Test Plan

Overview

This test plan outlines the comprehensive strategy for testing the ConnectFour application to ensure robust functional and regression validation. It is designed to cover critical gameplay mechanics, including player turn management, win condition determination, input validation, and the NPC opponent's logic. The plan adopts an Agile-compatible approach, emphasizing early and iterative testing to detect and resolve issues efficiently. Through structured scaffolding, such as modular drivers, parameterized tests, and end-to-end testing workflows, the plan provides thorough coverage of functional requirements. Additionally, instrumentation tools like logging, assertions, and automated regression testing are integrated to maintain system integrity and traceability during development and subsequent updates.

1. Priority and Pre-requisites

The functional requirements shall be prioritized because they specify the essential functionality of the ConnectFour application, such as player turn management, win condition determination, and NPC opponent logic. Early detection of issues will reduce the time and resources necessary for Verification and Validation (V&V) activities during Analysis and Testing (A&T). It is critical to incorporate V&V activities, particularly unit and integration-level tests, at the earliest stage of development. This approach fits seamlessly within the chosen Agile lifecycle, as it allows iterative testing and development without requiring all tests to be completed in advance. Non-functional requirements, such as performance and efficiency, are given lower priority as they are not directly concerned with core gameplay mechanics. Thus, fewer resources will be allocated to meet them initially.

Among all functional requirements, **Board Initialization (R7)** and **Input Validation (R6)** serve as the primary Pre-requisites for the functionality of the ConnectFour application. These requirements ensure the proper setup and operation of the game and must be tested thoroughly before any other features are evaluated. To effectively test these requirements, we identify their corresponding **inputs**, **outputs**, and provide a clear **specification**.

a) R7: Board Initialization

This requirement ensures the board is initialized with the correct dimensions and connect-to-win rules based on user input. Proper initialization is foundational to ensure other gameplay functionalities work correctly. Without it, the game's logic would be unreliable.

- **Input:** User-defined board dimensions (rows, columns) and connect-to-win value.

- **Output:** An empty board of the specified dimensions.
- **Specification:**
    1. The board must initialize as an empty grid with the correct size.
    2. Inputs must be validated to ensure compatibility (e.g., connect-to-win ≤ rows and columns).
    3. The game must correctly display the empty board to players.

b) R6: Input Validation

This requirement ensures that all player inputs, such as column selection, are valid. The Partition Principle is used to split this requirement into three further unit requirements: R3: Full Column Restriction, Non-Integer Input Validation, and Negative Numbers Validation. Proper input validation prevents runtime errors and ensures compliance with game rules.

➢ **R3: Full Column Restriction**

Ensures players cannot place a disc in a column that is already full. This prevents game logic errors and maintains game rules.

- **Input:** Column selected by the player or NPC.
- **Output:** A boolean indicating whether the move is valid.
- **Specification:**
    1. The system should reject moves made in a full column.
    2. The game should provide appropriate error feedback for invalid moves.

➢ **Non-Integer Input Validation**

Ensures the game rejects non-integer inputs during gameplay. This prevents invalid data from breaking the game logic.

- **Input:** Player input (non-integer values).
- **Output:** An error message or prompt for re-entry of valid input.
- **Specification:**
    1. The game should reject non-integer inputs during turn prompts.
    2. The game should not proceed until a valid integer input is provided.

➢ **Negative Numbers Validation**

Ensures the game rejects negative column numbers, which are invalid inputs during gameplay. This validation ensures compliance with the game's input boundaries.

- **Input:** Negative column number selected by the player.
- **Output:** An error message or prompt for re-entry of valid input.
- **Specification:**

1. The game should reject negative column numbers during turn prompts.
2. The game should not proceed until a valid positive integer input is provided.

Note: As the primary functionality Pre-requisites, **R7 (Board Initialization)** and **R6 (Input Validation)** must be fully functional to support the implementation and testing of subsequent requirements.

c) **R1: Player Turn Management**

This requirement ensures the alternation of turns between Player 1 and Player 2 (or NPC in single-player mode). It is fundamental to the gameplay mechanics and enforces the flow of the game. Correct turn alternation ensures fairness and adheres to the game's rules. A failure in this functionality could cause severe game logic errors, making other features irrelevant.

- **Input:** Player action (column number for disc placement).
- **Output:** Updated board state showing the player's disc at the correct position.
- **Specification:**
  1. Turns must alternate between Player 1 and Player 2 (or NPC).
  2. Moves are only valid during a player's turn.
  3. The board must reflect the correct player's disc (○ or ×) after a valid move.

d) **R2: Win Condition Determination**

Description: This requirement checks whether a player has met the winning condition (e.g., four connected discs horizontally, vertically, or diagonally). It is essential for the system to accurately declare a winner or draw at the correct point in the game. Any inaccuracies here could lead to unfair results or disruptions in gameplay.

- **Input:** A series of player moves leading to a potential winning configuration.
- **Output:** A game state indicating a winner or a draw (if the board is full).
- **Specification:**
  1. The system must detect winning conditions (horizontal, vertical or diagonal streaks).
  2. A draw is declared when the board is full, and no player has won.
  3. No further moves should be allowed after a win or draw is declared.

e) **R9: NPC Opponent**

Description: This requirement ensures the functionality of playing against an NPC in single-player mode. The NPC must make logical and rule-compliant moves,

ensuring fair and competitive gameplay. Proper implementation of this requirement guarantees that the application functions correctly in single-player mode, maintaining the game's integrity and providing a consistent user experience.

Input: Player's move and the current board state.

- **Output:** NPC's move (a valid column number) and the updated board state.
- **Specification:**
    1. NPC should choose moves that follow game rules (e.g., no moves in a full column).
    2. NPC should prioritize blocking a player's winning move or achieving its own victory.
    3. The system must handle scenarios with minimal delays for NPC response.

f) **R4: Game Restart Option**

This ensures that players can restart the game with a clean board after a game session ends. Proper restart functionality is crucial for reusability during testing or replaying.

Inputs:

- Player input indicating a restart request.

Outputs:

- A re-initialized board with default dimensions or custom specifications.

Specification:

- The game must reset all board states and variables (e.g., player turn) when the restart option is selected.

2. Scaffolding and Instrumentation

Scaffolding and Instrumentation establishes the foundation for testing the ConnectFour application by integrating modular drivers, parameterized tests, and system-level validations. It aims to automate testing workflows, ensure comprehensive coverage, and validate the robustness of key functionalities, such as player turn management, input validation, win condition determination, and game restarts. By incorporating logging, assertions, and the testing framework ensures traceability and resilience against invalid inputs. Additionally, leveraging an automation framework with CI/CD integration enables efficient and repeatable regression testing to maintain the integrity of core functionalities amidst future updates.

Below provides a more detailed scaffolding and instrumentation for ConnectFour.

Scaffolding

1. **Test Drivers**:Automate testing for functional requirements and simulate user actions.
    a) Modular Drivers:
        - **PlayerTurnTest**: Automate validation of Player Turn Management (R1).
        - **InputValidationTest**: Use unit test to ensure robust validation of inputs such as column bounds, invalid characters, and full columns (R6).

    b) **End-to-End Driver**: A system test to simulate a full game, covering:
        - Board Initialization (R7)
        - Input Validation (R6)
        - Win Condition Determination (R2)
        - NPC Opponent Behavior (R9)

2. **Parameterized Tests**: Validate system behavior under different configurations and edge cases.
    - Use parameterized tests for Board Initialization (R7), ensuring compatibility across varying board sizes and win conditions (e.g., Connect 3, Connect 5).

3. **Regression Testing Support:** Ensure that core functionalities (e.g., game restart) remain unaffected by future code updates or modifications.
    - Utilize the test cases in GameRestartTest to simulate a complete game restart workflow.
    - Verify that player state variables and the game board are correctly reset after the restart.
    - Simulate new game actions post-restart to validate that the operational flow is restored to normal.

4. **Automation Framework**: Enable automated, repeatable testing.
    - Utilize JUnit 5 to manage test cases, ensuring compatibility with CI/CD pipelines (e.g., Maven).

Instrumentation

1. **Logging:** Track game flow and debug issues during test execution.
    - Integrate System.out.println() for immediate logging of key events (e.g., player moves, NPC decisions, board state updates).
    - Replace with a structured logging library (e.g., SLF4J) in later development stages for enhanced traceability.

2. **Assertions:** Validate correctness of system behavior.

- Use JUnit assertions (e.g., assertEquals, assertTrue) to validate outputs at every critical step of testing.
- Examples include verifying win conditions, board state after each move, and turn alternation.

3. **Error Handling Validation:** Ensure robustness against invalid inputs.
   - Simulate invalid input scenarios, such as out-of-bound moves, full columns, and non-integer inputs.
   - Add specific tests for edge cases like resetting the game state after invalid actions.

4. **Coverage Analysis:** Measure test completeness and identify gaps.
   - Use IDE-integrated tools (IntelliJ) to monitor class, method, line, and branch coverage.

Evaluation of the instrumentation and Scaffolding

Strengths: The instrumentation and scaffolding implemented in the test plan are comprehensive and robust, effectively covering critical functional requirements such as player turn management, input validation, win condition determination, and game restart functionality. Logging provides immediate feedback for debugging, while JUnit assertions validate the correctness of system behavior at critical stages. Regression testing ensures core functionalities remain stable during iterative development, and error handling tests safeguard the system against invalid inputs. Additionally, the integration with IntelliJ's coverage tools and JUnit 5 ensures compatibility with CI/CD pipelines, facilitating seamless automated testing.

Areas for Improvement: While the instrumentation is well-rounded, it could be enhanced by incorporating dynamic logging for better traceability, stress and performance testing for non-functional requirements, and more in-depth validation of NPC opponent behavior. Expanding edge case exploration, such as testing maximum board sizes and overlapping win conditions, would improve coverage. Scenario-based tests could provide insights into real-world gameplay interactions, and automated test data generation would streamline testing by creating diverse board states. Addressing these gaps would strengthen the framework's ability to comprehensively test both functional and non-functional requirements.

3. Process and Risk

Process

The **Process** outlines the structured approach to testing the ConnectFour application based on a lifecycle methodology. This ensures that testing is seamlessly integrated

into the development process, enabling comprehensive coverage, early issue detection, and robust verification of core functionalities.
The following phases define the testing process:

1.Planning Phase

- Prioritize functional requirements such as Board Initialization (R7), Input Validation (R6), Player Turn Management (R1), Win Condition Determination (R2), and NPC Opponent (R9).
- Design test cases for critical pre-requisites (e.g., R7 and R6) to ensure early validation of foundational functionality.
- Allocate resources for modular drivers, parameterized tests, and system-level validations.

2. Design Phase

- Implement modular drivers for unit testing key functionalities:
  - **PlayerTurnTest**: Validate R1 (Player Turn Management).
  - **InputValidationTest**: Ensure robustness of R6 (Input Validation).
- Design parameterized tests for **R7 (Board Initialization)** to support varying configurations.
- Create an **end-to-end system test** to validate the interaction between R7, R6, R2, and R9.

3. Execution Phase

- Run unit tests (e.g., InputValidationTest, FullColumnRestrictionTest) during the development of individual modules.
- Conduct integration tests to validate interactions between modules (e.g., NPCOpponentTest for R9).
- Execute the **end-to-end system test** to simulate a complete game, including gameplay, win conditions, and restart scenarios.
- Use assertions (e.g., assertEquals, assertTrue) and logging to validate correctness and trace issues.

4. Monitoring and Validation Phase

- Monitor test coverage using IDE-integrated tools (e.g., IntelliJ's coverage analysis).
- Validate that all functional requirements are covered by existing tests and address gaps as needed.
- Regularly update and re-run regression tests (e.g., GameRestartTest) to verify that core functionalities remain intact after code changes.

5. Feedback and Refinement Phase

- Review test results and address any failures or anomalies.
- Refactor and optimize test cases to enhance maintainability and reusability.
- Document issues and resolutions to guide future development and testing cycles.

Risks

Resource constraints and limited time may lead to delays in testing or a compromise in test quality. To mitigate this risk, it is essential to prioritize testing critical functionalities, including Board Initialization (R7), Input Validation (R6), and Win Condition Determination (R2), ensuring the most vital features are thoroughly tested within the available resources.

mistakes in writing or maintaining test cases, can result in false positives/negatives or overlooked bugs. To mitigate this risk, it is crucial to implement descriptive logs and structured assertions, which facilitate easier debugging and ensure that potential issues are identified and resolved efficiently.

Evaluation of the quality of the test plan

The test plan demonstrates a robust and well-thought-out framework for testing the ConnectFour application, encompassing functional and regression testing through structured scaffolding and instrumentation. By prioritizing critical functional requirements, such as Board Initialization (R7), Input Validation (R6), and Win Condition Determination (R2), the plan ensures that core gameplay mechanics are thoroughly validated. The use of modular drivers, parameterized tests, and an end-to-end testing approach contributes to comprehensive coverage and compatibility with Agile development practices.
However, he plan lacks explicit provisions for performance, stress, and usability testing, which could be critical for scalability and user experience. While descriptive logs and structured assertions are mentioned, it does not include specific strategies for maintaining test cases in response to evolving requirements.