SWEN-777 Final Report

Authors: Geoffrey Tse, Lucille Blain, Vanessa Nava-Camal

Project: JabRef

Overview

JabRef is built to support researchers throughout the entire workflow, offering four core functions that make managing academic sources much easier. It helps you **collect** material by retrieving full-text articles and fetching complete bibliographic data from identifiers like ISBN, DOI, PubMed, and arXiv. You can **edit** entries to clean up or enhance metadata and even have JabRef automatically rename and organize attached files. Its **organization** features let you structure your library with hierarchical groups, keywords, tags, and custom search terms. And when it's time to write, JabRef's **cite** tools provide native BibTeX and BibLaTeX support along with smooth cite-as-you-write integration for external editors.

Testing Stage

Unit Testing

For unit testing around coverage and expansion, the focus was on making sure the system handled an important edge case: BibEntries that contain multiple authors. Tests were added for both the ItemDataProvider and BibEntry classes, confirming that the code correctly identifies each author in a single entry without breaking existing functionality. This work directly improved code coverage and highlighted a small issue that was fixed to ensure the logic stays reliable.

For unit testing involving mocking and stubbing, the goal was to see how the system responds when things go wrong, especially during PDF imports. The tests showed that the PDF importer deals with I/O failures, encrypted files, and other problematic PDFs in a safe and predictable way. Since the importer already handles these scenarios consistently, the main takeaway was simply to document this behavior rather than change anything in the implementation.

Mutation Testing

Mutation testing gave a deeper look at how resilient the current test suite really is. Out of 918 generated mutations, only 337 were killed, which put the mutation score at a low 37%. More than half of the mutations had no coverage at all, but the tests that were exercised showed strong assertions, giving the suite an overall test strength of about 89%. Since the results pointed to a large amount of untested logic across JabRef's core classes, even with around 700 test cases, it was clear that this wasn't just

something to document. The gaps were addressed directly, improving coverage and strengthening the reliability of the codebase.

Static Analysis

Static analysis with SonarQube highlighted several areas where code quality and maintainability could be improved, especially in files like `ModsImporter.java`, `XmpUtilWriter.java`, and `Formatters.java`. The most impactful change came from simplifying `ModsImporter.java`, where switching to the Java XML library and breaking out the exception handling into its own function helped cut down unnecessary complexity. These findings weren't just noted, but rather they were addressed by adding new test cases to support the cleaner, more maintainable structure.

Integration Testing

Integration testing showed that most combined workflows behaved normally, but it also uncovered a subtle issue that unit tests alone didn't catch. When testing interactions between modules like `BibDatabase`, `PdfImporter`, `ParserResult`, and their components such as `Formatter`, `Author`, and `StudyCatalog`, the duplicate detection logic flagged entries as duplicates even when they had the same author and title but different publication years. This revealed that the system behaved differently once the pieces were working together. Since the root cause was in how BibDatabase interpreted these entries, the outcome was documented so the behavior is clearly understood going forward.

System Testing

System testing focused on how JabRef behaves end-to-end, especially across the model layer and the string utility layer. These tests looked at how BibEntry fields interact with various StringUtil transformations and verified that the system handled those conversions safely. One of the more interesting outcomes was confirming that mixing HTML transformations with regular ASCII codes didn't cause any issues for BibEntry data. Since everything ran cleanly, with no unexpected errors or broken functionality, the results were simply documented rather than prompting any fixes.

Security Testing

Security testing took a closer look at JabRef's major subprojects, including *jabgui*, *jabkit*, and *jablib*, to get a better sense of how many vulnerabilities the codebase might actually contain. The results were more reassuring than expected. Because JabRef is installed and used locally, the surface area for security risks is smaller, and that likely contributed to the relatively low number of issues found. Most of the concerns that did appear were tied to outdated dependencies rather than direct flaws in the application itself. Since the main takeaway was about package age and developer preference for backward compatibility, the findings were documented rather than prompting immediate changes.

Performance Testing

Performance testing showed that JabRef's startup routine puts a heavy load on system resources, especially when the application is launched repeatedly under stress.

Stress testing was done by using a PowerShell script to force ten rapid startup cycles. The tests measured startup time, CPU usage, and memory consumption. The clearest pattern appeared around the fourth run, where CPU usage climbed to a steady 100 percent and available memory dropped sharply from 2285 MB to 464 MB. Since the stress testing script was run locally, these results are heavily machine dependent. The results also point to a CPU-bound initialization path that becomes more expensive over time. In terms of load testing the team computed metrics by measuring many multithreaded database operations under load. Since addressing this was outside the scope of the project, the behavior was fully documented rather than fixed.

Subsequently, spike testing, implemented using JMeter, allowed a basic spike pattern that ramped up on the JabLib module to execute operations on the BibDatabase. These operations included entry insertions, retrieving strings, and checking the size of the database. The initial normal operations occurred with 20 threads running simultaneously for a good minute, and then a sudden spike was induced by a ramp-up of 30 seconds, with the peak shortly arriving in an explosive spike on the CPU. Although the CPU usage jumped and spiked occasionally, after the peak of 100%, the levels began to dwindle again, thus normal operations resumed.

Significant Issues

Issue 1: High CPU & Memory Load on Repeated Launches

One significant issue identified during the project was the high CPU and memory load that occurs when JabRef is launched repeatedly. Performance testing made this clear, showing that the startup process becomes increasingly expensive with each run.

Since this behavior pointed to deeper bottlenecks in the initialization sequence and wasn't within scope to fix, the issue was fully documented. It matters because these spikes in resource usage can slow down startup responsiveness and compete with other applications for system resources, which ultimately reduces efficiency and stability when the application is opened multiple times in a row.

Issue 2: Mocking Components

Another important issue centered on how mocking is used within the test suite. Mock testing revealed that several components, especially those involving BibDatabase logic handling and the PdfImporter, weren't being mocked even though they should be. This was partly fixed by introducing mocked versions where appropriate, such as a mock entry into the BibDatabase and a mock get size. The broader need for consistent mocking practices was documented. For instance, executing a SearchQuery on the database must also be mocked, but this has not been implemented. The issue matters because relying on real components can introduce side effects, slow down tests, and add unnecessary overhead when database integrations are involved. Improving mocking not only speeds up testing but also helps prevent avoidable operations and keeps test execution time under control. In practice, developers often default to real databases or full components to ensure things work end to end, but incorporating more mocking allows them to avoid rare API or network issues, control test conditions more effectively, and produce more reliable and repeatable results.

Issue 3: High-severity dependency vulnerability

Another major issue involved a high-severity dependency vulnerability uncovered during security testing. Most of the vulnerabilities identified were linked to outdated packages, with four out of five coming directly from dependencies rather than JabRef's own code. Since addressing these updates was outside the project's scope, the risks were documented in the security notes. This matters because a single deprecated dependency can break an entire build. In the actual system, outdated libraries can also increase the chances of crashes, open the door to denial-of-service risks when interacting with public databases, and raise the likelihood of exposing data during online lookups.

Improvements

Improving test coverage was one of the least visible gains in the project. While the percentage increases look small on paper, they came from adding meaningful tests rather than padding the suite. Class, method, line, and branch coverage all rose slightly as new cases were added to previously untested paths. The numbers show careful expansion rather than broad, unfocused testing, and the result is a more reliable baseline for future development.

Mutation testing also saw real improvement. By adding an extra test and tightening existing ones, the mutation score increased from 37 percent to 40 percent, and the number of uncovered mutations dropped noticeably. Line coverage for the mutated classes also rose, and the overall test strength nudged up to 90 percent. The

increase in total executed tests shows that the suite is now appropriately carrying out its job of exercising code that previously went untouched.

Despite the changes being small in scope, code quality improved. Only a few code smells were fixed, roughly fifty lines out of a very large codebase, but they targeted specific problem areas that were causing unnecessary complexity. While the overall percentage shift was minimal, the fixes contributed to cleaner logic and helped reduce long-term maintenance concerns.

The last improvement came from stabilizing the test suite itself. Before the updates, eighteen tests were failing, but after the fixes, that number dropped to fourteen, and the total number of passing tests rose from 681 to 739. Reducing failures while increasing the number of successful test executions shows that the codebase became healthier overall, with fewer fragile or unreliable areas holding the test suite back.

Quality Assessment

Overall, the quality assessment showed that JabRef has a solid foundation, but there are still many risky areas. Due to the complexity of the multi-module project, the project was difficult for our team to fully understand and test. The GUI stood out as the least tested part of the system, which means issues could easily slip through. As a team, we did not run many tests against the graphical user interface. On the other hand, the security and mutation tests proved to be much more valuable. They uncovered weaknesses in outdated dependencies and highlighted gaps in test coverage that the normal unit tests didn’t catch.

From a maintainability and testability standpoint, the project also faced some challenges. Because JabRef is a desktop application built on Java and Gradle, many modern testing or analysis tools did not integrate smoothly, which made it challenging to introduce new testing approaches. The repository included basic documentation for formatting tests, but it does not provide enough detail about what each test actually does. Expanding that documentation would make the project more approachable for new contributors and help maintain consistent testing practices moving forward.

Lessons Learned

One of the biggest lessons the team learned during this project was how important it is to choose the right tools and how important it is to configure those tools correctly. A lot of the challenges we ran into had less to do with the tests themselves and more to do with configuring them. Tools like Snyk, Gradle, or even the Windows stress test script were very difficult to configure correctly in the multi-module project. Getting everything properly set up took more time than expected, but once the configuration issues were out of the way, the testing process became much smoother and the results were far more meaningful. It was a good reminder that tool setup isn't optional overhead, it's part of the testing effort.

Another key lesson was understanding what to test and how to scope those tests effectively. Thinking through expected outputs, identifying risks, and defining a good test oracle all played a role in shaping useful tests instead of just writing more of them. Learning to focus on functionality and clarity helped ensure that tests provided real value rather than noise.

We also learned the importance of early communication and shared responsibility when evaluating tools. Relying on one person to handle setup created delays, especially with tools that needed API keys or extra configuration. A more collaborative approach would have helped us identify working solutions faster and kept the team moving at a steadier pace. This experience made it clear that testing isn't just technical work; it also depends heavily on coordination and shared understanding.

Team Members & Roles

Our team worked collaboratively across all testing stages, with each member taking on responsibilities that aligned with their strengths and the areas they analyzed most deeply.

Geoffrey had a stronger focus on performance testing and tooling setup. His work on identifying the high CPU and memory load during repeated launches played a key role in revealing JabRef's startup bottlenecks. He also led efforts to configure tools like Snyk, Gradle, and the performance scripts, which shaped his lessons learned section about the importance of choosing and setting up tools properly.

Vanessa concentrated on test design, especially around mocking and mutation testing, although she got the setup of SonarQ done. She helped uncover the need for better mocking practices in the components of BibDatabase and SearchQuery, reducing unnecessary side effects during testing. For mutation testing, the FileDirectory component was targeted to kill mutants, as it is vital to know if a path had changed during runtime. Many of the edge cases were handled by relevant test oracles, such as an empty string or an Optional object. Her work emphasized understanding what to test, how to define proper outputs, and how to design reliable test oracles.

Lucille contributed heavily to security testing and team coordination. Her discovery of high-severity dependency vulnerabilities highlighted the risks of outdated packages. She also

recognized the challenges that came from relying on a single person for tool setup, which informed her lesson on improving early communication and evaluating tools collaboratively.

Together, these roles complemented one another and helped shape a more complete understanding of JabRef's strengths, risks, and opportunities for improvement across the entire testing process.