**20240825 Jomas Benfell McCormick**

**Project Assignment 3**

1. Introduction

This report is for the Testing aspect within the back end of the Bowling Game, it presents all plans, strategies, unit testing, bugs identified, bugs fixed, and the implementation of the debugging tools. The tests are designed to ensure that the problem is operating properly as per its functional and non-functional requirements. A methodical approach with a focus on modularity was created to identify any issues or bugs, validate expected outcomes as well as fix any unexpected outcomes, documenting any issues encountered and the fixes that were applied.

The testing process focused primarily on verifying the accuracy of the score calculations, the handling of strikes and spares, frame progression, and edge-case scenarios that would typically not be expected such as invalid inputs or rare scenarios such as scoring a perfect game. Debugging tools were used to check for inconsistences or run-time errors. This report outlines the entirety of the testing lifecycle, focused on the flow of the backend system and the testing methods used were designed to ensure reliable testing.

1. Test Plan

2.1 Test Objectives

The main objectives of testing the tin-pin bowling game prototype are to:

* Verify that the game correctly calculates scores based on the standard rules of bowling
* Ensure the program can handle all core gameplay functions, including strikes, spares, and the bonus turns in the 10th frame.
* Identify and document any bugs or errors that exist in the current implementation
* Identify how the system handles any edge cases, such as games with all strikes or all misses.
* Evaluate the reliability of the code through testing coverage.
  1. Test Strategy

An in-depth multi-layer testing approach will be used:

* Unit Testing: Core functions (Scoring, strike/spare detection, and bonus score calculations)
* Scenario Testing: Common scenarios that may occur such as perfect games, all spares, and a mix of both will be tested against expected outcomes.
* Edge Case Testing: Unusual cases that may occur (All gutter balls, incomplete games)
* Manual Testing: Reviewing example outputs will help by being able to observe real-world behavior and verify that the expectations are being met.
  1. Test Environment

The following environment will be used during testing:

* Ryzen 7 5800X Computer with 32GB RAM.
* Operating System: Windows 11 Home
* Python Version: 3.12
* Testing Framework: unittest
* Development Tools: VS Code, Git, CLI  
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
  1. Risk Assessment

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Risk | Probability | Severity | Impact | Mitigation |
| Incorrect Scoring Logic | High | High | Inaccurate score, fails to meet the rules of Bowling | Create detailed unit tests that allow us to validate scoring during each frame. |
| Incorrect 10th Frame Logic | High | High | Final frame bonus logic may be incorrect or entirely skipped | Design specific tests for all 10th frame conditions (strike, spare, open) |
| Incomplete test coverage | Medium | Medium | Bugs may go undetected in untested areas | Write tests for all common and edge cases. |
| Crashes due to unexpected inputs | Low | Medium | The system may crash unexpectedly. | Validate inputs, ensuring only valid pins (0-10) can be input. |
| Bugs hidden in working scenarios | Low | Medium | Test cases may pass even if the logic is faulty. | Write test cases for not only expected passing but also expected failed test cases. |
| Version control Issues | Low | Medium | Could either lose work or otherwise create a very confusing version history of the code. | Commit frequently, have descriptive messages, and push to a remote repository often. |

Table 2: Risk Assessment Matrix

1. Unit Test Design

3.1 Test Suite Structure

The unit tests for the bowling game are contained within a single Python file, test\_bowling.py. The test suite is structured using Python’s built-in unittest framework. The structure is designed to validate the core functionality of the BowlingGame class. Each test is contained within a sub-class and is inherited from unittest.TestCase.

The structure includes:

* A setUp() method which is run before each test, creating a fresh instance of the BowlingGame class.
* A helper method roll\_many(), allowing rolling the same number of pins multiple times, reducing repetitive code.
* Individual test methods that focus on different gameplay scenarios, such as: gutter games, all ones, all spares, all strikes, and regular frames.
* Each test uses the assertEqual() method to compare the expected score against the actual result returned by the score() method of the game.

By using a modular structure like this, it allows for easy implementation of additional test cases during debugging and development.

* 1. Test Cases

Key test cases include:

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Purpose | Input (rolls) | Expected Output |
| test\_gutter\_game | Test score for a game with all gutter balls | [0] \* 20 | 0 |
| test\_all\_ones | Test score for a game with 1 pin knocked down every roll. | [1] \* 20 | 20 |
| test\_perfect\_game | Test score for a perfect game (12 Strikes) | [10] \* 12 | 300 |
| test\_all\_spares | Test 10 frames of 5+5, and one bonus 5. | [5] \* 21 | 150 |
| test\_regular\_game | Test a mixture of normal frames, without strikes or spares | [3, 4, 2, 5, 1, 6, 4, 2, 8, 1, 7, 1, 5, 3, 2, 3, 4, 3, 2, 6] | 72 |
| test\_mixed\_game | Test a regular game, except with strikes and spares for bonus testing | [10, 3, 6, 5, 5, 8, 1, 10, 10, 10, 9, 0, 7, 3, 10, 10, 8] | 190 |
| test\_spare\_bonus | Test that the bonus for spares adds to next roll’s pins | [5, 5, 3] + [0] \*17 | 16 |
| test\_strike\_bonus | Test that the bonus for strike adds next two rolls’ pins | [10, 3, 4] + [0] \* 17 | 24 |
| test\_double\_strike | Test correct scoring when there have been two strikes in a row | [10, 10, 4, 2] + [0] \* 16 | 46 |
| test\_10th\_frame\_bonus | Test correct bonus rolls in 10th frame | [0] \* 18 + [10, 10, 10] | 30 |

Table 2: Key Test Cases with Expected Results

* 1. Test Data

The test data used for this project was created manually, to simulate a variety of realistic as well as edge-case gameplay scenarios. This allows us to evaluate the scoring accuracy across different conditions. The rolls were input using the roll() method in the BowlingGame class.

The key types of test data include:

* Basic Valid Cases
  + All Gutter Balls
  + All ones
* Bonus Score Scenarios
  + One spare followed by a roll
  + One strike followed by two rolls
  + Two strikes in a row, followed by regular rolls
  + A perfect game
  + All spares with a final bonus roll
* Regular Game without Bonuses
  + A game where there are no strikes or spares, thus no bonus points in this testcase.
* Realistic Game
  + A game with a mix of regular frames, spares, as well as bonus scenarios to test a variety of scenarios that would typically appear in a normal game.
* 10th Frame Bonus Handling
  + Rolls that test the 10th frame bonus case (Extra turn(s) if strike or extra turn if spare)

1. Debugging Implementation

4.1 Debugging Methodology

To identify and correct issues in the bowling game, a structured debugging methodology was used. The debugging process follows these key steps:

1. Reproduce: Test scenarios using known scoring examples (e.g. all gutters, all spears, all strikes) which are used to consistently reproduce potential bugs in the scoring logic.
2. Isolate: Going through the code and creating targeted print statements are used to isolate where the bug is occurring (e.g. incorrect bonus calculations for strikes)
3. Analyze: The scoring method is then examined to understand the logic surrounding how bonuses were applied to see what went wrong.
4. Fix: Adjusting the scoring implementation based on findings, making sure that the changes address the bugs that had occurred.
5. Verify: Validate each fix via Unit Tests (e.g. test\_bowling.py) and manual testing via example\_usage.py.
6. Document: Document the identified bugs, test results, and implemented fixes, which results in both bug tracking, as well as refactoring in the future.

4.2 Debugging Tools

Tools being used throughout the debugging process:

* Print Statements
  + By extensively using print statements inside the score() method, we can monitor the values of frame\_index, current rolls, as well as the total score after each frame.
* Python unittest Framework
  + Unit tests are run using test\_bowling.py to verify common bowling scenarios, such as all gutter balls, all ones, all spares, and perfect games.

Example of debug print insertion in the score() method:

|  |
| --- |
| print(f"Frame {frame\_index}, Rolls: {self.rolls[frame\_index:frame\_index+3]}, Total so far: {score}") |

4.3 Identified Bugs

During testing, several critical bugs were identified, one of the primary ones was that the 10th Frame was being ignored in the code, due to this every game resulted in one turn being missed.

|  |  |  |  |
| --- | --- | --- | --- |
| Bug Description | Explanation | Discovery Method | Severity |
| Strike Bonus only adds 20, instead of 30 in a perfect game | For perfect games it only returns 270, when it should be 300. | Test\_perfect\_game failed | High |
| Spare bonus not adding to the next roll | Test\_all\_spares returns 135, when it should be 150. | Test\_all\_spares failed | High |
| Bonus rolls in the 10th frame aren’t counted | Final frame strike bonus scores 0, instead of 30. | Test\_strike\_bonus failed | Medium |
| Mixed game with spares/strikes gives wrong overall score | Score returned 155, when it should be 190. | Test\_mixed\_game failed | High |
| Regular frames score half the value they’re supposed to | A normal game with no strikes/spares scores 36, instead of the expected 72 | Test\_regular\_game failed | Medium |
| Game with all 1s returns 9, instead of 20. | Frame scoring logic seems to either skip rolls, or frames. | Test\_all\_ones | High |
| Two consecutive strikes not correctly scored | Second strike bonus was miscalculated: was calculated as 44 instead of 46. | Test\_double\_strike failed | High |
| Single strike bonus calculated incorrectly | Score returned 20, when it should be 24. | Test\_strike\_bonus failed | Medium |

4.4 Bug Fixes

The following fixes were implemented with detailed debugging insights:

1. **10th Frame being Ignored**

**Fix:** The loop in the score() method only proceeded 9 frames instead of 10. The final frame, which can include up to 3 rolls thanks to the bonus rolls, was not being counted or ran at all. This has been fixed by adjusting so that the frames will be 10 instead of 9, as well as explicitly handling the logic of the 10th frame thanks to the bonus rolls for a strike or spare that can occur.   
  
**Debugging Insight:** During testing with testing\_perfect\_game, test\_all\_spares, test\_strike\_bonus and test\_mixed\_game, it was observed that the total score was 270 instead of 300. By checking the manual logs, it was able to be shown that there were only 9 frames, at which point the programmed stopped running, resulting in the final frame being skipped entirely, including its bonus rolls (if any).

1. **Strike Bonus only adds 20 instead of 30 points in a perfect game.**

**Fix:** The \_strike\_bonus method was not handling strikes in the 10th frame. It added the two rolls on the 10th frame, however it failed to count the bonus rolls within the 10th frame. By updating the code it now includes all bonus rolls for the final strike.

**Debugging Insight:** In test\_perfect\_game, a score of 270 was returned instead of the expected 300. By manually inspecting the logs it was noted that the two bonus rolls after the final strike were being completely ignored in the score, although it was being recorded. Upon updating the code so that it accounts for the bonus rolls, it now correctly sets the score to 300 instead of 270.

1. **Strike Bonus not added to the next roll**

**Fix:** The \_spare\_bonus method was attempting to use rolls[frame\_index+2] at all times, including the 10th frame (once the fix was applied), where the logic differs for bonuses compared to other frames. After adjusting the function, it now correctly retrieves the next roll, even during the 10th frame when a spare is scored.

**Debugging Insight:** In test\_all\_spares, the total scored returned was 135, instead of the expected 150. By checking the logs, it was revealed that the 21st roll (Bonus roll for the last spare) wasn’t being added to the score, despite being in the list.

1. **Bonus Rolls in the 10th frame weren’t counted.**

**Fix:** The original loop code was treating the 10th frame like the rest of the frames, which resulted in it being impossible to have up to 3 rolls in the frame. This was fixed by adding a condition that specifically handles the bonus logic in the 10th frame, based on whether there is a strike or a spare.

**Debugging Insight:** test\_strike\_bonus and related tests were showing that a strike in the 10th frame only added 10 points to the total score. By checking the logs it was confirmed that while the rolls were present in the data, they weren’t being added into the total score due to the loop closing earlier than intended.

1. **Mixed game with spares/strikes giving an incorrect score.**

**Fix:** The combination of previous bugs caused an incorrect handling of bonus scares across several frames, especially the 10th. Once all of the frame logic (strike/spare/open) was corrected alongside the 10th frame bonuses being handled in its own way, the mixed game score became accurate and fixed this bug.

**Debugging Insight**: test\_mixed\_game consistently returned 155 or 183, purely depending on whether the 10th frame bug was addressed or not. Thanks to this, I was able to identify the multiple bugs that were compounding that caused this issue, which were due to open frame miscalculations, incorrect handling of strike bonuses, as well as the missed logic for the 10th frame.

1. **Regular frames score half the value they’re supposed to.**

**Fix:** The scoring method for open frames was originally only adding the first roll of each frame. The code was updated to sum both rolls of each frame by using rolls[frame\_index] + rolls[frame\_index +1], causing the sum to combine to a total score per frame.

**Debugging Insight:** In test\_regular\_game, the score sequence should have returned 72, however it returned 36. The logs confirmed that only one roll per frame was being summed, cutting the frame’s overall value in half. By fixing this, the frame scoring is now accurate.

1. **Game with all 1s returns a total score of 9 instead of 20.**

**Fix:** An error in the code caused both rolls as well as one frame to be skipped which was caused by incorrect frame indexing. By adjusting the loop and ensuring that all frames and rolls were counted corrected this issue.

**Debugging Insight:**

Test\_all\_ones showed a return value of 9, instead of the expected 20. Checking the logs revealed that several rolls were being ignored due to an early loop exit, as well as one case of an incorrect frame\_index increment. This was corrected by refining the frame progression logic, counting both scores of each frame, and including the final turn.

1. **Two consecutive strikes were not correctly scored.**

**Fix**: When two strikes occurred in a row, the strike bonus only counted for one roll, which is wrong as it was supposed to have included the second strike and its following roll. The \_strike\_bonus method was fixed, allowing for consecutive strikes to work appropriately.

**Debugging Insight:** In test\_double\_strike, the total was 44, as opposed to the expected 48. By viewing the code and figuring out how the bonuses were calculated, I noticed that the second strike’s next roll was not being properly counted as part of the first strike’s bonus, causing this unintended bug.

1. **Single strike bonus calculated incorrectly.**

**Fix:** The strike bonus logic was missing the next two rolls at times due to incorrect indexing. This was fixed by ensuring that both rolls[frame\_index +1] and rolls[frame\_index +2] were working as intended.

**Debugging insight:** In test\_strike\_bonus, the total was 20 instead of 24. Logs show that it was only the strike and one bonus roll being counted, by fixing the indexing and the range check, proper scoring was once again restored and working as intended.

5 Code Refactoring

**5.1 Refactoring Approach**

The code refactoring process followed these core principles:

* **Less is More:** Instead of rewriting chunks of code, sometimes the smaller, more thoughtful changes can simplify the code, reducing bugs. Cleaner code also often removes what’s unnecessary – meaning its not adding additional complexity and prioritizes optimization.
* **Single Responsibility:** Each method does a single task that it’s responsible for. This is important because it not only improves readability but makes the codebase more modular, resulting in a code that is easier to maintain or add additional features to in the future.
* **Minimize Duplication:** There should not be any repeated code, for anything that has duplications it should be under a single method that is reusable and can be called for on demand. Reducing duplication lowers the chance of bugs, especially in the case of copy/pasting with different indentation or not copy/pasting correctly and makes future updates more efficient.
* **Refactor Confidently**: Changes were made with constant validation with the existing unit tests for every chance to ensure it continued to work as intended, if not better than intended, which results in improved performance.

**5.2 Refactoring Changes**

1. **Added Logging for Debugging and Transparency**

**Before:** Before adding, there was no logging allowing us to trace how the game ran and the numbers that were selected per roll.

1. def roll(self, pins):

2. self.rolls.append(pins)

3. self.current\_roll += 1

4.

**After:** Debug logging was implemented, allowing us to track all roll actions and frame outcomes for testing purposes.

1. def roll(self, pins):

2. self.rolls.append(pins)

3. self.current\_roll += 1

4. logging.debug(f"Roll {self.current\_roll}: {pins} pins.")

**Reason for the Refactor:** This being refactored makes it easier to identify where the code may or may not fail and verify that the information it displays is correct while testing without requiring temporary print statements.

1. **Corrected Scoring Loop for the 10th Frame**

**Before:**

1. for frame in range(9):

**After:**

1. for frame in range(10);

**Reason for the Refactor:** The original loop only calculated 9 frames, which meant the final frame was not included; additionally, that included any potential bonus rolls that can occur in the final frame. This fix ensures that the game is scored correctly.

1. **Improved Bonus Calculations with Checks to ensure it is correct.**

**Before:**

1. def \_strike\_bonus(self, frame\_index):

2. return self.rolls[frame\_index + 1] + self.rolls[frame\_index + 2]

3.

**After:**

1. def \_strike\_bonus(self, frame\_index):

2. bonus = 0

3. if frame\_index + 1 < len(self.rolls):

4. bonus += self.rolls[frame\_index + 1]

5. if frame\_index + 2 < len(self.rolls):

6. bonus += self.rolls[frame\_index + 2]

7. return bonus

8.

**Reason for the Refactor:** Without the correct checks, the original version could crash via an IndexError if the rolls that existed were lower than what was intended. By updating this, it now handles both partial games (if a game is incomplete), as well as incorrect input.

1. **Expanded Unit Test Coverage**

**Before:** Only basic tests were provided, and the logic had no modularity.

1. def test\_gutter\_game(self):

2. self.roll\_many(20, 0)

3. self.assertEqual(0, self.game.score())

**After:** Added multiple new test cases, and increased modularity.

1. def roll\_sequence(self, sequence):

2. for pins in sequence:

3. self.game.roll(pins)

4.

5. def test\_mixed\_game(self):

6. rolls = [10, 3, 6, 5, 5, 8, 1, 10, 10, 10, 9, 0, 7, 3, 10, 10, 8]

7. self.roll\_sequence(rolls)

8. self.assertEqual(190, self.game.score())

**Reason for the Refactor:** By adding new costs, we ensure that we have covered edge cases, such as perfect games, spares, strikes, as well as the 10th frame bonuses.

1. **Added clear and modular docstrings for each method**

**Before:** The documentation was extremely minimal, and many of the methods have no documentation at all.

**After:** Each method now includes a docstring that explains its purpose, arguments, and return values. Additionally, it is easy to view via a .html that is much like many of the available documentation websites out there (such as docs.python.org.) which as the project evolves, will make it easier for the project to expand and be understandable between teams.

1. def \_is\_spare(self, frame\_index):

2. """

3. Checks if the rolls at both frame\_index and frame\_index + 1 form a spare

4. (if the total number of pins knocked down is equal to 10, it is a spare).

5.

6. Args:

7. frame\_index (int): The index of the first roll in the frame.

8.

9. Returns:

10. bool: True if the rolls form a spare (pins knocked down equals 10), else False.

11. """

**Reason for the Refactor:** This improves readability, makes it easier to maintain for both present and future developers, and is easily accessible via web browser.

* 1. **Refactoring Challenges**
* **Ensuring Backwards Compatibility:** 
  + By adding new features, such as logging as well as adjusting the scoring calculations, it introduced the risk of changing existing behavior or existing code too much that would result in it being unrecognizable. To ensure this didn’t happen, I ensured that the unit tests were consistently run between changes to ensure that no existing functionality had not changed.
* **Handling Edge Cases**
  + Edge cases are often unpredictable, it was important that I implemented safety checks such as the bonus roll calculations in \_strike\_bonus, which required that while actively testing that I didn’t run into any IndexError issues regardless of how far the game has progressed.
* **Balancing Logging with Readability**
  + While logging was an amazing help with debugging, it was a hard balance to figure out the difference between being easy to read and being excessive or hard to read due to how much information is being spat out. By coding it so that they are all debug level logs, it can be toggled off without requiring any change in the primary code and means it doesn’t get stuck inside the normal execution of the program.

Although there were challenges, they were small in the refactoring process thanks to the test-driven validation as well as the modular codebase making it easy to adjust and modify as required.

**Summary Report**

In Summary, this report documented the completed user testing and maintenance process of the bowling game prototype. A structured test plan was created to assess the state of its functionality, edge case handling, and how robust and modular it is. The unit testing covers the general scenarios you would expect while playing, as well as rarer cases such as consecutive strikes and spares, invalid input handling, and even a perfect game with the whole game having strikes.

Of the testcases, there were 8 bugs identified and fixed throughout the process. Each bug was found via the test cases and then fixed with all fixes being applied to the code.

All the code was then annotated using PythonDoc, improving readability and understanding, which as a result with help with maintaining the project in the future. Version control via Git was used to track the development and changes. Refactoring primarily focused on Unit test coverage and additional checks in case of errors, ensuring that the calculations remain correct, and the software runs as intended.

The game now runs as expected with the score and handles a variety of gameplay scenarios reliably. The final system is stable, and functions as intended in a normal user state.  
  
I recommend that this project continues by doing the following:

1. **Add a Front-End Interface:** Now that the back end is fully tested, I recommend developing a easy to use UI. This will improve the user interaction and improve accessibility for the player making the game more appealing for the end user.
2. **Introduce Automated Testing in the CI/CD Pipeline:** To make it easier and streamline updates in the future with ongoing stability, I believe it would be worth adding a CI/CD pipeline, using tools such as GitHub Actions or GitLab CI, incorporating the automated testing will help catch any possible issues earlier on during development.