SonarQube

Introduction

SonarQube is a software development tool used to statically analyze projects and improve security and code quality. It is available for use among many project formats and languages, and provides opinionated guidance on the best practices for more maintainable codebases. With this in mind, we wanted to find out if the developers of SonarQube practice what they preached.

For the purpose of this project, we forked the open-source SonarQube Community edition. This is a multi-module project developed in Java and built with Gradle. The 15 submodules contain code for the core static analysis, a web server and its related components, and other features. It contains ~230K lines of code, with ~23K JUnit unit tests providing code coverage of 91.6%. Initial impressions of the code implies a high code quality.

To assist in analyzing the overall project, we decided to use Docker containers for PostgreSQL and SonarQube to analyze the local fork on our respective machines. Analysis was performed on the *master* Git branch and care was taken to ensure that created solutions were compatible with both Windows and Mac environments. Extending project tooling proved to be a formidable task, but we were successful in integrating new tasks in the root projects *build.gradle* module. This approach ensured continued improvement across all submodules, even those not targeted by a test stage's goal. In the following sections, we will discuss our findings and additions.

Testing Stages

Unit Testing

Given its already high coverage and wide breadth of module functionality, finding new areas to add unit tests proved to be difficult. Since most of the core functionality was covered, we decided to target uncovered edge cases. Our investigation found various utility functions that had tested base expected behavior but did not cover all possible variants of outcomes, such as different human readable units of file sizes in *FileUtils* or the testing of exception handling for invalid date formats in *UtcDateUtils*. While these tests did not significantly improve coverage, they helped make the existing test bed more robust against accidental regressions.

Within the repository were many interface definitions that interacted with concrete implementations. This provided a good opportunity to create a mock implementation of these interfaces and ensure their interactions were properly recorded. We targeted *LanguagesProvider*, which is dependent on the *Languages*

interface, and *CoreExtension*, which is dependent on the *Context* interface. By mocking these interfaces, we more firmly solidified the expected behavior of each target implementation, ensuring that new changes will not modify the existing behaviors without regard for impact.

Mutation Testing

The original repository for SonarQube did not have an existing configuration for mutation testing. We set up Pitest, a common mutation testing tool for Java projects, with the JUnit 5 plugin. The tool was configured as a task on the root-level Gradle build file, so it would run tests for all of the modules in the repository.

When run, Pitest would generate static HTML reports that could be used to identify problematic test files and would suggest remediation to kill surviving mutants. The base mutation coverage varied widely, with modules having any between a 0% and 82% mutation score. The average mutation coverage was approximately 70%.

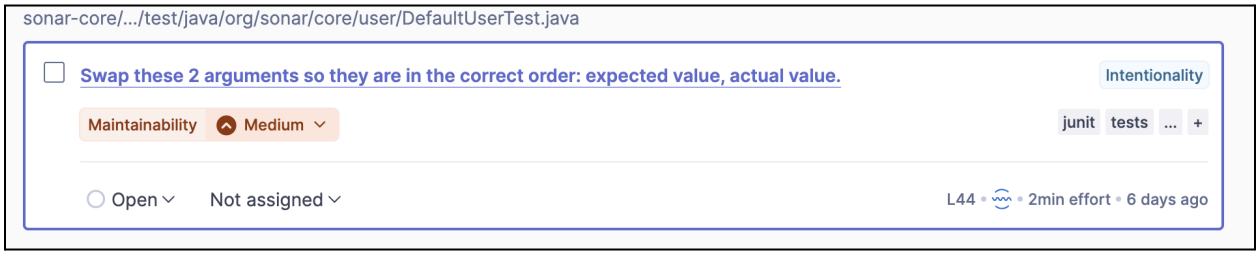
We targeted the *sonar-core* module, as it had the most understandable logic and generated a total of 1,499 mutants. We then focused on the DefaultUser class, as it was fairly simple and had 0% mutation coverage due to a previous lack of test coverage. After adding tests and verifying that they handle mutants appropriately, the DefaultUser.java file had its mutation coverage increased to 100%, with all 18 mutants being killed. This change increased the total average mutation coverage from 70% to 71%. The table below shows the final results of running Pitest on the *sonar-core* module.

sonar-core	Value
Total Mutants	1499
Killed Mutants	1060
Surviving Mutants	439
Mutation Coverage (%)	71

Static Analysis & Code Smell Detection

As indicated by the earlier use of the SonarQube local server to gather code coverage, it was only natural that we also used it to perform static analysis and code smell detection. As a result, more focus can be placed on the issues, rather than the work to configure necessary tooling.

When running the code analysis, we found that we had actually introduced 8 new issues since we had forked the project. These issues were present in the changes made for mutation testing—unsurprising, since Pitest did not require checking the SonarQube project dashboard to gather results. The introduced issues were flagged by SonarQube in two categories: Maintainability and Intentionality. The former was brought about by using a deprecated method in a utility file, and the latter suggested that we swap the expected and actual values of assertion statements. Once the deprecated method was replaced and the assertions followed the format *assert(expected, actual)*, the issues were removed from the dashboard. Below is an example of an issue in SonarQube Server.



Of note, the original SonarQube repository had over 8,000 open issues; the large majority of them related to maintainability. Although this number is quite large, this is not unreasonable given the scale of the SonarQube codebase. Also contributing to this is the use of deprecated code, which appears to be in part due to an ongoing migration to JUnit5. Maintainability remains an ideal place for improvement, but it appears to be low priority for the developers.

Integration Testing

Integration testing proved to be challenging; most of the accessible mockable interactions were already covered. The remaining cases required so many layered dependencies that test harnesses would have been incredibly complex. Regardless, an attempt was still made to create integration tests for edge cases. We used JUnit as the testing framework and Mockito for creating mocked components.

The included tests involved the integration between the *ComponetKeys* and *Uuids* classes. An existing test case was modified to mock the static method *Uuids.create()* to return a fixed UUID value. All the tests passed successfully, indicating that the integration worked as expected when generating component keys.

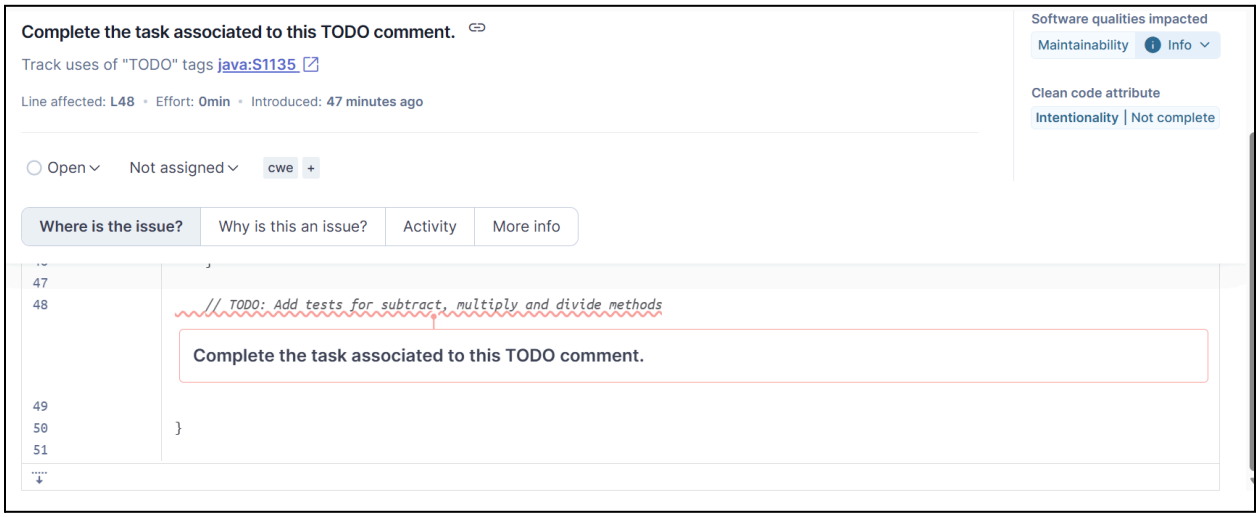
A large part of the issues with creating integration tests sprung from confusion about the existing project. With classes often including multiple ways to do the same

thing—for example, the UUID class being able to generate a UUID both statically or on an instance—additional complexity made it difficult to evaluate the impact of integration testing. We attribute this complexity to existing technical debt and the ongoing migration efforts to JUnit5.

System Testing

The system testing involved detecting code smells, a key feature of SonarQube's static analysis. To this end, we set up a simple Java project with unit tests and a Gradle workflow that would run the tests and upload the results to a SonarQube instance. We intentionally included code that SonarQube should mark as problematic: a TODO comment and commented-out code. These are common code smells that can cause issues with readability and maintainability if left alone.

When the analysis ran, SonarQube dashboard displayed two issues: one for the TODO comment and the other for the commented-out code. As expected, the issues included category information (in this case Maintainability), highlighted and displayed a view of the offending lines of code, and provided a suggestion for remediation. Below is an example of one of the issues, as seen in the SonarQube dashboard.



When creating this test, we had hoped for a means to automate this approach. Creating an entire example repository, adding code with issues, and then configuring the SonarQube project to receive analysis results was a lengthy manual process. At the time, it seemed that a more automated solution did not exist, but the process involved in the performance testing assignment indicated that the SonarQube Web API could have been used to perform most of these steps programmatically.

Security Testing

In addition to static code analysis and code smell detection, SonarQube also has built-in support for security analysis. This allows it to detect common vulnerabilities

such as SQL injection or token leaks, with each vulnerability being assigned a ranked priority. These vulnerabilities come from a variety of sources, including OWASP Top 10 2021, OWASP Mobile Top 10 2024, and CWE.

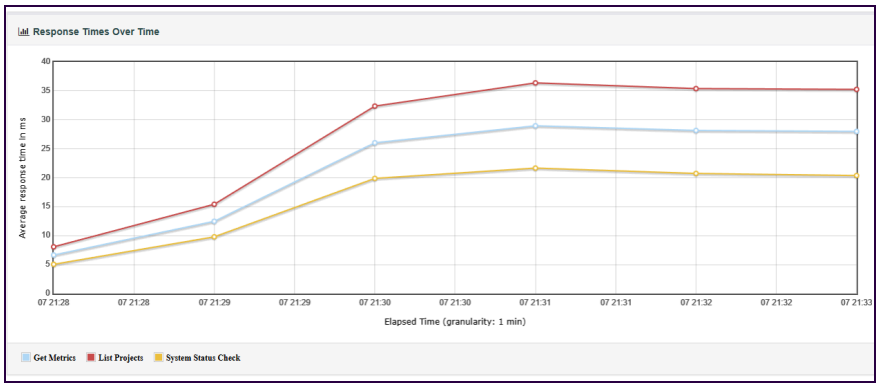
SonarQube is able to detect 23 security vulnerabilities in the SonarQube code base. Upon closer inspection, these vulnerabilities were not a high risk to the security of the code base, and several of them were false positives. A few prominent detected vulnerabilities included hard coded credentials, SQL statements vulnerable to injection, and use of weak hashing. In all of these cases, the code was either not production-facing—the credentials were in test files and the SQL statements were for internal use only—or was not being used in a situation that required high security. Based on these results, we decided to simply track the vulnerabilities instead of fixing them for minimal impact.

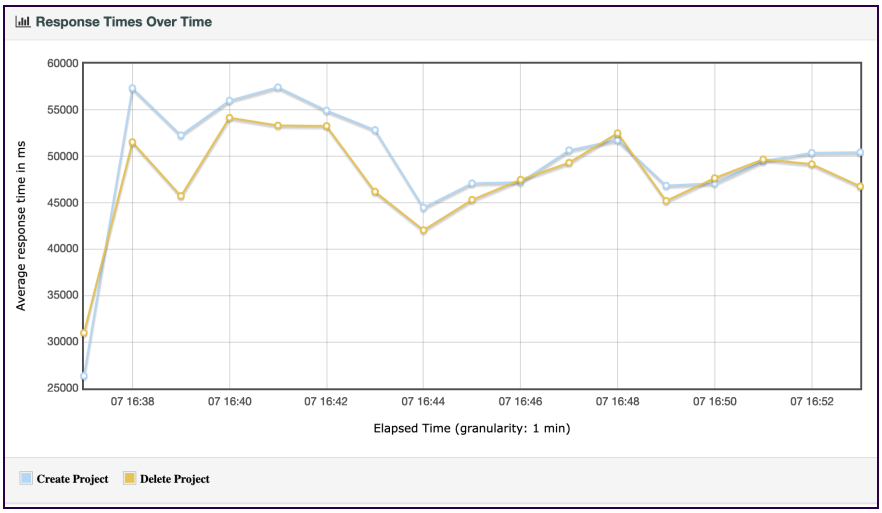
Performance Testing

The original repository of SonarQube also did not feature performance testing. So we implemented JMeter, a performance testing tool for Java, to perform load and stress tests. Using a local SonarQube instance, a Gradle task was made to run the tests from the CLI, without requiring knowledge of the test plan files. When run, JMeter would generate an HTML report with graph-based metrics for visualization.

Our tests sought to gauge how well the SonarQube is able to handle concurrent users accessing and creating projects. JMeter simulates a set number of users over a time frame of 5-15 minutes and makes requests to the SonarQube Web API. The load tests focused on GET endpoints that retrieved system and project information. This established a baseline for what the average user of SonarQube might experience. The stress tests tested how well SonarQube handles high quantities of concurrent requests to create and delete projects.

Unsurprisingly, the load tests performed quite well. The GET requests were quick, averaging a response time less than 50 milliseconds, even with simulated users hitting the endpoint once every 2 seconds for 5 minutes. The stress tests, given the increasing compute required to create projects, were significantly slower. Response times averaged around 50 seconds—impressive considering that 2,000 simulated users were attempting to create projects at the same time—with few failed requests. Graphs depicting the response times of the system over the course of the tests are shown below (top = load tests, bottom = stress tests).





Encountered Issues

As mentioned above, SonarQube does not appear to have any significant issues relating to the quality of its code and tests. The contributors of this project seem to adhere to the philosophy of “continuous improvement” that SonarQube boasts. The most glaring issue has to do more with the maintainability and understandability of the code base. The project’s 15 submodules are organized but not documented, so attempting to follow the relationships within the architecture often came from navigating through import statements that often resolve in simple interfaces that did not provide much context to their use.

In addition to the over 8K open issues on maintainability, the internal structure of the submodules was difficult to navigate and reason about. There were license statements at the head of every file, but little to no documentation on the internal functionality. This may be an artifact of being the open source version of a purchasable software, but more information about the structure and tests of the project would have been incredibly helpful in understanding the intention of the original authors and identifying potential gaps of thought.

As an end user, there ironically seems to be a bit of the opposite problem. SonarQube's documentation as a user is gargantuan, to the point that it is sometimes difficult to identify where the functionality you're attempting to implement may reside. This problem is preferable to the alternative, though some APIs (specifically the Web API) were usable but seemed incomplete.

Finally, initial project setup and tool additions proved to be somewhat difficult since the root *build.gradle* had to be compatible with all submodules. This took a while to set up, but thankfully seemed relatively straightforward once the tasks were set up. The other major issue was long builds. Gradle required several hours upon the first pull down of the repository, but this time sink was a one shot and subsequent task runs would take significantly less time thanks to the Gradle cache.

Improvements Made

Overall, making significant improvements to this codebase was outside of our abilities within the context of testing and improvements. High inter-module complexity combined with an already high level of test coverage made it difficult to find new opportunities to add testing. Some of the larger-scale testing of the system proved a bit easier, but had additional complexity and more difficulty in creating an automated test solution. That said, we were able to integrate Pitest and JMeter test tasks to apply for all submodules to a complete test bed. Our biggest impact seems to have been improvements to the *sonar-core* modules mutation score.

Quality Assessment

The SonarQube software as we found it has been high quality. The test cases we examined seemed largely robust, edge cases withstanding. We ensured that our additions to the repository were high quality by SonarQube's standards and did not introduce unnecessary errors. We did raise some quality concerns in our test cases, but these were often false positives (e.g., fake user credentials). The current official SonarQube repository has not had a significant increase in test coverage or issue resolution either. An initial glance at their repository shows a 2.5K issue resolution, with the majority of these changes simply being acknowledged and ignored. This could indicate a code smell, but it seems like that there is no good solution to some of the issues they are tracking and it is more effective to concretely document these choices.

Learnings

Working with a repository of this size taught us about the importance of high quality documentation. While the documentation for use as an end user was quite good, the internal documentation was severely lacking. Architecture diagrams, basic workflows, or even simple comments were not found, which was a significant bottleneck

weekly in attempting to identify where best to contribute. If we had more access to a bigger picture, it would have been significantly easier to, for example, perform integration tests by viewing the relationships between different modules.

Additionally, because of the nature of the repository being a mono-repo, there was probably too much for us to review that made it difficult to narrow down possible modules to review. Instead, we should have focused on the *sonar-core* repository from the beginning and limited our scope. We were successful in ensuring that functionality was implemented for all subsystems, but this effort took a significant amount of time that could have been better spent focusing on the tests and quality of the system. To its credit, working with this level of complexity serves as both a cautionary tale but also proves the worth of the build tools involved. Gradle was initially a roadblock in our efforts, but once we had a better understanding of its role then it became a bit more streamlined, though plugin version management added unnecessary overhead.

Team Members and Roles

Team members were Andrew Bradbury and Christian Ashley. Andrew performed test authoring, report scaffolding, and set up the JMeter tasking necessary to complete the performance testing. Christian investigated and documented the initial SonarQube instance set up and added Pitest.

References

1. https://github.com/SonarSource/sonarqube
2. https://pitest.org/
3. https://jmeter.apache.org/