Master of Information Technology (Software Engineering)

Course Code: PRT582

Course Title: Software Engineering: Process and Tools

Project Title: Software Unit Testing Report

Submitted To:

Charles Yeo

Senior Lecturer - Information Technology

Submitted By:

Name	Student ID
Anupoma Angasree Toma	S394044

Date of Submission: 5th of September 2025

Table of Contents

Introduction:

Hangman is a classic word-guessing game that has been popular for decades. The main objective of the game is for the player to guess a hidden word or phrase by suggesting letters, one at a time. It's a word or phrase guessing game, where the word to guess is represented by a row of dashes representing each letter of the word. If the guessing player suggests a letter which occurs in the word, the other player writes it in all its correct positions. If the suggested letter does not occur in the word, the other player adds one element of a hangman diagram. Generally, the game ends once the word is guessed, or if the hangman figure is complete.

The main objective of this project is to design and implement a Hangman game in Python while applying Test-Driven Development (TDD) principles. The game aims to be interactive, user-friendly, and fully functional with the following key goals:

1. **Develop a Functional Hangman Game:** Implement a fully working Hangman game in python that allows users to guess letters, display the hangman diagram, and keep checking on the game progress.
2. **Enhance Programming Skills:** Apply core programming concepts such as loops, conditionals, string manipulation, and lists, dictionaries.
3. **Implement Unit Testing:** Use automated unit testing to ensure each component of the game functions as intended. This will improve code readability and maintainability.
4. **Improve User Experience:** Include clear prompts, game instructions, and visual representation of the hangman to make the game user-friendly.

Requirements:

Functional Requirements:

- Random word selection from a words list.
- User can guess one letter at a time.
- Keep 15sec timer for each letter guess.
- Display the word's progress with correct guesses.
- Track wrong guesses and show the hangman visually.
- The game will be finished when the word is guessed or maximum attempts are reached.

Non-Functional Requirements:

- Clear and user-friendly interface with instructions and error handling.
- Readable and maintainable code with comments and meaningful variable names.
- Unit tests covering all major functionalities.
- Portable across systems with Python installed.

Automated unit testing tool

Alongside the implementation, the automated unit testing tool has been used for testing purpose. Automated unit testing is a software testing approach where individual units or components of a program are tested automatically to ensure they function as intended. The primary goal of unit testing is to verify that each component behaves correctly under different conditions and handles both expected and unexpected inputs. Automated unit testing uses testing frameworks like **pytest**, **unit test**, or **JUnit** to run test cases without manual intervention. By automating the tests, developers can quickly check the correctness of the code after every change, detect error early and easily improve the code. This approach is especially valuable in projects following **Test-Driven Development (TDD)**, where tests are written before the actual code to guide implementation and ensure correctness from the beginning.

In this Hangman project, I used "Pytest" as the unit testing tool. Pytest is a popular Python framework that makes it easy to write and run tests for the code. It allows to test small parts of the program at a time like individual functions, to make sure that each function can work accordingly in a perfect manner. It helps each function such as selecting a word, checking guessed letters, and updating the hangman figure or components to be tested individually. By running tests automatically not only bugs can be detected at an early stage but also reducing the chances of issues in the final program. Another benefit of using Pytest is, it gives detailed and easy-to-read results, making every command straightforward to identify and fix problems in a short time.

Some of the key benefits of Pytest include:

1. Verify all the codes are correct or not.
2. Detect bugs and catch errors early.
3. It saves time compared to the manual testing process.
4. Helps to maintain code quality and increased readability.
5. Provide cleared and easy to understand test result. So that, one can easily take the feedback and work on the code.

Process:

The Hangman game project was developed following the Test-Driven Development (TDD) approach. In TDD, development starts by writing tests before implementing the actual functionality. This ensures that every feature is validated as it is developed and reduces the chances of introducing bugs. The TDD cycle usually follows three simple steps (Red–Green–Refactor).

1. **Write a test (Red):**
   Start by writing a test case for a feature or function that does not exist yet. At this point, the test will fail because the code hasn't been implemented.
2. **Write the code (Green):**
   Write just enough code to make the test pass.
3. **Refactor (Clean up):**
   Once the test passes, review and improve the code; make it cleaner, remove duplication, improve readability.

In this project of hangman game. I had to do some unit test. For this, I applied ‘pytest’ method for testing each function.

Basic level: In this step, the code is written for generate the basic part of the game. This test checks whether the function generate_word() returned any word or not. Since, the functions are not implemented yet, it showing ‘FAILED’ as expected. The necessary code has been implemented for the basic level word generation and they ‘PASSED’ the test.

```python
import pytest
from hangman import HangmanGame

def test_basic():
    game = HangmanGame(level="basic")
    word = game.generate_word()
    assert isinstance(word, str)
    assert len(word) > 0
```

```
test_hangman.py::test_basic FAILED                                       [100%]

=================================== FAILURES ===================================
_____________________________ test_basic _____________________________

    def test_basic():
        game = HangmanGame(level="basic")
```

Figure1: Test for basic level word generation before any implemented function.

```python
class HangmanGame:
    def __init__(self, level="basic", lives=6):
        self.level = level
        self.lives = lives
        self.max_lives = lives
        self.word_list = self.load_words()
        self.word = ""
        self.guessed_letters = []
        self.timer_thread = None
        self.time_up = False
        self.timer = 15

    def load_words(self):
        try:
            with open("words.txt", "r") as f:
                words = [line.strip() for line in f if line.strip()]
            return words
        except FileNotFoundError: #If some words are missing in the word list; it will show these words.
            return ["python", "hangman", "testing", "programming", "unit", "test"]

    def generate_word(self):
        if self.level == "basic":
            self.word = random.choice(self.word_list)
        elif self.level == "intermediate":
            # Ensure we have enough words for a phrase
            if len(self.word_list) >= 2:
                self.word = random.choice(self.word_list) + " " + random.choice(self.word_list)
            else:
                self.word = random.choice(self.word_list)
        return self.word
```

```
test_hangman.py::test_basic PASSED                                       [100%]

============================== 1 passed in 0.01s ===============================
PS C:\Users\hp\OneDrive\Documents\Python\Hangman>
```

Figure 2: Code for basic level word generation.

Intermediate level: In this step, the code is written for generate the basic part of the game. This test checks whether the function generate_word() returned any word or not. Since, the functions are not implemented yet, it showing 'FAILED' as expected. Here, the necessary code has been implemented for the intermediate level word generation and they 'PASSED' the test.

Figure 3: Test for intermediate level word generation before any implemented function.

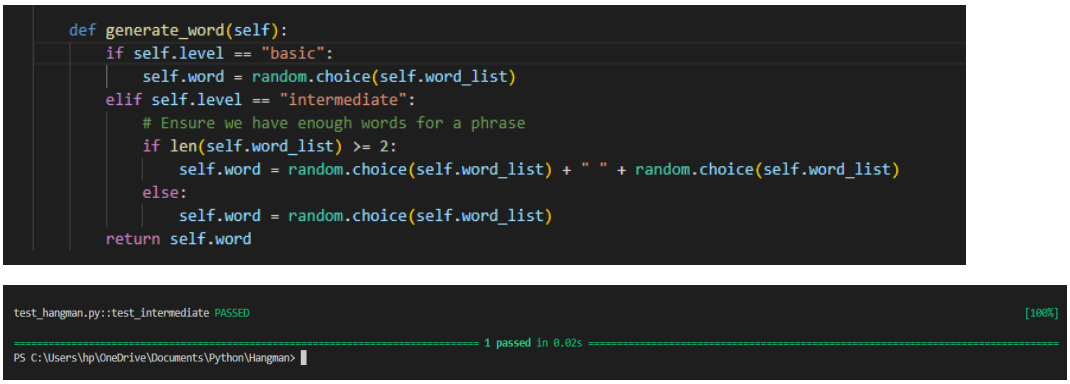

Figure 4: Code for intermediate level word generation and the result of passing the test.

Words from dictionary: This step confirms that the valid words are load from the dictionary. And if the chosen word is not in the dictionary, it will return some built-in words.

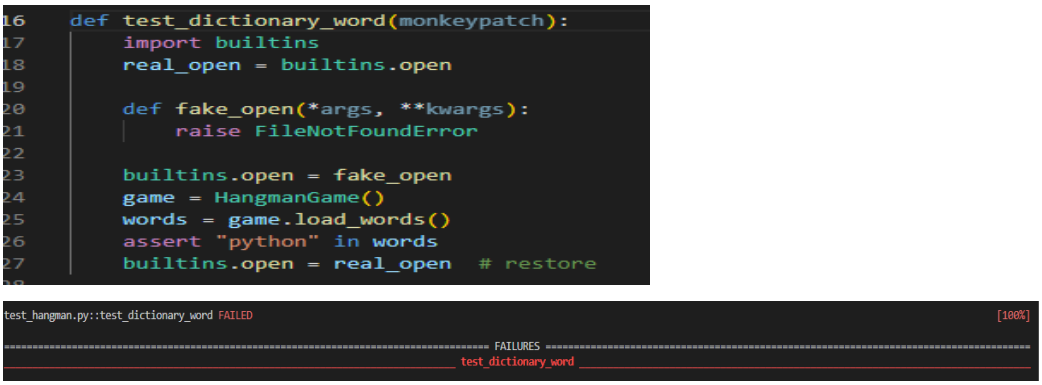

Figure 5: Test for proper word(s) generation from dictionary.

In this part, this function loads each line as a word. If words.txt contains valid dictionary word the game going to use appropriate word. Otherwise, it will return some built-in words such as, hangman, testing test etc.

```python
    def load_words(self):
        try:
            with open("words.txt", "r") as f:
                words = [line.strip() for line in f if line.strip()]
            return words
        except FileNotFoundError:
            return ["python", "hangman", "testing", "programming", "unit", "test"]
```

```
test_hangman.py::test_dictionary_word PASSED                                                        [100%]

=============================== 1 passed in 0.03s ===============================
PS C:\Users\hp\OneDrive\Documents\Python\Hangman>
```

Figure 6: Code for proper word(s) generation from dictionary and the passing test.

Hidden letter handler: The hidden letter handler manages the display of the secret word by showing underscores for unguessed letters and revealing letters as the player guesses them correctly. It tracks all guessed letters to prevent repeated penalties, updates the display dynamically after each guess, and preserves spaces in multi-word phrases, ensuring the player can gradually uncover the word while maintaining the challenge of the game.

```python
def test_display_word():
    game = HangmanGame()
    game.word = "python"
    game.guessed_letters = []
    assert game.get_display_word() == "______"

def test_phrase_keeps_spaces():
    game = HangmanGame()
    game.word = "hello world"
    assert game.get_display_word() == "_____ _____"
```

```
test_hangman.py::test_display_word FAILED                                                           [100%]

=============================== FAILURES ===============================
_______________________________ test_display_word _______________________________
```

Figure 7: Test for hidden letter handler and the result.

```python
# This function shows the word with underscores (_) for letters
    def get_display_word(self):
        display = ""
        for ch in self.word: #Check character in the words
            if ch == " ":
                display += " "
            elif ch.lower() in self.guessed_letters: #If user guessed ch; display
                display += ch
            else:
                display += "_" #If can't guess; show '_'
        return display
```

```
test_hangman.py::test_display_word PASSED                                [100%]

============================== 1 passed in 0.03s ==============================
PS C:\Users\hp\OneDrive\Documents\Python\Hangman>
```

Figure 8: Code for hidden letter handler and the passing test.

15-Second Guess Timer: The timer function gives the player 15 seconds to make a guess; if no input is received within this time, one life is deducted and the hangman state is updated. The test simulates this by mocking the time.sleep function to avoid delays, starting the timer, and joining the countdown thread to completion. Since no guess is made (time_up remains False), the test verifies that the player's lives decrease from 3 to 2, confirming the timer's penalty logic works correctly.

```python
def test_timer_deducts_life(monkeypatch):
    game = HangmanGame(lives=3)

    # Monkeypatch time.sleep to speed up test (no actual delay)
    monkeypatch.setattr(time, "sleep", lambda x: None)

    # Start timer
    game.start_timer()

    # Wait a little so the thread finishes execution
    game.timer_thread.join()

    # Since time_up was never set to True, life should be deducted
    assert game.lives == 2
```

```
test_hangman.py::test_timer_deducts_life FAILED                          [100%]

================================== FAILURES ===================================
_____________________________ test_timer_deducts_life _____________________________
```

Figure 9: Test result for Time-Limited Guessing Mechanism.

```python
#Timer
    def start_timer(self):
        self.time_up = False
        self.timer = 15

        def countdown():
            for i in range(15, 0, -1):
                if self.time_up:
                    return
                self.timer = i
                time.sleep(1)

            if not self.time_up:  # if user hasn't guessed yet
                self.lives -= 1
                print("\nTime's up! You lost a life.")
                self.display_hangman_art()

        self.timer_thread = threading.Thread(target=countdown, daemon=True)
        self.timer_thread.start()
```

```
test_hangman.py::test_timer_deducts_life PASSED                          [100%]

============================== 1 passed in 0.03s ==============================
PS C:\Users\hp\OneDrive\Documents\Python\Hangman>
```

Figure 10: Code for Time-Limited Guessing Mechanism with passing test.

Letter Reveal Handler: The letter reveal handler checks if a guessed letter exists in the word and, if so, uncovers all positions where it appears. The test verifies this by guessing "a" in "banana" and confirming that all occurrences are revealed, resulting in the display _a_a_a, ensuring multiple instances of a letter are correctly handled.

```
def test_reveals_all_positions():
    game = HangmanGame()
    game.word = "banana"
    game.guess_letter("a")
    assert game.get_display_word() == "_a_a_a"
```

```
test_hangman.py::test_reveals_all_positions FAILED                                                    [100%]

================================================================================ FAILURES ================================================================================
_____________________________________________________________________ test_reveals_all_positions _____________________________________________________________________
```

Figure 11: Testing code for proper all position of the guessed word revealed and the test revealed.

```
#This function checks guessed letters
    def guess_letter(self, letter):
        letter = letter.lower()     #Convert all letters to lowercase

        if letter in self.guessed_letters:
            return None  # already guessed then no life take away
        self.guessed_letters.append(letter) #Add letters to the guessed letters list

        if letter in self.word.lower(): #If letter is in the word
            return True
        else:
            self.lives -= 1 #If letter is not present in the word; kill one
            return False
```

```
test_hangman.py::test_reveals_all_positions PASSED                                                    [100%]

================================================================================ 1 passed in 0.02s ================================================================================
PS C:\Users\hp\OneDrive\Documents\Python\Hangman>
```

Figure 12: Code for proper all position of the guessed word revealed and the passing test.

Life Deduction on Wrong Guess: The life deduction handler reduces the player's lives by one whenever a guessed letter is not present in the word. The test confirms this by guessing "x" in "python"; since the letter is incorrect, the function returns False and the player's lives decrease from 3 to 2, ensuring wrong guesses are properly penalized.

```
def test_guess_letter_incorrect():
    game = HangmanGame(lives=3)
    game.word = "python"
    result = game.guess_letter("x")
    assert result is False
    assert game.lives == 2
```

```
test_hangman.py::test_guess_letter_incorrect FAILED                                                   [100%]

================================================================================ FAILURES ================================================================================
_____________________________________________________________________ test_guess_letter_incorrect _____________________________________________________________________
```

Figure 13: Testing code for Life Deduction on Wrong Guess.

```python
#This function checks guessed letters
    def guess_letter(self, letter):
        letter = letter.lower()     #Convert all letters to lowercase

        if letter in self.guessed_letters:
            return None  # already guessed then no life take away
        self.guessed_letters.append(letter) #Add letters to the guessed letters list

        if letter in self.word.lower(): #If letter is in the word
            return True
        else:
            self.lives -= 1 #If letter is not present in the word; kill one
            return False
```

```
test_hangman.py::test_guess_letter_incorrect PASSED                                    [100%]

================================ 1 passed in 0.03s ================================
PS C:\Users\hp\OneDrive\Documents\Python\Hangman>
```

Figure 14: Code for Life Deduction on Wrong Guess.

Game Outcome Evaluation: The win and loss handlers determine the game outcome. is_won() checks if all letters in the word have been guessed, signaling a win, while is_lost() checks if the player's lives have reached zero, signaling a loss. The tests confirm these conditions by simulating a fully guessed word for a win and zero lives for a loss, ensuring the game correctly identifies both outcomes.

```python
def test_win():
    game = HangmanGame()
    game.word = "hi"
    game.guessed_letters = ["h", "i"]
    assert game.is_won() is True

def test_lose():
    game = HangmanGame(lives=0)
    assert game.is_lost() is True
```

```
test_hangman.py::test_win FAILED                                                       [ 50%]
test_hangman.py::test_lose FAILED                                                      [100%]

================================ FAILURES ================================
________________________________ test_win ________________________________
```

Figure 15: Testing code for Game Outcome Evaluation.

```python
#if the player has guessed the whole word.
    def is_won(self):
        for ch in self.word.lower():
            if ch.isalpha() and ch not in self.guessed_letters:
                return False    #if there are still missing letters
        return True  #if all letters have been guessed

#if the player has run out of lives.
    def is_lost(self):
        return self.lives <= 0
```

```
test_hangman.py::test_win PASSED                                                       [ 50%]
test_hangman.py::test_lose PASSED                                                      [100%]

================================ 2 passed in 0.06s ================================
PS C:\Users\hp\OneDrive\Documents\Python\Hangman>
```

Figure 16: Code for Game Outcome Evaluation.

Game Continuation and Quit Handling: This section manages the main game loop, ensuring the Hangman game continues until the player either guesses the word correctly or runs out of lives. It also provides a quit option, allowing the player to exit at any time. The system dynamically updates the display after each guess, shows the current word state and remaining lives, and reveals the correct answer when the game ends, ensuring a smooth and user-friendly gameplay experience.

```
# Time and Quit Option
    def play(self):
        self.generate_word()
        print("************Welcome to Hangman!**************")
        print(f"Level: {self.level.capitalize()}")
        print(f"You have {self.lives} lives. Good luck!\n")

        while not self.is_won() and not self.is_lost():
            print("\nWord:", self.get_display_word())
            print(f"Lives: {self.lives}")
            self.display_hangman_art()
            # Start the timer for this turn
            self.start_timer()
            try:
                guess = input(f"Time left: {self.timer}s - Enter a letter (or 'quit' to exit): ")
            except EOFError:
                print("\n########Thanks for playing!############")
                return
```

```
                self.time_up = True  # stop timer once input is received
                if guess.lower() == "quit":
                    print("########Thanks for playing!#########")
                    return
                if len(guess) != 1 or not guess.isalpha():
                    print("!!Please enter a single letter.")
                    continue
                result = self.guess_letter(guess)
                if result is True:
                    print(f"Good guess: {guess}")
                elif result is False:
                    print(f"Wrong guess: {guess}")
                    self.display_hangman_art()
                else:
                    print(f"Already guessed: {guess}")
            if self.is_won():
                print(f"*******Congratulations! You guessed the word: {self.word}*******")
            else:
                print(f"Game Over! The word was: {self.word}")
                self.display_hangman_art()
```

Figure 17: Code for Game Outcome Evaluation.

Output Interface Design and User Experience:

In this Hangman game features a clean command-line interface that clearly displays the essential elements: the partially revealed word, remaining lives, visual hangman progress, and a 15-second countdown timer. I run the code on the terminal of VS Code.

The interface provides user to choose the level of the game as well as correct guesses fill in letters while wrong answers advance the hangman drawing and reduce lives. The timer adds urgency, deducting a life if time expires.

The game ends by revealing the full word with either a congratulatory message for wins or a gentle game over notice for losses. The design maintains classic Hangman tension through visual progression and time pressure while ensuring clear communication and fair gameplay throughout the experience. A basic stage of the code outcome are showing below:

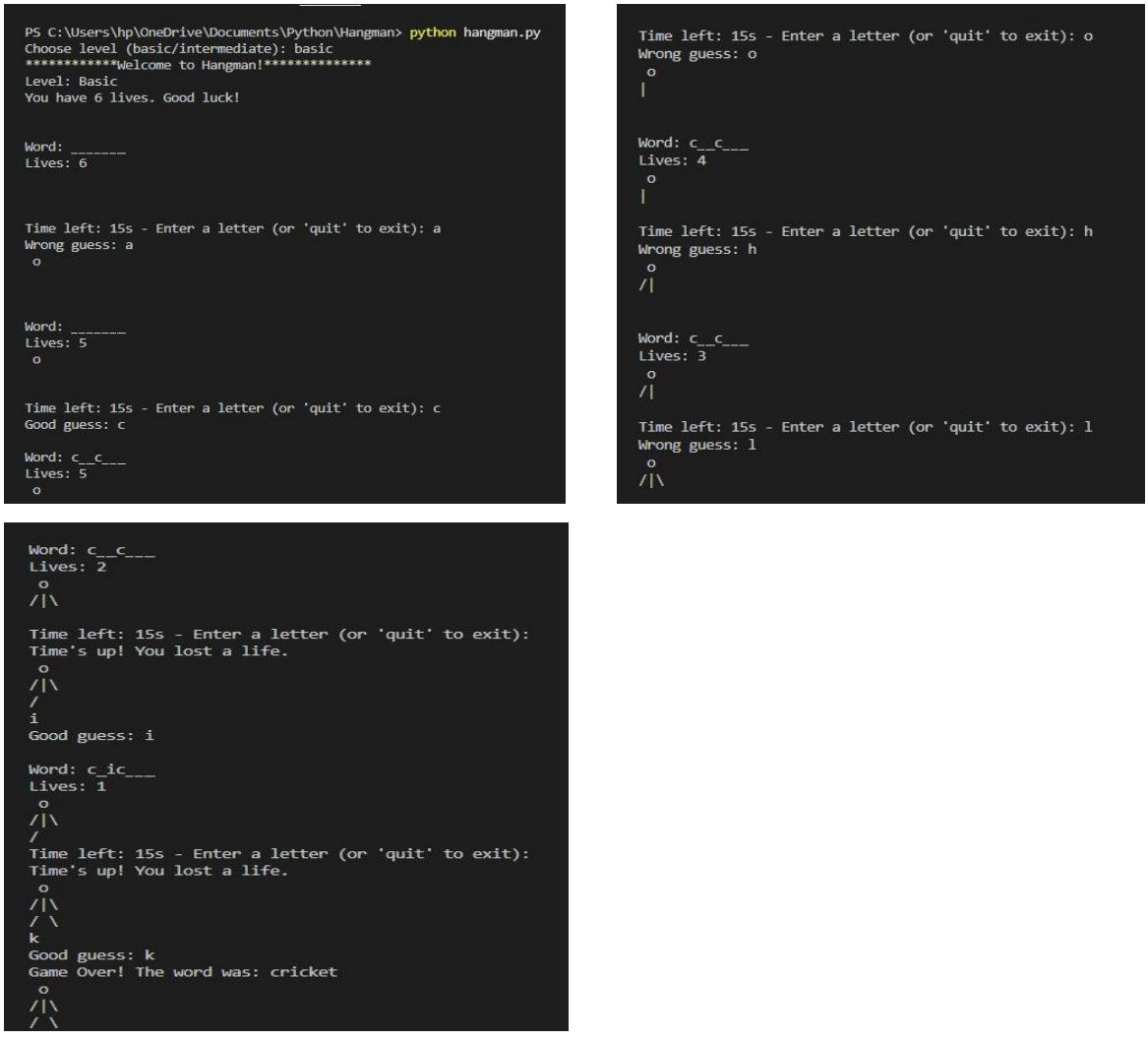

Figure 16: Code for Game Outcome Evaluation.

Conclusion:

The Hangman game project allowed me to practically apply Test Driven Development (TDD) principles and automated unit testing using Pytest to build a reliable and interactive game. By following the TDD cycle of writing tests first, implementing code to make the tests pass, and then refactoring, I was able to ensure correctness at every stage of development. Each major functionality such as random word and phrase generation, letter reveal handling, life deduction on wrong guesses, timer based guessing, and outcome evaluation was validated through unit tests, which helped detect bugs early and maintain stability throughout the project. This process highlighted the benefits of automated testing in saving time, improving maintainability, and preventing regressions. Beyond testing, the project enhanced my programming skills in Python, improved my understanding of modular and clean code design, and emphasized the importance of creating user friendly software. Overall, the project not only resulted in a fully functional Hangman game but also strengthened my confidence in using professional software development practices.

GitHub Repository Link: [https://github.com/Anupoma/Hangman-Game]