Administrator); or, if you prefer a command-line interface, open a privileged session by clicking the Start button and typing `cmd` without pressing Enter. This time, press Shift-Ctrl-Enter. You're asked if you want to "allow the program to make changes to this computer." Accept, and at the command line, type

```
sc start sonar
```

SonarQube should start. But depending on how permissions are set up on your machine, you may get an error when you try to start SonarQube for the first time. If that's the case, refer to section A.2.2.

UBUNTU

Enter the following in a terminal:

```
sudo service sonar start
```

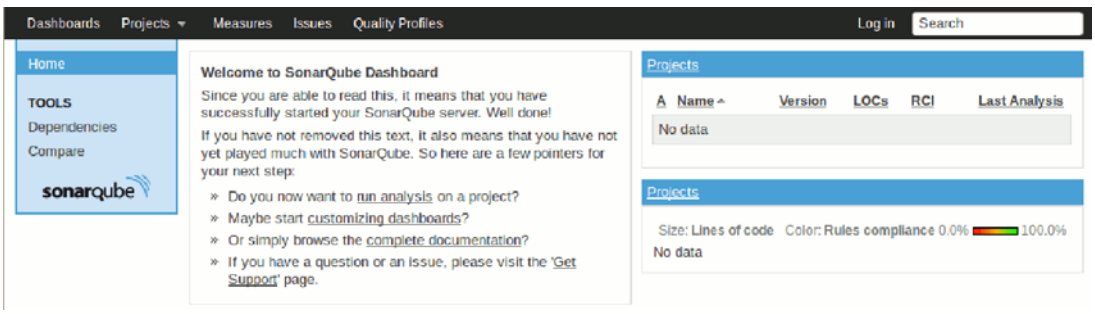


Figure A.1 Congratulations! SonarQube is up and running.

A.2.5 Default admin account

The final thing you need at this point is the credentials for the default admin account. You need these if you want to make any changes to SonarQube's default setup. You probably could have guessed, but the login is `admin` and the password is also `admin`.

Go ahead and log in now—use the link at upper right in the interface—to change the password. Once you're logged in, click the username at upper right (Administrator, in this case), and choose My Profile to open the password-change form.

A.3 Upgrading

You've just gotten SonarQube installed and up and running. Now seems terribly premature to talk about upgrades. But the SonarSource folks release fairly frequently—as often as eight times a year. And almost every release has compelling new functionality. So when new versions come out, you'll probably *want* to upgrade. Go to http://mng.bz/ahZm to sign up to be notified when new versions are available.

The steps for upgrading aren't onerous:

1. Read the upgrade guide, and download the new version. Expand the zip file into a new directory.
2. Copy your plugins (if any) to the new version.
3. Copy your configuration to the new version.
4. Stop SonarQube.
5. Back up your database.
6. Update your links, aliases, and service to point to the new version.
7. Start SonarQube.
8. Point a browser to http://[*yourSonarHost*]:9000/setup, and click the button.

As you can see, the steps themselves aren't difficult, but the list is a little long, and stepping through it can be tedious. Because laziness and impatience are among the chief virtues of a programmer (thank you, Larry Walls!) we've scripted the steps in an upgrade. Not only will these steps save you tedium and time, but they ensure consistency of process as well.

You'll still have to do a few things manually. Of course, you'll need to perform the download manually—the link is different every time. You should also consult the upgrade guide (it's linked from the email announcing a new version) in case there are any configuration tweaks you need to make for forward compatibility. After those two steps, though, the rest should be pretty painless. Continue to the next section for the Ubuntu version or to section A.3.2 for Windows 7.

A.3.1 Ubuntu

The Linux upgrade script relies on your having followed our installation guidelines. If you didn't, then this script won't work for you. Feel free to use it as a guide for writing your own upgrade script, but don't run it as-is.

If you did follow our advice, then put the script in listing A.3 into a clearly named file (say, sonarUpgrade.sh) in the same directory as the zip file of the new version, and make it executable (`chmod 755 sonarUpgrade.sh`).

It's ready to go, but you have to pass it two arguments: the version number you're upgrading *from* and the version number you're upgrading *to*.

Kicking it off looks something like this:

```
./sonarUpgrade.sh 3.4 3.5
```

If the script is able to verify that you have the 3.5 zip file waiting (or already expanded in the correct location) and that an install of 3.4 exists, it will start the upgrade. You're asked to enter two different passwords. The first is your sudo password. The script uses the sudo access of the person who invoked it to create directories in privileged locations, to move files around, and to stop and restart the SonarQube service.

The second password is the MySQL root password. That access is used to make a backup copy of your database, which is compressed (with gzip) and placed in the directory of the version you're upgrading from—the version it went with. If worse comes to worst (not that it should), you'll be able to restore SonarQube to the instant at which you started the upgrade.

When the script is finished, it instructs you to point a browser at http://*[yourSonarHost]*:9000/setup and follow the instructions on the screen. That screen presents a button. When you click it, SonarQube kicks off the database structure changes that accompany most upgrades. Depending on the size of your database, these can take a few minutes. When those changes are done, your browser will automatically refresh to SonarQube's front page. Here's the script.

Listing A.3 Ubuntu upgrade script

```
#!/bin/bash

old=$1
oldDir=/usr/share/sonar-${old}
new=$2
newDir=/usr/share/sonar-${new}

if [[ -z $old || -z $new ]]
then
  echo "usage: ${0} oldVer newVer"
  echo "E.g ${0} 2.9 2.10"
else
  if [[ ! -e $oldDir ]]
  then
    echo "Cannot upgrade from ${old}. Directory does not exist: ${oldDir}"
    exit
  else
    echo "Upgrade from sonar-${old} to sonar-${new}"

    if [[ ! -e "sonar-${new}.zip" ]]
    then
      echo "sonar-${new}.zip not found. Skipping expansion"
```

```
    else
      if [ ! -e /usr/share/sonar-${new} ]
      then
        echo "expanding sonar-${new}.zip to /usr/share"
        sudo unzip -d /usr/share sonar-${new}.zip

        echo "copying plugins"
        sudo cp $oldDir/extensions/plugins/*.jar $newDir/extensions/plugins/.

        echo "copying conf"
        sudo mv $newDir/conf/sonar.properties $newDir/conf/
     sonar.properties.bak
        sudo cp $oldDir/conf/sonar.properties $newDir/conf/.
      fi
    fi

    if [[ ! -e ${newDir} ]]
    then
      echo "Cannot complete upgrade. Directory for new version does not
     exist: ${newDir}"
    else
      echo "Stopping service"
      sudo service sonar stop

      echo "Updating symlink in /usr/bin"
      sudo rm /usr/bin/sonar
      sudo ln -s ${newDir} /usr/bin/sonar

      echo "Backing up database"
      d=`date +"%y%m%d"`
      mysqldump -u root -p sonar | gzip -9 > sonar-db-ver${old}-${d}.sql.gz
      sudo mv sonar-db-ver${old}-${d}.sql.gz ${oldDir}/.

      echo "Starting service"
      sudo service sonar start

      echo ""
      echo "To complete upgrade, access the following in a browser and follow
     instructions:"
      echo "http://$(hostname):9000/setup"
      echo "Once the front page renders, run a trial anaysis"
    fi
  fi
fi
```

A.3.2 Windows 7

The Windows upgrade script relies on your having followed our installation guidelines earlier. If you didn't, then this script won't work for you. Feel free to use it as a guide for writing your own upgrade script, but don't run it as-is.

If you did follow our advice, then put the script in listing A.4 into a clearly named file with a .bat extension under the SonarQube root directory, say sonarUpgrade.bat. When you're ready to use it, download the new version and unzip it into a subdirectory of the SonarQube root named sonar-<*version*>. Your directory listing should look something like this:

- sonar-<*oldVersion*>
- sonar-<*newVersion*>
- sonarUpgrade.bat

Now you're ready to kick off the upgrade script, but you can't just double-click it. You need to pass it two arguments: the version number you're upgrading *from* and the version number you're upgrading *to*.

To do so, open a privileged command-line session by clicking the Start button, typing `cmd`, and then pressing Shift-Ctrl-Enter. You're asked if you "want to allow the following program to make changes to this computer." Say Yes. At the command line, `cd` to your SonarQube root directory and then start the script:

```
cd C:\Sonar
sonarUpgrade 3.4 3.5
```

If the script is able to verify that the directories exist for the old and new versions, it starts the upgrade. Part way through, you're asked to enter the MySQL root password. That access is used to make a backup copy of your database, which is placed in the directory of the version you're upgrading from—the version it went with. If worse comes to worst (not that it should), you'll be able to restore SonarQube to the instant at which you started the upgrade.

When the script is finished, it instructs you to point a browser at http://[*yourSonarHost*]:9000/setup and follow the instructions on the screen. That screen presents a button. When you click it, SonarQube kicks off the database structure changes that accompany most upgrades. Depending on the size of your database, these can take a few minutes. When those changes are done, your browser will automatically refresh to SonarQube's front page.

You may need to make one small adjustment to the script; it assumes you're running 64-bit Windows. If you're using 32-bit instead, then look for this line about halfway down the script:

```
call %new%\bin\windows-x86-64\InstallNTService
```

And change the 64 to a 32, like so:

```
call %new%\bin\windows-x86-32\InstallNTService
```

Here's the script.

Listing A.4 Windows upgrade script

```
@echo off
if "%1"=="" goto :USAGE
if "%2"=="" goto :USAGE

set old=sonar-%1
set new=sonar-%2

if not exist %old% goto :1NOTEXIST
if not exist %new% goto :2NOTEXIST
```

```
echo Upgrade from %old% to %new%
echo.

echo Copying conf
move %new%\conf\sonar.properties %new%\conf\sonar.properties.bak
copy %old%\conf\sonar.properties %new%\conf /Y
echo Copying plugins
copy %old%\extensions\plugins\*.jar %new%\extensions\plugins /Y
echo.

echo Stopping service
sc stop sonar
sc delete sonar
echo.

echo Backing up database
set date=%date:~4,2%%date:~7,2%%date:~10,4%
set outfile=%old%\sonar-db-ver%1-%date%.sql
mysqldump -u root -p sonar > %outfile%
echo.

echo Starting service
call %new%\bin\windows-x86-64\InstallNTService
sc start sonar
echo.

echo To complete upgrade, access the following in a browser and follow
     instructions:
echo http://%hostname:9000/setup

goto :DONE
:2NOTEXIST
echo Cannot upgrade to %2. Directory does not exist
goto :DONE
:1NOTEXIST
echo Cannot upgrade from %1. Directory does not exist
goto :DONE
:USAGE
echo "Usage: %0 oldVersion newVersion"
:DONE
```

appendix B
Analysis

In this appendix, we'll show you how to run a SonarQube analysis via three different methods: SonarQube Runner, Maven, and Ant. Each of the first three sections stands alone, so feel free to skip to the one that interests you the most. For SonarQube Runner analysis, continue to section B.1. For Maven, skip to B.2, and for Ant you want B.3. If you're not sure which one to use, then you probably want SonarQube Runner.

> **NOTE** In each analysis section, we'll show you configurations that use localhost in URLs that refer to the SonarQube server or database. If you're not running your analyses on the SonarQube server itself, you'll need to replace every instance of localhost with your SonarQube server's hostname.

When you're done with the analysis section of interest, be sure to check section B.4 for a full listing of properties you can use to modify the execution of any type of analysis.

B.1 Analyzing with SonarQube Runner

SonarQube Runner is a Java application you fire from the command line. You feed it your project and a simple set of properties, and with those two things it can run the analysis for any language SonarQube handles (Java, C, C++, C#, ABAP, COBOL and so on).

Before you can analyze with the SonarQube Runner, you need to download it: http://docs.codehaus.org/x/N4KxDQ. The best place to go for the download is the SonarQube wiki, which lists all the available versions, newest first. Pretty much without exception, the newest will be the one you want.

You also need to make sure you have Java installed—because the SonarQube Runner was written in Java, you need to have Java on your machine to run it.

With Java on board, you're ready to install SonarQube Runner. Continue to section B.1.1 if you're on Windows, and skip to section B.1.2 if you're using Ubuntu.

B.1.1 Install: Windows 7

Once you've downloaded the SonarQube Runner zip file, expand it into the location of your choice. When you do so, by default the name of the directory that's created contains the version number. Leaving the version number in that directory name is a good idea because it will make things clearer at upgrade time.

Now create an environment variable called `SONAR_RUNNER_HOME` that points to your sonar-runner-<*version*> directory. You also need to make sure you have a `JAVA_HOME` environment variable that points to your Java directory. If Java was preinstalled for you, you may have a Java directory under Program Files. This isn't the directory you want to point your `JAVA_HOME` variable at. Open C:\Program Files\Java, and find the subdirectory named something like jre6. The full path to that subdirectory is what should be in `JAVA_HOME`.

After you set up `SONAR_RUNNER_HOME` and `JAVA_HOME`, add the following to the end of your `PATH` environment variable, and restart:

```
;%SONAR_RUNNER_HOME%\bin;%JAVA_HOME%\bin
```

When your machine is back up, head to section B.1.3 for instructions on verifying and configuring your installation.

B.1.2 Install: Ubuntu

Installing SonarQube Runner is short and sweet for Linux. Run the following commands to expand the zip file and sym-link it into the default path:

```
sudo unzip -d /usr/share sonar-runner-<version>.zip          <-- Expand zip
sudo ln -s /usr/share/sonar-runner-<version> /usr/bin/sonar-runner   <-- Create link
```

Once you're done, continue to the next section to verify and configure your installation.

B.1.3 Verify and configure : Ubuntu and Windows 7

Now that SonarQube Runner is installed—whatever your platform—the instructions for using it are the same for everyone. First you need to verify your installation, so head to a command line and type:

```
sonar-runner -h
```

If you get a usage message in response, then you're good to go.

You could choose to use SonarQube Runner as is, but you'll save yourself some effort down the road if you configure a few of the more common properties globally rather than on a project-by-project basis. Every analysis, no matter how it's performed, needs to know where its target database is and what the credentials are to access that database. So you should configure those properties centrally.

To set these central properties, go to the SonarQube Runner home directory, and open conf/sonar-runner.properties in your favorite text editor. Uncomment (remove the leading `#` character) from the `sonar.host.url` line as well as the lines with the cor-

rect database driver and URL for your SonarQube install, and the database username and password. Be sure to correct the hostname in all the uncommented `.url` properties if you're not setting up SonarQube Runner on the machine that hosts SonarQube. Also correct the database password—assuming you didn't use the default. (You *didn't*, did you?)

B.1.4 Analyze

With SonarQube Runner installed and configured, you're ready to run your first analysis. In the target project's root directory, create a file named sonar.properties and open it in your favorite text editor. The following listing shows a sample properties file. Adjust it to match your project. Note that the `libraries` property is a comma-delimited list of explicit library files or the wildcard paths to them, as shown in the listing.

Listing B.1 SonarQube Runner project properties file

```
# required metadata
sonar.projectKey=test:project2
sonar.projectName=Chapt 2 Javarunner
sonar.projectVersion=1.0

# Comma-separated list of library directories
sonar.libraries=lib/*.jar

# comma-delimited list of paths to source directories (required)
sonar.sources=src

# comma-delimited list of paths to test source directories (optional)
#sonar.tests=testDir1,testDir2

# path to project binaries (optional)
sonar.binaries=bin

sonar.host.url=http://localhost:9000

sonar.jdbc.url=jdbc:mysql://localhost:3306/sonar?useUnicode=
true&amp;characterEncoding=utf8
sonar.jdbc.driver=com.mysql.jdbc.Driver

sonar.jdbc.username=sonar
sonar.jdbc.password=sonar
```

Skip if configured centrally

Once your properties file is configured, you can use it as is, like this:

```
cd <project root directory>
sonar-runner -Dproject.settings=sonar.properties
```

In response, you'll get about 80 lines (or more) of output. Here's a condensed version:

```
Runner configuration file: /usr/share/sonarRunner/conf/sonar-
     runner.properties
Project configuration file: <path to project root>/sonar.properties
Runner version: 1.2
Java version: 1.6.0_31, vendor: Sun Microsystems Inc.
OS name: "Linux", version: "3.0.0-19-generic", arch: "i386"
Server: http://<your server>:9000
```

```
Work directory: <path to project root>/.sonar
18:32:31.238 INFO      o.s.c.p.Database - Create JDBC datasource
18:32:31.869 INFO  actDatabaseConnector - Initializing Hibernate
18:32:33.740 INFO  .s.b.b.ProjectModule - -------------  Analyzing <Project
     Name>
...
18:32:41.453 INFO  p.PhasesTimeProfiler - Sensor CoberturaSensor done: 0 ms
18:32:42.207 INFO  p.PhasesTimeProfiler - Execute decorators...
18:32:43.793 INFO  .b.p.UpdateStatusJob - ANALYSIS SUCCESSFUL,
you can browse http://localhost:9000
18:32:43.794 INFO  b.p.PostJobsExecutor - Executing post-job class
org.sonar.plugins.core.batch.IndexProjectPostJob
18:32:43.901 INFO  b.p.PostJobsExecutor - Executing post-job class
org.sonar.plugins.dbcleaner.ProjectPurgePostJob
...
Total time: 14.082s
Final Memory: 8M/112M
```

When you see the `ANALYSIS SUCCESSFUL` line, the runner isn't quite finished—there are still database cleanup routines to go through—but your metrics will be ready, and you can head over to your SonarQube host to see them.

B.1.5 Multi-module projects

If you're dealing with a multi-module project, you need to set up the properties for each module. Fortunately, module properties follow an inheritance model, so you can set the common ones (or the most common ones) at the parent level and override what you need to at the module level.

You have two choices of how to set up your module properties. You can put everything in a parent-level sonar-project.properties file, or you can combine a parent-level properties file (it's needed either way) with a properties file per module.

Again, a parent-level sonar-project.properties file is needed either way. At a minimum, it will contain the following:

- `sonar.projectKey=com.myCompany:multiModuleProject`
- `sonar.projectName=Multi Module Project`
- `sonar.projectVersion=1.0`
- `sonar.modules=moduleA,moduleB`

The `sonar.modules` property holds a comma-delimited list of module names or aliases. If your module's name doesn't match its directory, you also need to specify its directory, like so: `moduleB.sonar.projectBaseDir=path/to/moduleB`.

Notice how the base directory property is prefixed with `moduleB`. All module-specific properties specified in the parent properties file should begin with the project name/alias you specified in the modules list. Module properties specified in a separate, child-level file don't need the prefix. Make sure you set a property for the `projectName` of each module; after that, it's just a matter of adding any module-specific properties you need to override from the parent level.

Figure B.1 Editing settings.xml with NetBeans and Eclipse IDE

B.2 Analyzing with Maven

If your favorite build tool is Maven, you're in luck, because SonarQube has supported Maven analysis from the start. In fact, for quite a while you could only perform a SonarQube analysis with Maven, and even today the Maven analysis functionality is richer than that offered by the other methods.

To perform your own Maven-based analysis, we'll start by assuming that you already have Maven installed and configured and that Maven's bin directory is in your system path variable. You can verify that it is by typing `mvn` at a command prompt. You should see something like this in response:

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[ERROR] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
```

The beauty of Maven is that you don't need to explicitly download any additional libraries. Setting up Maven was the biggest step in the process. Let's look at how to configure an analysis.

B.2.1 Setup

First you need to set up a SonarQube profile specifying how to connect to SonarQube's database. To do that, edit Maven's global configuration file, settings.xml, which is located in $MAVEN_HOME/conf or in your home folder under a .m2 subdirectory. You can open it in your favorite text editor; or sometimes your IDE will show it with your project files, as shown in figure B.1, and you can edit it from there.

The next listing shows a sample SonarQube profile using MySQL. (For details on installing and configuring SonarQube to use MySQL, see appendix A.)

Listing B.2 Maven settings.xml configuration

```
<...>
  <profiles>
    <profile>
      <id>sonar</id>
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
      <properties>
          <sonar.jdbc.url>jdbc:mysql://localhost:3306/sonar?useUnicode=
          ➥ true&amp;characterEncoding=utf8
```

Profile name: choose something that makes sense to you

Activation must be enabled by default

MySQL connection string

```
        </sonar.jdbc.url>
        <sonar.jdbc.driverClassName>com.mysql.jdbc.Driver
        </sonar.jdbc.driverClassName>
        <sonar.jdbc.username>username you picked</sonar.jdbc.username>

        <sonar.jdbc.password>password you picked</sonar.jdbc.password>

        <sonar.host.url>http://localhost:9000</sonar.host.url>
      </properties>
    </profile>
  </profiles>
<...>
```

MySQL JDBC driver name

MySQL username you picked during database installation

MySQL username password you picked during database installation

SonarQube's host URL: if SonarQube isn't installed on the same machine with Maven, change it accordingly

Now that you've configured the SonarQube profile, you need to add a `sonar-maven-plugin` dependency to your project's pom.xml file. You can skip this step if you want, but it's a good idea to add this dependency because SonarQube provides two Maven plugins: one for Maven 2 and another for Maven 3. Adding the `sonar-maven-plugin` section to pom.xml ensures that the correct one is always used. If you don't know which version of Maven you're using, type this at a command line: `mvn -v` or `mvn -version`. You'll get something like this:

```
Apache Maven 2.2.1 (r801777; 2009-08-06 22:16:01+0300)
Java version: 1.6.0_26
Java home: C:\Program Files\Java\jdk1.6.0_26\jre
Default locale: el_GR, platform encoding: Cp1253
OS name: "windows 7" version: "6.1" arch: "amd64" Family: "windows"
```

Version number listed here

The good folks at SonarSource make setting up the plugin dependency as easy as possible by providing the exact XML to use, and we've replicated it here for your convenience in listings B.3 through B.5. If you're on Maven 2, then add the code in listing B.3 to your pom.xml. If you're on Maven 3, then use what's in listing B.4.

Listing B.3 Maven 2 plug-in specification for project pom.xml

```
<build>
  <pluginManagement>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>sonar-maven-plugin</artifactId>
      <version>1.0</version>
    </plugin>
  </pluginManagement>
</build>
```

Listing B.4 Maven 3 plug-in specification for project pom.xml

```
<build>
  <pluginManagement>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>sonar-maven-plugin</artifactId>
```

```
      <version>2.0</version>
    </plugin>
  </pluginManagement>
</build>
```

If you need to be able to build your project with either Maven version, then you must create two profiles in pom.xml and let Maven decide which one to use. Listing B.5 shows you how. The tricky part is in the `activation` tag, which relies on the existence (or not) of the `${basedir}` expression. Maven 3 recognizes `${basedir}` but Maven 2 doesn't, and you set the `sonarVersion` attribute accordingly.

Listing B.5 Analyzing a project with either Maven version

```
<build>
  <pluginManagement>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>sonar-maven-plugin</artifactId>
      <version>${sonarVersion}</version>
    </plugin>
  </pluginManagement>
</build>
<profile>
  <id>m2-profile</id>
  <activation>
    <file>
      <missing>${basedir}</missing>
    </file>
  </activation>
  <properties>
    <sonarVersion>1.0</sonarVersion>
  </properties>
</profile>
<profile>
  <id>m3-profile</id>
  <activation>
    <file>
      <exists>${basedir}</exists>
    </file>
  </activation>
  <properties>
    <sonarVersion>2.0</sonarVersion>
  </properties>
</profile>
```

B.2.2 First analysis

You're ready to run your first SonarQube analysis using Maven. At a command prompt, `cd` to the directory holding your pom.xml file. You could skip straight to the analysis, but it's recommended that you run the Maven `install` goal first, like so:

```
mvn clean install
```

Once that's complete, run the analysis:

```
mvn sonar:sonar
```

Alternately, you can run both goals in a single command:

```
mvn clean install sonar:sonar -Dmaven.test.failure.ignore=true
```

We need to mention a couple of points here:

- The parameter `-Dmaven.test.failure.ignore` should always be used to instruct SonarQube to continue with analysis even if one or more tests fail.
- By default, your unit tests are executed twice: once for the install goal and once for the sonar goal. It's possible to skip the second run and reuse the test results from the first run. For details, see `sonar.dynamicAnalysis` and its companion properties in section B.4.
- Avoid using the Maven parameters `-Dtest=false` and `-DskipTests=true`. They'll prevent SonarQube from running your unit tests, and you won't get any test metrics.

B.2.3 Multi-module

If you have a multi-module project, you can easily analyze it using the steps just outlined. Each child module automatically inherits its parent properties. If there are properties you want overridden at the child level, such as `sonar.language`, define them in the child module's pom.

B.3 Analyzing with Ant

If you're in an Ant-build shop, you'll be happy to know that the good folks at SonarSource have ported analysis functionality into an Ant task. Before you can use it, you'll need to download the Ant task jar from http://docs.codehaus.org/x/QYKxDQ. The best place to go for the download is the SonarQube wiki, where the latest links and documentation are maintained.

Put the Ant task jar somewhere central. You can make it readily available to everyone by dropping it into the lib directory in the Ant installation (`${ant.library.dir}`), or put it in your user home in a directory named .ant/lib.

Once that's done, you're ready to go, and listing B.6 gives you a sample target. Notice that it refers to the Ant task jar as sonar-ant-task.jar, rather than sonar-ant-task-<*version*>.jar. That's because we typically cheat by either renaming the jar to remove the version number or adding a version-less symlink to the directory. If it's not your practice to do either, then you need to correct the name of the jar.

Listing B.6 Ant script

```
<property name="src" value="src"/>
<property name="classes" value="build/classes"/>

<target name="sonar" depends="compile">
  <taskdef uri="antlib:org.sonar.ant"
       resource="org/sonar/ant/antlib.xml">
    <classpath path="${ant.library.dir}/sonar-ant-task.jar" />
  </taskdef>
```

```
<!-- if you pass in libraries, it needs to be in the form
     of a comma-delimited list of jar files -->
<path id="tmpPath">
  <fileset dir="lib" includes="**/*.jar"/>
  <fileset dir="${someother.library.dir}" includes="*.jar"/>
</path>
<pathconvert dirsep="/" pathsep=","
    property="sonar.libraries" refid="tmpPath"/>

<!-- database access -->
<property name="sonar.jdbc.url"
    value="jdbc:mysql://localhost:3306/sonar?useUnicode=
      true&amp;characterEncoding=utf8" />
<property name="sonar.jdbc.driverClassName"
    value="com.mysql.jdbc.Driver" />
<property name="sonar.jdbc.username" value="sonar" />
<property name="sonar.jdbc.password" value="${yourSonarPassword}" />

<!-- required -->
<property name="sonar.sources" value="${src}" />

<!-- optional -->
<property name="sonar.projectName" value="My Sample Project" />
<property name="sonar.binaries" value="${classes}" />

<sonar:sonar key="org.example:example" version="0.1"
xmlns:sonar="antlib:org.sonar.ant"/>

</target>
```

Omit this property if compiled classes aren't available

There are a couple of things to note about this listing. The first is that it's not a full project file. We're assuming you've already got one into which you need to incorporate analysis. Second, we've made the `sonar` target dependent on an (unprovided) `compile` target.

Of course, you're free to remove that dependency (also remove the `sonar.binaries` property); but you'll probably want to leave it in because the more you give SonarQube, the fuller the analysis it returns, and the compiled binaries can be an important part of a full analysis.

And of course, the elephant in the room for true Ant-heads is that the whole thing isn't very Ant-y. At all. That's probably because SonarSource is a Maven shop, and Ant-based analysis isn't its primary focus.

That's why you *must* give the properties the names shown in the example if you want the analysis to run correctly. That's also why this example turns what you might expect to see as a path reference into a comma-delimited string that lists paths to jars.

Telling SonarQube where your dependencies are can enhance the quality of the analysis you get, but unfortunately you can't just pass the SonarQube Ant task a resource collection, or even a stereotypical **/*.jar path. Instead it needs a comma-delimited list of jars. Assembling that list is what this chunk of code early in the target does:

```
<path id="tmpPath">
  <fileset dir="lib" includes="**/*.jar"/>
  <fileset dir="${ant.library.dir}" includes="*.jar"/>
```

```
</path>

<pathconvert dirsep="/" pathsep="," property="sonar.libraries"
 refid="tmpPath"/>
```

Like binaries, libraries are optional, so feel free to leave this part out.

If you're doing a multi-module build and want to make it a multi-module analysis as well, you need a few more (specifically named) properties. The first is

```
<property name="sonar.modules" value="moduleA,moduleB"/>
```

The `sonar.modules` property holds a comma-delimited list of module names or aliases. If your module's name doesn't match its directory, you also need to specify its directory, like so:

```
<property name="moduleB.sonar.projectBaseDir" value="path/to/moduleB"/>
```

Notice how the base directory property is prefixed with `moduleB`. All module-specific properties should begin with the project name/alias you specified in the modules list. Make sure you set a property for the `projectName` of each module, and after that, it's just a matter of adding any module-specific properties you need to vary from what you've already set for the analysis as a whole.

B.4 Analysis properties

SonarQube provides a variety of advanced properties to customize analysis. Some of them can be set permanently on a project through the SonarQube interface, but others are available only as analysis parameters. They allow you to adjust individual analysis runs to meet the needs of the moment. The tables in this section list the available properties and discuss how they affect an analysis.

Table B.1 lists required analysis properties. Some must be explicitly stated for all analysis types; others, marked "Required for non-Maven analysis," are inherited from elsewhere in the project settings for Maven analyses but must be explicitly stated for other analysis types.

Table B.1 Required analysis properties

Property	Example	Details
`sonar.projectKey`	`com.myDomain:projectId` Format should be `a:b`	What's shown here is a stereotypical Maven project ID. There's no requirement that your project use Maven to be analyzable, but you still need to give it an ID in the Maven format of `something :somethingElse`. Don't worry that it's ugly. It's not shown in many places. For an Ant analysis, this requirement is fulfilled by the `key` argument to the analysis task.

Table B.1 Required analysis properties *(continued)*

Property	Example	Details
`sonar.projectName`	`My Project`	This value is what's usually shown to users. It may contain spaces.
`sonar.projectVersion`	`1.0`	As this value changes, SonarQube records events, which are used to flag project snapshots, for long-term retention and comparison.
`sonar.sources`	`srcDir1,srcDir2`	Required for non-Maven projects. This is a comma-delimited list of paths from project root to the directories holding source files. Directories are scanned recursively, so there's no need to list subdirectories. If the source files are *in* the project root, set this value to a single period (`.`)

`sonar.jdbc.driverClassName`	One of the following:
	`com.mysql.jdbc.Driver`
	`oracle.jdbc.OracleDriver`
	`org.postgresql.Driver`
	`net.sourceforge.jtds.jdbc.Driver`
		During an analysis, SonarQube must know how to reach its database. This property tells it which database driver class to use to make a connection.

`sonar.jdbc.url`	`jdbc:mysql://localhost:3306/sonar?useUnicode=true&characterEncoding=utf8&rewriteBatchedStatements=true`	This property tells the local SonarQube analysis engine where its database is. It's used in conjunction with the `sonar.jdbc.driverClassName` property and the two credentials properties.
`sonar.jdbc.username`	`sonar`	This is the name of the SonarQube database user you created during setup and installation in appendix A.
`sonar.jdbc.password`	`<password you picked>`	This is the password you assigned to the SonarQube database user you created during setup and installation in appendix A.

Table B.1 Required analysis properties *(continued)*

Property	Example	Details
`sonar.login` `sonar.password`	Credentials of a user with permissions to SonarQube and/or User role on the project	These properties are required only if one of the following two conditions is true: ▪ Your SonarQube instance is secured and requires a user to log in for basic access. ▪ The Anyone group doesn't have the User role on the project under analysis.

None of the properties in table B.2 are required, but their use will help you get the most out of a SonarQube analysis, because the more data you give SonarQube, the richer the analysis you'll get back. As with some of the properties in table B.1, some of these properties are available implicitly for a Maven analysis.

Table B.2 Analysis enrichment properties

Property	Example	Details
`sonar.tests`	`testDir1,testDir2`	This is a comma-delimited list of paths from project root to the directories holding test files. Implicit for Maven analysis.
`sonar.binaries`	`binDir`	This is a comma-delimited list of paths from project root to the compiled byte code. Implicit for Maven analysis.
`sonar.libraries`	`lib/lib1.jar,lib/*.jar`	This is a comma-delimited list of paths to individual library files or directories. For a SonarQube Runner analysis, you can make this a comma-delimited list of paths to lib directories and end each entry with *.jar or whatever the appropriate extension is for your language. For an Ant analysis, this needs to be a comma-delimited list of paths to exact libraries. Listing B.6 in the Ant analysis section of this appendix gives you a way to do that. Implicit for Maven analysis.

Table B.2 Analysis enrichment properties *(continued)*

Property	Example	Details
`sonar.dynamicAnalysis`	One of `true`, `false`, `reuseReports`	Unit tests are executed by default, but you can choose to turn that off or to reuse previously generated reports. If you choose `reuseReports`, you need to use one of the companion properties to specify the report type and location.

`sonar.jacoco.reportPath`
`sonar.surefire.reportsPath`
`sonar.coberatura.reportPath`
`sonar.clover.reportPath`
	`path/to/reports`	This is the absolute or relative path to the chosen reports.

Table B.3 lists properties you can use to tune various aspects of your analysis, such as which language is being analyzed and whether byte-code analysis is performed. It also lists a number of properties you can use to override values set at the server, such as which rule profile to use.

Table B.3 Analysis tuning and overrides

Property	Example	Details
`sonar.language`	One of `abap`, `c`, `cobol`, `delph`, `cs` (C#), `flex`, `grvy`, `java`, `js`, `natur`, `php`, `plsql`, `py`, `vb`, `web`, `xml`	This specifies the single non-Java language under analysis. Many properties may be set to comma-delimited lists, but this property may not be. It only accepts a single value.
`sonar.importSources`	`true / false`	This defaults to `true`, which means by default, SonarQube imports your source files into its database for display in the interface as a reference and for clearer illustration of the issues it finds. There are some controls you can place around who can view project sources in SonarQube, but if it's ultra-sensitive, set this property to `false` to prevent the source files from being imported.
`sonar.projectDate`	`yyyy-MM-dd`	This defaults to `now`. If you'd like SonarQube to catch up the history on a project, use this property to tell SonarQube what date you checked out from your SCM.
`sonar.branch`	`[your SCM branch name]`	If you have multiple SCM branches of a project under analysis, use this property to differentiate them from `Head` and each other.

Table B.3 Analysis tuning and overrides *(continued)*

Property	Example	Details
`sonar.sourceEncoding`	`UTF-8`	This is the character set your source files are saved in. `UTF-8` is the default and is typically correct for most projects.
`sonar.exclusions`	`com/myCo/genned/*.java, com/**/*Dummy.java`	This is a comma-delimited list of paths to files or directories that should be ignored during analysis. No metrics—including the basics like size—will be calculated on anything listed here. Wildcards in file or path names will be expanded. The examples at left will exclude the following: ▪ `com/myCo/genned/*.java` Every Java file directly under com/myCo/genned. Files in subdirectories *will* be included in the analysis. ▪ `com/**/*Dummy.java` Every file anywhere under com/ that ends with Dummy.java. For example, com/mom/ShesNoDummy.java will be excluded. Exclusion paths may also be set through the SonarQube interface, which is the preferred method. If set there, they're remembered from analysis to analysis, without the need to repeat them each time in the analysis properties.
`sonar.skippedModules`	`names,ofModules,toSkip`	If your Maven project is composed of multiple modules and you want some of them skipped during analysis, list their IDs here in comma-delimited format. Does not apply to non-Maven projects.
`sonar.includedModules`	`names,ofModules,toInclude`	If your Maven project is composed of multiple modules and you want only some of them analyzed, list their IDs here in comma-delimited format. Does not apply to non-Maven projects.

Table B.3 Analysis tuning and overrides *(continued)*

Property	Example	Details
`sonar.profile`	`Sonar way with Findbugs`	Set this property to the name of the profile you want your project analyzed with. This property overrides the profile that's set at the server. For long-term use, it is preferable to assign a project to the chosen profile via SonarQube's configuration interface. Note that using this property may cause confusion—because the dashboard description widget displays the name of the profile the project is assigned to (via the configuration interface) rather than the one it was last analyzed with (via this `analysis` property).
`sonar.skipDesign`	`true` / `false`	SonarQube implements byte-code analysis to determine dependencies and other design metrics. Set this property to `true` if for some reason you need to skip that analysis.
`sonar.phase`	`[Maven phase name]`	Use this property to have a Maven goal or phase executed before analysis starts. When SonarQube needs a phase or Maven goal to be executed prior to analysis, this parameter can be used. For example, `sonar.phase=generate-sources`.
`sonar.java.source` `sonar.java.target`	One of `1.5`, `1.6`, `1.7`	These properties align the source and target parameters to the Java compiler. If you're using those parameters, set these properties correspondingly.
`sonar.findbugs` `.excludesFilters`	`relative/path/to/file`	SonarQube lets you exclude certain classes or packages from analysis, but FindBugs, one of the tools SonarQube can use for Java analysis, provides a much more fine-grained ability to set up exclusions for specific bugs or bug types against classes or packages. See: http://findbugs.sourceforge.net/manual/filter.html. Those FindBugs filters are defined in an XML file. If this property is defined, SonarQube passes in its value when the FindBugs portion of an analysis is invoked—assuming the rule profile in force includes FindBugs rules.

The properties in table B.4 shouldn't usually be needed, but they're useful if you're trying to debug an analysis problem.

Table B.4 Troubleshooting and debugging

Property	Example	Details

`sonar.host.connectTimeoutMs`
`sonar.host.readTimeoutMs`
	`100000`	Measured in milliseconds, these timeouts apply to Maven-based analyses, which make some HTTP requests to the server. Because two timeouts will make the call fail, you may want to increase these values if your server is slow and you're having trouble.
		Defaults:
		`connectTimeoutMs` = `30000`
		`readTimeoutMs` = `60000`

`sonar.verbose`	`true` / `false`	Defaults to `false`. Set this to `true` to see a lot more detail in your analysis logs.
`sonar.showSql`	`true` / `false`	Defaults to `true`. Set this to `true` to see all SQL queries executed by SonarQube during batch analysis. Useful for debugging purposes and when you're developing custom plug-ins.
sonar.showSqlResults	`true` / `false`	Defaults to `true`. Set this to `true` to see the results of SQL queries executed by SonarQube during batch analysis. Useful for debugging purposes and when you're developing custom plug-ins.

index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

SOFTWARE DEVELOPMENT

SonarQube IN ACTION

Campbell • Papapetrou

SonarQube is a powerful open source tool for continuous inspection, a process that makes code quality analysis and reporting an integral part of the development lifecycle. Its unique dashboards, rule-based defect analysis, and tight build integration result in improved code quality without disruption to developer workflow. It supports many languages, including Java, C, C++, C#, PHP, and JavaScript.

SonarQube in Action teaches you how to effectively use SonarQube following the continuous inspection model. This practical book systematically explores SonarQube's core Seven Axes of Quality (design, duplications, comments, unit tests, complexity, potential bugs, and coding rules). With well-chosen examples, it helps you learn to use SonarQube's review functionality and IDE integration to implement continuous inspection best practices in your own quality management process.

What's Inside

- Gather meaningful quality metrics
- Integrate with Ant, Maven, and Jenkins
- Write your own plug-ins
- Master the art of continuous inspection

The book's Java-based examples translate easily to other development languages. No prior experience with SonarQube or continuous delivery practice is assumed.

Ann Campbell and **Patroklos Papapetrou** are experienced developers and team leaders. Both actively contribute to the SonarQube community.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/SonarQubeinAction

"A unique source of information for successful implementation."

—From the Foreword by Olivier Gaudin, CEO of SonarSource

"Not just a reference manual for Sonar, but a guide to retooling your entire software development process."

—Alex Garrett
Hot Towel Consulting

"Lives up the high standards of Manning *In Action* books ... provides a great narrative on how to complement and extend Sonar's online documentation."

—Steve Hicks, MyDonate

"Highly recommended for all agile engineers."

—Michael Hüttermann
Author of *Agile ALM*

ISBN 13: 978-1-617290-95-4
ISBN 10: 1-617290-95-5
54999
9 781617 290954

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]