Software Testing 2024: Portfolio

# Outline of the Software Being Tested

The software being tested is a project from the ***Object-Oriented Programming*** course, implementing a grid-based game where two players take turns dropping discs into columns, aiming to connect a specified number of discs in a row horizontally, vertically, or diagonally to win. The project demonstrates key principles of object-oriented programming, such as modular design and encapsulation.

# Learning Outcomes

1. Analyze requirements to determine appropriate testing strategies [default 20%]
   1. Range of requirements, functional requirements, measurable quality attributes, qualitative requirements, …
   2. Level of requirements, system, integration, unit.
   3. Identifying test approach for chosen attributes.
   4. Assess the appropriateness of your chosen testing approach.

插入requirement list

At the early testing stage, we adopted unit, integration, and end-to-end testing to validate key functional requirements such as player turn management, input validation, and win condition determination. This approach ensures comprehensive coverage of the system’s core logic, with parameterized tests and modular drivers enhancing scalability. However, the focus on functional requirements leaves non-functional aspects, such as response time and memory usage, less thoroughly tested.

1. Design and implement comprehensive test plans with instrumented code [default 20%]
   1. Construction of the test plan
   2. Evaluation of the quality of the test plan
   3. Instrumentation of the code
   4. Evaluation of the instrumentation

插入test plan

1. Apply a wide variety of testing techniques and compute test coverage and yield according to a variety of criteria [default 20%]
   1. Range of techniques

**Unit Testing** was employed to validate core functionalities like input validation, player turn management, and win condition determination. This technique allowed for early detection of defects in individual components, such as handling invalid inputs or ensuring correct turn alternation.

**Integration Testing** verified the interaction between modules, such as the NPC logic and board updates, ensuring smooth collaboration between components. For example, integration tests confirmed the NPC followed game rules while selecting moves.

**System Testing** simulated full gameplay scenarios, including board initialization, gameplay, and restart functionality. This ensured the application met user expectations across complete workflows.

* 1. Evaluation criteria for the adequacy of the testing

Key techniques included **code coverage analysis** and **boundary value analysis (BVA).**

**Code Coverage Analysis**  
Code coverage metrics, such as line, branch, and method coverage, were used to assess how thoroughly the tests executed the program, such as unit tests for **F6: Input Validation** and **F3: Full Column Restriction** verified that all valid and invalid input scenarios were covered. Achieving high code coverage minimized the risk of undetected bugs in untested code paths, building confidence in the robustness of critical components.

**Boundary Value Analysis (BVA)**  
BVA was applied to test the system’s behavior at input limits, ensuring robustness under extreme conditions. For **F7: Board Initialization**, tests verified correct functionality for minimum and maximum board sizes. Similarly, **F2: Win Condition Determination** included edge cases, such as a winning move in the final slot, to confirm proper handling of boundary scenarios.

* 1. Results of testing

文本

AI 生成的内容可能不正确。

Figure 1

This application passed 19/20 tests created. Since the NPC in this project can only randomly place discs in the remaining columns without attempting to block the player from winning, the *testNPCLogicalDecision()* did not pass.

* 1. Evaluation of the results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **[Class](file:///Users/angelzhang/Downloads/index_SORT_BY_NAME_DESC.html)** | **Class** | **Method** | **Branch** | **Line** |
| BoardInitializationTest | 100% (1/1) | 100% (2/2) | 100% (4/4) | 100% (8/8) |
| ConnectFourSystemTest | 100% (2/2) | 100% (6/6) | 100% (12/12) | 100% (38/38) |
| FullColumnRestrictionTest | 100% (1/1) | 100% (4/4) | 100% (4/4) | 100% (16/16) |
| GameRestartTest | 100% (1/1) | 100% (4/4) | 100% (8/8) | 100% (21/21) |
| InputValidationTest | 100% (1/1) | 100% (7/7) | 100% (2/2) | 94.1% (16/17) |
| NPCOpponentTest | 100% (1/1) | 100% (5/5) | 66.7% (4/6) | 100% (22/22) |
| PlayerTurnTest | 100% (1/1) | 100% (4/4) | 100% (2/2) | 100% (33/33) |
| WinConditionTest | 100% (1/1) | 100% (6/6) | 100%(7/7) | 100% (42/42) |

The testing achieved **100% class, method, and line coverage**, with most tests reaching **100% branch coverage**, except for *NPCOpponentTest* at **66%**, highlighting gaps in the NPC logic. Unit, integration, and system tests effectively validated key functionalities like board initialization, input validation, and win condition determination.

1. Evaluate the limitations of a given testing process, using statistical methods where appropriate, and summarise outcomes. [default 20%]
   1. Identifying gaps and omissions in the testing process

The testing process had several key omissions. **Non-functional testing**, such as performance and response time validation, was not conducted due to time constraints. **NPC logic testing** achieved only 66% branch coverage, as the NPC cannot block player wins. **Stress testing** for extreme scenarios like maximum board sizes was absent, limiting robustness evaluation.

* 1. Identifying target coverage/performance levels for the different testing procedures

The target for **code coverage** was 100% line and 90% branch coverage, ensuring thorough validation. Functional requirements like **input validation (F6)** aimed for comprehensive edge case testing, while a **response time** target of <100ms (N1) ensured usability.

* 1. Discussing how the testing carried out compares with the target levels

The testing met most targets, including **100% line and method coverage**, ensuring thorough validation of core functionalities like **input validation (F6)**. However, branch coverage fell short at **66%** for NPCOpponentTest, due to the NPC's lack of strategic logic. Non-functional targets, such as a <100ms response time, were not evaluated due to time constraints and limited tools. Stress testing for large board sizes was also omitted.

* 1. Discussion of what would be necessary to achieve the target levels.

**Improving NPC Logic and Branch Coverage**：Enhancing NPC logic to block player wins would add decision paths, increasing branch coverage from 66% to the 90% target. Additional test cases simulating varied board states are essential.

**Performance Testing**：Tools like **JMeter** should evaluate response time (<100ms) and memory usage under realistic loads, ensuring non-functional requirements are met.

**Automated Data Generation**：Using automated tools to create diverse board states and edge-case inputs would improve test coverage, reduce manual effort, and enhance robustness.

1. Conduct reviews, inspections, and design and implement automated testing processes. [default 20%]
   1. Identify and apply review criteria to selected parts of the code and identify issues in the code.

code inspections were conducted to ensure quality. The review focused on critical components, including NPCOpponent logic and InputValidation. Review criteria included logical correctness, adherence to coding standards, and robustness in error handling.Using a checklist-driven approach, issues were identified. For instance, the NPCOpponent logic lacked the ability to block player-winning moves, reducing gameplay strategy.

* 1. Construct an appropriate CI pipeline for the software

For the ConnectFour project, a CI pipeline can automate essential tasks to maintain code quality and functionality. The pipeline would use **GitHub Actions** and include stages for building the project with **Maven**, running **JUnit** tests for core functionalities, and checking code quality and test coverage using **JaCoCo**.

* 1. Automate some aspects of the testing

Automating testing in the CI pipeline improves efficiency and reliability, aligning with principles from textbook Chapter 23. **JUnit** automates unit tests for core functionalities, while integration tests validate system interactions. Regression testing ensures new changes don’t break existing features, with **JaCoCo** tracking test coverage. This approach reduces manual effort, and ensures a robust,

* 1. Demonstrate the CI pipeline functions as expected.

Regularly review and update the CI pipeline configuration to adapt to changes in the development environment, such as new dependencies, updated tools, or changes in project requirements.

Use reporting tools to track test results, coverage, and static analysis feedback, ensuring identified issues (e.g., failed tests or insufficient coverage) are promptly addressed.

Conduct routine assessments to verify that automated testing tools (e.g., **JUnit**) are effectively executing unit and integration tests, identifying issues like logic errors or broken functionality.