ECSE 429 Project Part C

Name: Mario Bouzakhm ID: 260954086

Fall 2024

1 Project Deliverables Overview

We concentrated on exploratory testing and unit testing of a web application in part A of the project, which we interacted with using CRUD-style API requests. Through dynamic application interaction made possible by exploratory testing, we were able to identify usability problems and edge instances that might not have been expected. At the same time, JUnit was used for unit testing, which assessed the application's separate features separately. Creating separate test cases made sure that every module and method worked as it should have.

Story testing was added in project part B as a way to validate user situations and processes. Utilizing Cucumber, we developed scenarios for behavior-driven development (BDD) that emphasized the viewpoint of the user. In order to ensure that the program fulfilled the objectives and provided end users with a satisfying experience, this method highlighted the need of comprehending user stories.

Non-functional testing, including performance testing and static code analysis, was the focus of this last project phase. We tested the application's performance by measuring its transaction time, memory use, and CPU utilization during API calls under various database loads to make sure it operated at its best across a range of circumstances. To confirm the application's scalability and dependability, this testing was crucial. After that, we used SonarQube to do static analysis and examine the source code. By identifying possible problems with code quality, such bugs, vulnerabilities, and code smells, this procedure eventually helped to make the program more resilient and maintainable. All of these deliverables combined to offer a thorough assessment of the application's functional and non-functional elements.

2 Performance Testing Setup and Results

Performance testing is essential for assessing an application's scalability, efficiency, and dependability under varied operating circumstances. Performance testing makes sure the program can manage real-world usage by spotting any bottlenecks and resource-intensive procedures. By concentrating on measures like as CPU utilization, memory consumption, and transaction durations, developers may enhance user experience, optimize system performance, and lower the chance of failures during high load. We conducted these tests in this project by concentrating on three activities for each metric: adding, editing, and removing database items, especially "todos" and "category" objects, while changing the quantity of objects in the database and the transactions carried out. All API requests were made using the RestAssured library, which offers the developer a great deal of freedom and options, much like in Parts A and B of the project. The outcomes of every test were saved in a CSV file and then plotted using'matplotlib'. The repository and this document's Appendix include all of the graphs.

The system is restarted before and after each test to guarantee that it operates in total isolation and has no impact on the outcomes of subsequent tests. This configuration removes any lingering effects from earlier runs that could have distorted the measurements, including memory leaks or thread congestion. The approach guarantees accurate, consistent, and objective data gathering by executing memory and CPU tests in different threads and restarting the environment for each test. This gives a trustworthy picture of how the system behaves under various loads.

Transaction Time Testing

To determine how effectively the program responds to API requests, transaction time must be measured. System.nanoTime() or System.currentTimeMillis() are used to record the start and finish timings of API calls in order to quantify transaction time. The elapsed time is then computed for varying input sizes in order to track performance trends as workload rises.

The transaction time for adding, removing, and altering items rises nonlinearly with the number of objects, as seen in Figures 7, 9, and 9. The duration increases significantly for bigger inputs (beginning at 5,000 objects), especially for the Todo procedures, which exhibit a more pronounced increase than Category, although it is insignificant for low workloads. This suggests that the procedures do not scale well as the number of items increases. Long transaction durations can lead to resource contention by using memory, CPU, and database connections for prolonged periods of time, impede system resources, and impair user experience by causing bottlenecks. The concept of batch processing, in which each API call is handled separately, may also be added to reduce

transaction time. This could provide some quicker response times, particularly when the system appears to be having trouble with a greater object count. When handling large amounts of data, the use of algorithms with long time complexity becomes crucial and may be a solution to the system's sharp increase in transaction time. Given that the increase was nonlinear, it is likely that the algorithm used a quadratic, cubic, or even more complex algorithm.

Memory Usage Testing

In order to find inefficiencies or memory leaks, memory utilization testing assesses how much memory the program uses while running. Before and after API calls, the Runtime class is used to record the amount of memory that is accessible. The memory used during the API action is provided by the difference between totalMemory() and freeMemory(). To prevent interference from other processes, these tests are carried out in separate threads. Figures 4, 5, and 6 demonstrate how memory usage dramatically increases for Category operations in the beginning, peaks for larger inputs, and then sharply decreases for both Todo and Category as the number of objects rises. This behavior points to either ineffective memory management or potential process memory leaks. The decline after the peak can be a sign of optimization effects at increasing loads or garbage collection. Data should only be held in memory when necessary (loaded at the appropriate moment) and discharged as soon as we are finished using it. We should also use space-efficient algorithms that have a reduced space complexity and refrain from creating additional objects when they are not needed in order to decrease the excessive memory consumption. Given the large amount of data this program handles, it could be interesting to use the garbage collecting process and perhaps modify it to function better with this system at the Java level. The main risk of running memory by running out of space in the heap is still one of the many serious dangers that might result from poor memory management. If not handled properly, this would cause an exception in Java and probably bring down the system.

CPU Usage Testing

Testing CPU utilization aids in identifying an application's processing requirements under varied circumstances. The OperatingSystemMXBean class from the java.lang.management package, which offers system-level metrics like CPU load, is used to track CPU utilization. We discovered resource-intensive situations that potentially put the program under stress, particularly with bigger datasets, by separating the effects of these activities on CPU consumption.

Figures 1, 2, and 3 show that during item creation and modification, CPU utilization rapidly

increases for both Todo and Category, with Category using a disproportionately higher amount of resources. This implies that these procedures, especially for Category, have a larger computational complexity than the delete operation. When compared to creation and modification, the comparatively constant CPU consumption during deletion operations suggests a more straightforward rationale. Optimizing algorithms to avoid duplicate computations and using parallel processing to handle numerous activities simultaneously on separate cores can assist reduce single-core load, which in turn lowers CPU utilization overall. The two main dangers are CPU overload, which would also cause other processes to operate more slowly, and excessive OS scheduler interruptions if the system is taking too long to complete its tasks.

3 Static Code Analysis and Setup

Examining source code for possible errors, weaknesses, and compliance with coding standards without running the program is known as static code analysis, and it is a crucial procedure in software development. It lowers the cost and complexity of subsequent problem-solving by assisting developers in identifying problems early in the development lifecycle. Static code analysis improves security by identifying vulnerabilities like SQL injection and cross-site scripting and guarantees higher code quality, maintainability, and readability by enforcing best practices. Because they offer automated ways to scan code and generate comprehensive data, tools like SonarQube are frequently used for static code analysis, particularly for Java projects. Developers must first install a SonarQube server and integrate it with their build tools, such Maven or Gradle, in order to utilize SonarQube for Java. They may use the SonarScanner program to do the analysis after setting up the sonar-project.properties file with project-specific information. The SonarQube dashboard then makes the results available, providing metrics and actionable insights to constantly enhance the codebase's security and quality. Although there are other ways to launch SonarQube and SonarScanner (the application in charge of carrying out the code analysis), I chose to launch SonarQube locally. At first, the source code's.class files were absent from the repository that was made available. Maven's 'clean' and 'compile' tools were used to handle the generation of these files. The prepared code base was then used to run SonarScanner, and SonarQube running locally on the computer at 'http://localhost:9000/' was used to build and display the report. For this portion of the project, a SonarQube project was made so that I could analyze the code.

4 Static Code Analysis Results and Suggestions

The SonarQube analysis identifies a lot of problems with the codebase, such as medium-priority security hotspots, stability difficulties, and a large amount of code smells (623) as can be seen in Figure 10. The code smells in Figure 14 point to maintainability issues including redundant code, excessive cognitive complexity, and ineffective procedures, which might make future changes more difficult and obscure the code. If left unchecked, security flaws, especially those involving poor cryptography, as seen in Figure 13 (such as the use of pseudorandom number generators), can result in data breaches or attack susceptibility. Overall, it appears that there are several serious problems that require attention, particularly in the areas of dependability and maintainability.

The code was not reviewed by multiple people to remove the unused sections, and some features that were supposed to be implemented were never completed or tracked down, so some features are still missing. SonarScanner also finds a number of sections of the code that contain unused methods and a large number of TODO comments that were left in place. A few small code style problems were also found, such the existence of certain commented-out code blocks and the formation of duplicated variables. In terms of unit/integration testing, there also appears to be a lack of code coverage, which could result in a variety of issues, chief among them reliability issues, as the code being released is not adequately tested, which could help reduce the quantity of errors or bugs we observed with SonarQube.

Refactoring redundant code by adding constants and reusable functions, as well as simplifying difficult procedures to lower cognitive load (some functions have a cognitive complexity of 16, which is greater than the permitted 15), are ways to lessen these hazards. For security concerns, use cryptographically safe substitutes (like SecureRandom) in place of weak pseudorandom number generators. Conduct comprehensive unit testing and examine and correct logical mistakes to address dependability issues. A clean and secure codebase can be maintained by regular reviews and adherence to coding standards. This will also improve system scalability and maintainability, which will assist with redundancy in variables and the possibility of defining and reusing constants rather than using the same variable five or six times. Since there would only be a few issues to fix at each pipeline push rather than hundreds to fix all at once, the introduction of automatic static code analysis through the use of a CI/CD pipeline could help identify these problems earlier and prevent them from piling up and developing bad habits. Additionally, this would enable the developer to promptly address security flaws that expose the system to outside threats prior to deployment. Overall, most of the problems identified by SonarQube would be resolved by following more stringent coding standards and review procedures, in addition to the above-mentioned

security repair recommendations.

5 Conclusion

This project addressed testing, performance, and code quality while offering a comprehensive assessment of the application's functional and non-functional elements. While performance testing found bottlenecks in transaction time, memory, and CPU consumption and provided solutions like batch processing and algorithm optimization, exploratory, unit, and narrative testing guaranteed functionality and user-centric processes. Maintainability and security concerns were brought to light by static code analysis, which emphasized the necessity of organized refactoring and adherence to best practices. The application's performance, scalability, and long-term maintainability will all be improved by putting these suggestions into practice, guaranteeing that it successfully satisfies user and operational expectations.

6 Appendix

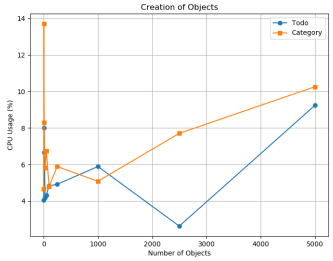

Figure 1: CPU Usage during Object Creation

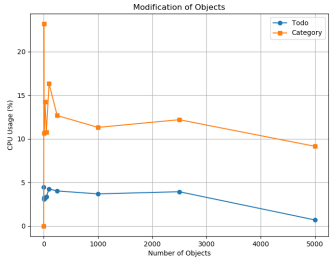

Figure 2: CPU Usage during Object Modification

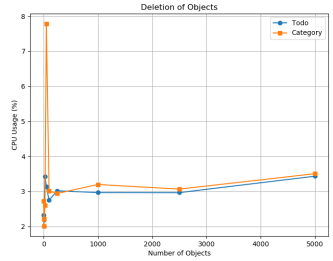

Figure 3: CPU Usage during Object Deletion

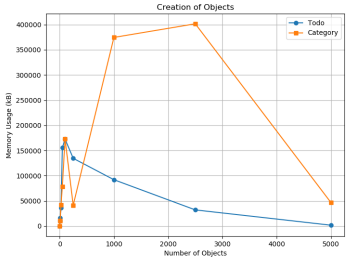

Figure 4: Memory Usage during Object Creation

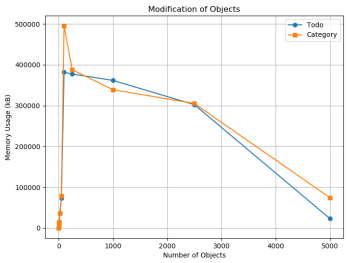

Figure 5: Memory Usage during Object Modification

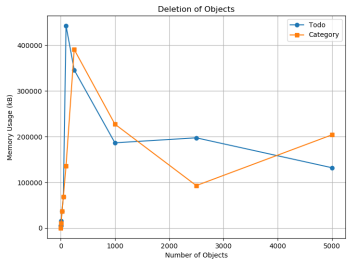

Figure 6: Memory Usage during Object Deletion

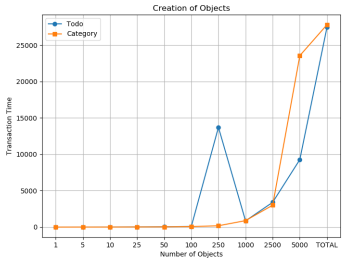

Figure 7: Transaction Time during Object Creation

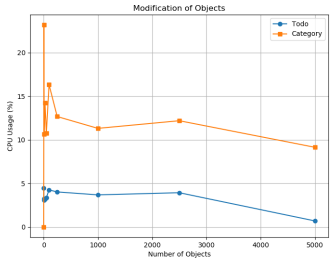

Figure 8: Transaction Time during Object Modification

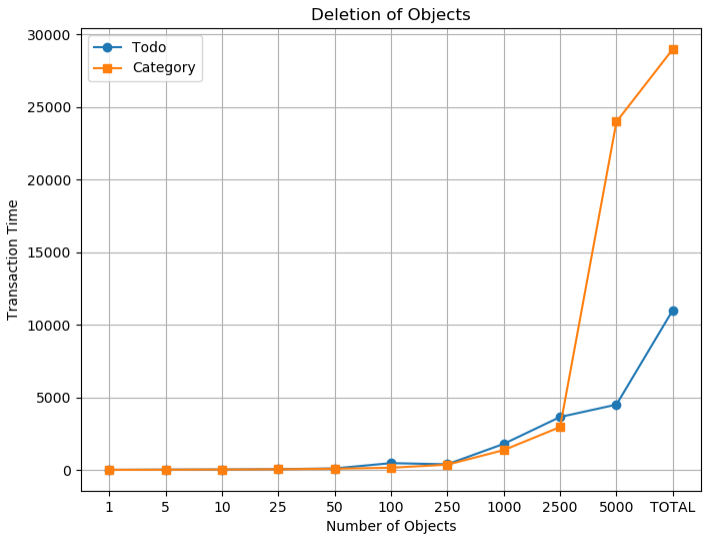

Figure 9: Transaction Time during Object Deletion

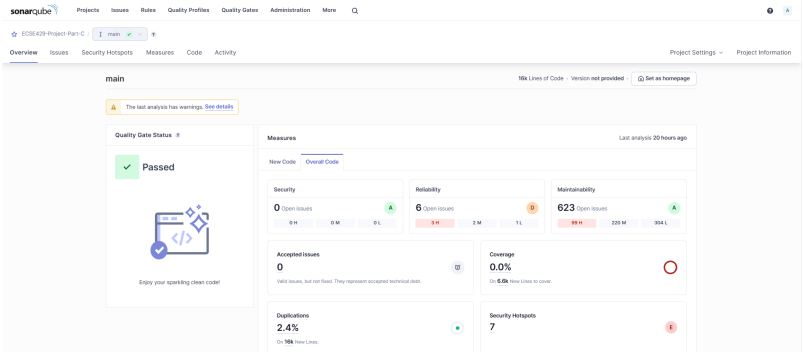

Figure 10: Overall Sonarqube Report

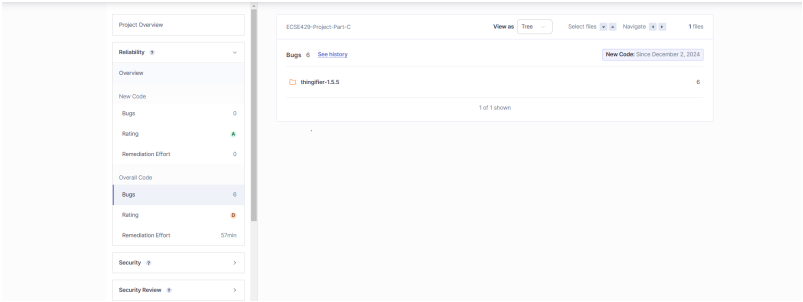

Figure 11: Reliability Issues Identified by SonarQube

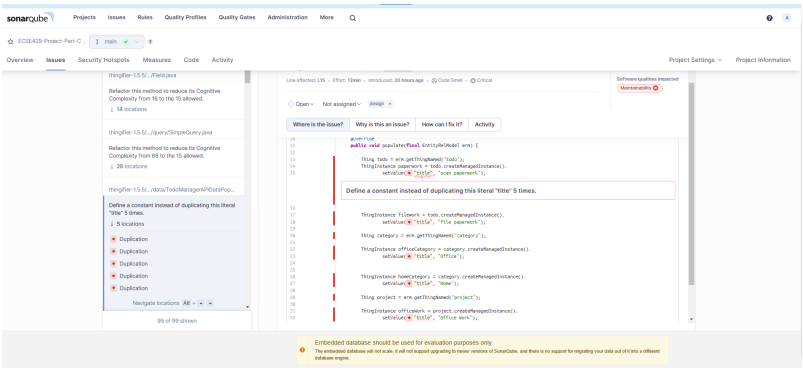

Figure 12: Scalability Issues Identified by SonarQube

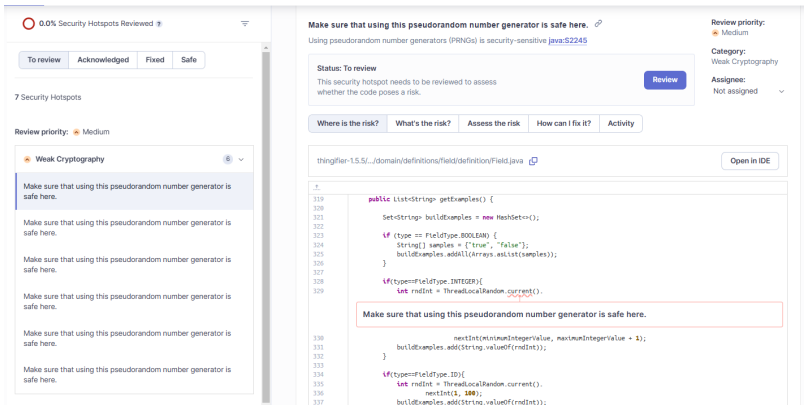

Figure 13: Security Issues Identified by SonarQube with Medium impact

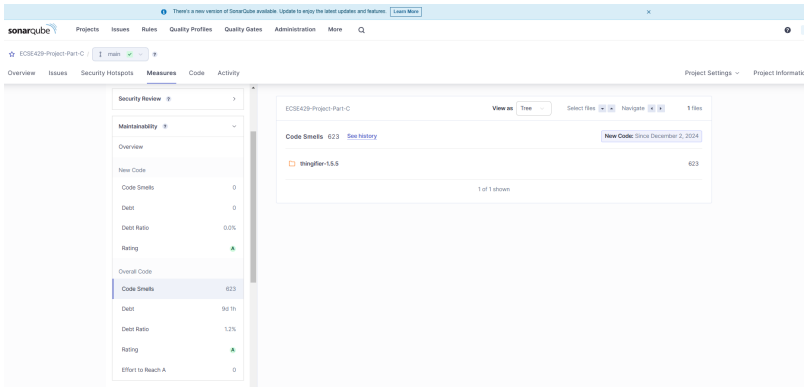

Figure 14: Code Smells Sonarqube Report