Cs-320 Project Two

In Project One, I developed and tested three critical features of the game application: the singleton pattern in GameService, object management in the Game, Team, and Player classes, and the overall integration of these components through ProgramDriver. My testing approach aligned closely with the software requirements, particularly in preventing data duplication and maintaining object identity. For the singleton requirement, I verified that all calls to GameService.getGameService()—whether from ProgramDriver or SingletonTester—returned the same instance. To test entity management, I validated that addTeam() and addPlayer() returned existing instances when passed duplicate names, avoiding redundant object creation. The final layer of testing came through ProgramDriver, which served as a simulation of how the user would interact with the full system, allowing me to confirm not just individual behaviors, but also end-to-end functionality.

Though I did not use external coverage tools like JaCoCo, I leveraged IntelliJ’s built-in code inspection features and careful manual review to measure functional coverage. All major branches of logic were exercised through unit tests. I tested both positive and negative paths—adding new players and teams, attempting duplicates, and verifying that object identity was preserved. For instance, I wrote a test that added the player "Alex" to a team, then added "alex" again with different capitalization. The second call returned the original object, confirming that equalsIgnoreCase() logic was functioning as intended. By structuring each test to target a distinct control path or condition, I minimized the risk of untested branches and strengthened confidence in overall reliability. Based on this analysis, I estimate coverage above 85%.

To ensure technical soundness, I followed a consistent test structure: initialize data, call the method, and assert the result. Tests were isolated and deterministic—no state carried over between tests, preventing false positives. Assertions like assertEquals() and assertSame() ensured both value and identity were validated. For example, I confirmed that calling GameService.addGame("Arena") twice returned the same object, and that calling getGame(0) afterward matched the expected output. I also tested toString() methods, which are often overlooked, to ensure that class data was represented clearly and consistently—especially important for debugging and logging.

Efficiency was also a priority. In the application code, I placed early exit checks in loops—for example, addPlayer() exits immediately when a duplicate is detected. This reduces unnecessary computation in larger data sets. In testing, I ensured efficiency by focusing each test on one behavior and avoiding excessive or redundant checks. I reused helper functions and consistent naming conventions to make the test suite easier to scale. For a future iteration, I would consider using parameterized tests to reduce repetition when testing multiple inputs. These choices reflect broader Java best practices for writing performant and maintainable code (Oracle, 2023).

The testing techniques I used were primarily black-box testing and equivalence partitioning. Black-box testing allowed me to focus on inputs and expected outputs without needing to understand internal implementation details. This was especially effective when testing how the system handled user-provided names. I grouped inputs into equivalence classes: valid entries, duplicates, and invalid cases (such as empty strings). For example, I tested that addPlayer("") returned null or handled the input gracefully, while "Bob" and "bob" returned the same player object. These techniques allowed me to achieve thorough validation while keeping the test suite concise and manageable.

I did not use white-box testing or mutation testing in this project. White-box testing, which involves analyzing internal logic and execution paths, was not necessary due to the simplicity of the code’s control flow. The methods were straightforward and readable, and black-box testing provided sufficient coverage for verifying expected behavior. Mutation testing was excluded due to time constraints and tooling complexity. While I understand its value in strengthening test suites—especially in safety-critical systems or continuous integration pipelines—implementing it would have required additional tools like PIT or Stryker, which were beyond the scope of this assignment.

These choices are consistent with real-world development practices. Black-box testing and equivalence partitioning are widely used in QA workflows for validating API responses, form inputs, and service logic. They allow teams to cover more ground with fewer test cases while still catching common issues. White-box and mutation testing are more valuable in specific contexts—such as developing compilers, financial systems, or medical applications—where it’s critical to validate every possible decision path and confirm that the test suite is truly detecting faults (Geisler, 2021; Stryker, n.d.). As applications mature, incorporating these techniques becomes more important for ensuring test reliability and preventing regressions.

Throughout the project, I maintained a cautious and critical mindset while testing my own code. I made a deliberate effort to challenge assumptions and test for failure, not just success. For example, I tested inputs with unexpected capitalization, spacing, or empty strings to uncover edge case issues. Rather than trusting that my logic was correct, I wrote tests specifically designed to break it. This approach helped me catch subtle flaws—like duplicate logic that was case-sensitive or missing validations that allowed invalid entries.

To avoid personal bias, I treated my implementation as if it had been written by someone else. I reviewed the method signatures and public behaviors independently of the code I had written and approached each test as an outsider might. In some cases, I even returned to the project after taking a break, which helped me spot logic gaps I hadn’t noticed earlier. I also took care to document the intention of each test, making it easier to identify whether a failure indicated a true bug or an outdated assumption.

Discipline was key throughout this process. I never skipped a test just because the method “seemed simple,” and I never ignored a failing test to move forward. Instead, I investigated the cause, corrected the logic, and reran the entire test suite to ensure nothing broke as a result. This habit of treating all tests as first-class citizens—regardless of complexity—ensured that my codebase remained clean and dependable. While I didn’t use test automation or continuous integration tools, I adopted the same habits those systems encourage: fast feedback, early detection, and long-term maintainability.

References

*Code Conventions for the Java Programming Language: Contents*. (2018). Oracle.com. https://www.oracle.com/java/technologies/javase/codeconventions-contents.html

Geisler, D. (2021, November 7). *An Introduction to Mutation Testing [2/3]*. Dietrich Geisler. https://www.cs.cornell.edu/~dgeisler/mutation/testing/2021/11/07/mutation-testing2.html

*What is mutation testing? | Stryker Mutator*. (n.d.). Stryker-Mutator.io. Retrieved June 22, 2025, from https://stryker-mutator.io/docs/