SAMPLE CHAPTER

SonarQube IN ACTION

G. Ann Campbell
Patroklos P. Papapetrou

FOREWORD BY Olivier Gaudin

MANNING

SonarQube in Action

by G. Ann Campbell

Patroklos P. Papapetrou

Chapter 5

brief contents

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You've warned Sam that the library's documentation is out of date, so she starts by looking at its code, but she realizes pretty quickly that it doesn't match what's in production.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.

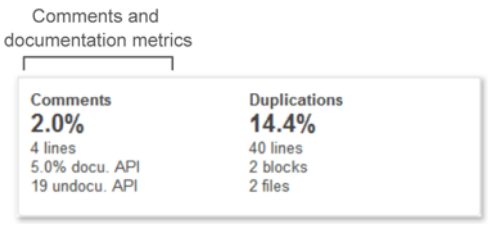

Figure 5.1 The comments and duplications widget appears in SonarQube's default dashboard.

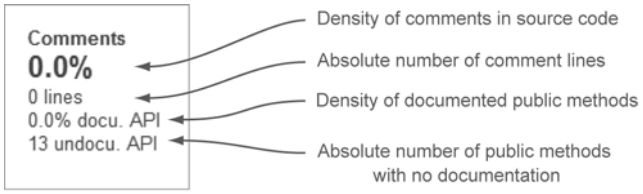

Figure 5.2 SonarQube metrics shown in the comments and documentation widget

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.

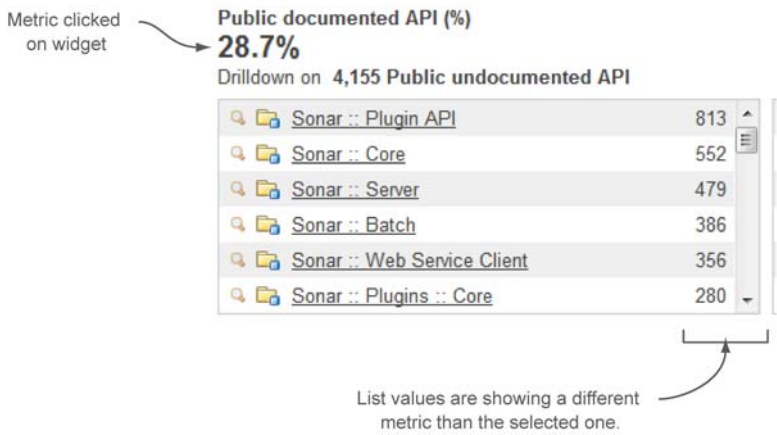

Figure 5.3 When the click-through metric is Public Undocumented API, the drilldown view is a variation on the norm. Instead of seeing files sorted by that density, they're shown with and sorted by the absolute number of undocumented API.

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.

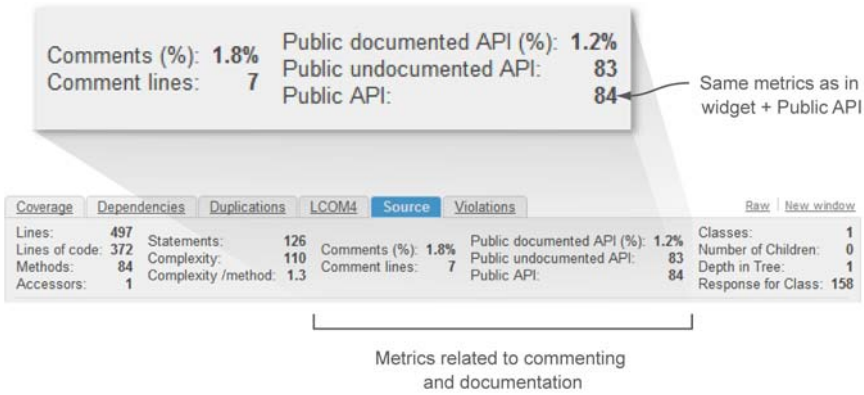

Figure 5.4 **Source code tab's header**

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.

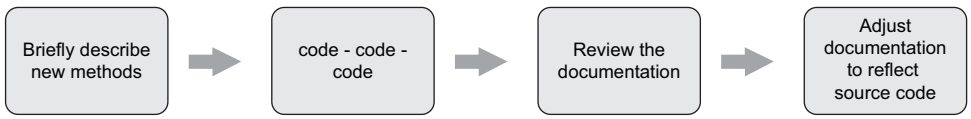

Figure 5.5 Simple documentation process

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.

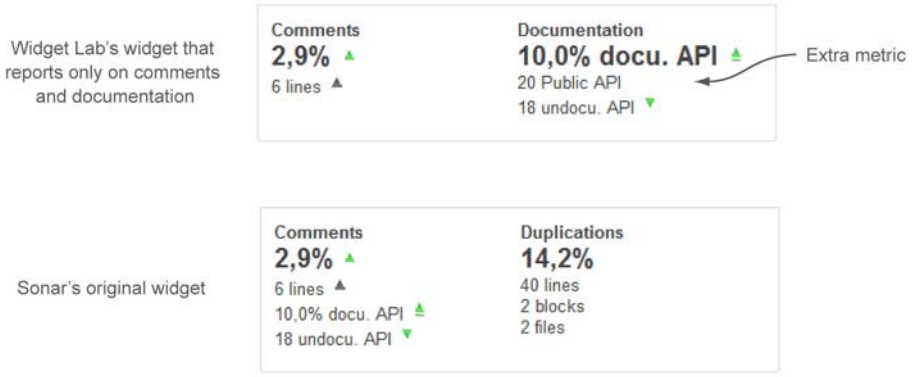

Figure 5.6 Visual comparison between SonarQube's original widget and the cloned widget provided by the Widget Lab plugin

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.

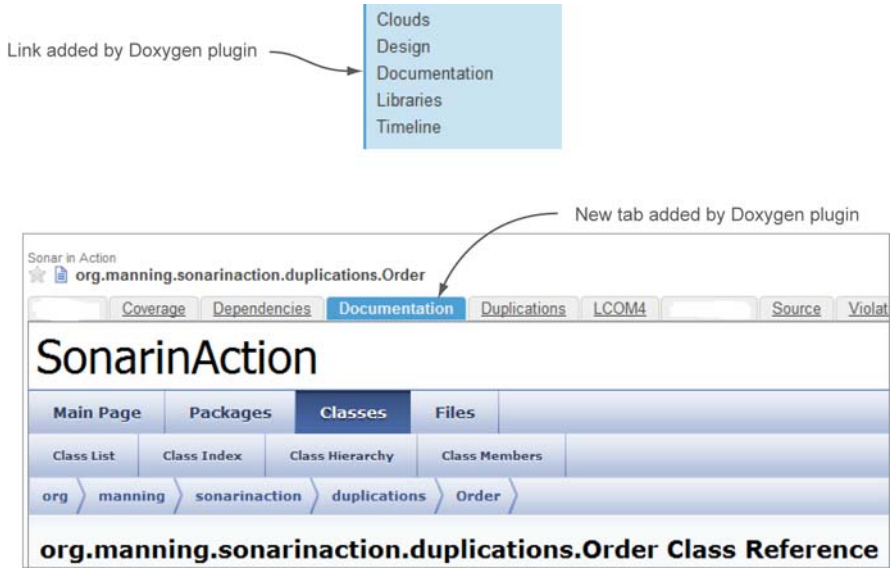

Figure 5.7 Doxygen plugin's new link and new Documentation tab in the source code viewer

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!

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]