**PRT582 SOFTWARE ENGINEERING: PROCESS AND TOOLS**

Assessment 2

Software Unit Testing Report

**Name: Amanparteek Singh  
Student ID: S391075  
Campus: ECP Darwin  
Course Code: SITSE2  
Submission Date: 2nd Sep 2025**



Contents

[Introduction 3](#_Toc207182206)

[Game Overview 3](#_Toc207182207)

[Process: Test-Driven Development (TDD) 4](#_Toc207182208)

[Requirement 1: Masking the Word 4](#_Toc207182209)

[Requirement 2: Correct Guess Handling 5](#_Toc207182210)

[Requirement 3: Wrong Guess Handling 6](#_Toc207182211)

[Requirement 4: Win and Lose Conditions 7](#_Toc207182212)

[Requirement 5: Timer Functionality 8](#_Toc207182213)

[Automated Unit Testing Tool 9](#_Toc207182214)

[Conclusion 11](#_Toc207182215)

[GitHub Link: 12](#_Toc207182216)

[References 13](#_Toc207182217)

# Introduction

This report describes the process of developing a Hangman game using Test-Driven Development (TDD) and using the automated unit testing framework provided by Python, the unittest module. The game challenges players to find a randomly chosen word or phrase in a predetermined amount of time, with lives being taken away as guesses become incorrect as well. It is intended to create a fully-functional game that is compelling and sustainable.

This project adopted TDD which is the test-driven development whereby the tests are coded first before code is written, and this will ensure that the software is behaving as required all the time. This step does not only validate the game properties, it also makes the code more maintainable and cleaner to deal with. The process of test-driven development, writing tests, implementing features, and refactoring the code incrementally allowed us to build and validate each of the components of the game.

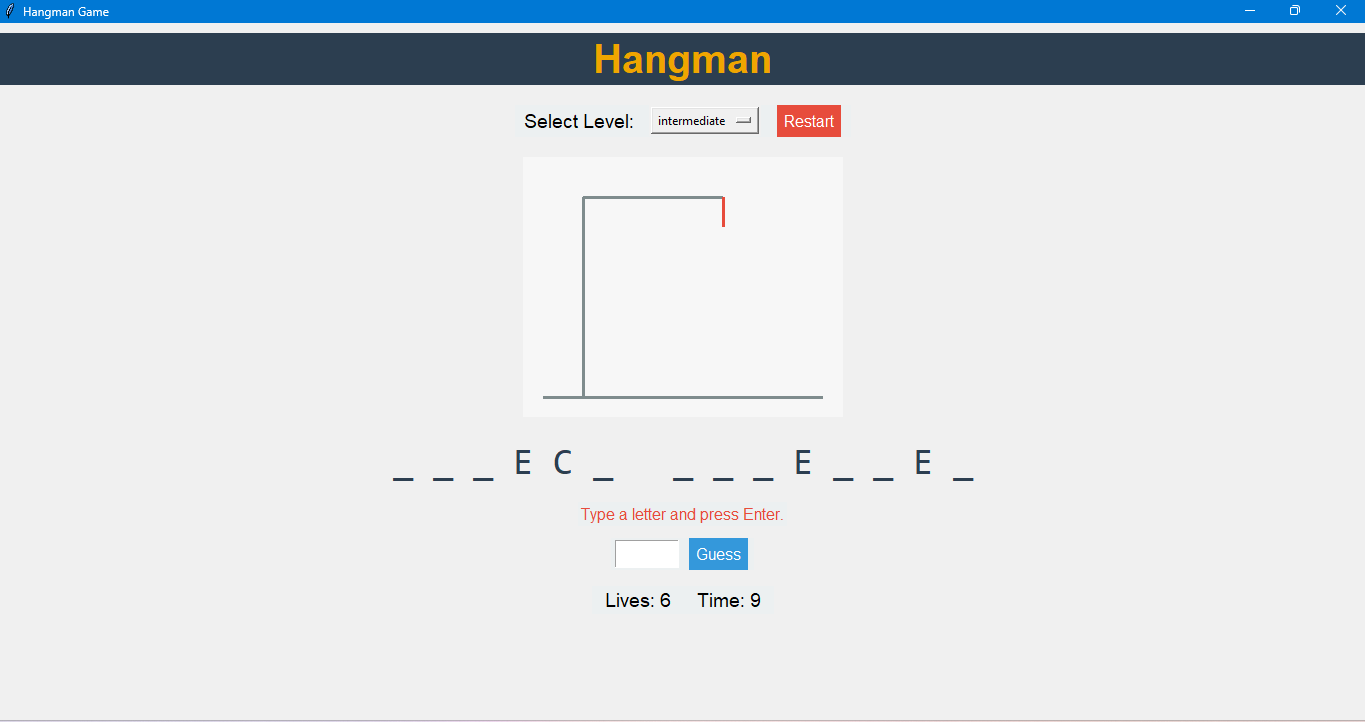
Python was applied to this project as it is simple and the unit testing support is comprehensive using “unittest”. This test framework helped in automating tests to check on different aspects of play including the selection of words, guesses made by the players, and the state changes in the game. The following section details the TDD process, and how it was implemented throughout game development.

# Game Overview

The **Hangman Game** is a classic word-guessing game where players must guess a randomly chosen word by suggesting letters one at a time. The player has a limited number of lives to guess the word before the hangman figure is fully drawn. The game starts with a masked word (represented by underscores) and reveals correctly guessed letters as the player proceeds.

Key Features/Requirerments:

* **Difficulty Levels**: The game offers two difficulty levels—**basic** and **intermediate**. The words selected for each level vary in difficulty, with **basic** containing simpler words and **intermediate** having more complex terms.
* **Lives and Timer**: Players start with 6 lives. A life is deducted for each incorrect guess, and the game also includes a timer to add more challenge. If time runs out, a life is lost.
* **User Interface :** The game has a **graphical user interface (GUI)** created in **Tkinter**. The interface shows the word, players may type guesses, how many lives they have left, as well as the time counter.
* **Hangman Drawing**: As incorrect guesses are made by the player , a drawing of the **hangman figure** is drawn on the screen, adding a visual element to the game.



# Process: Test-Driven Development (TDD)

**Test-Driven Development (TDD**) is a software development approach whereby tests are produced prior to writing of actual codes. This is so that the software produced can comply with the pre-determined requirements right at the onset and help prevent flaws in the development process. The TDD cycle follows four key steps:

1. **Write a Test**
2. **Run the Test**
3. **Write the Code**
4. **Refactor**

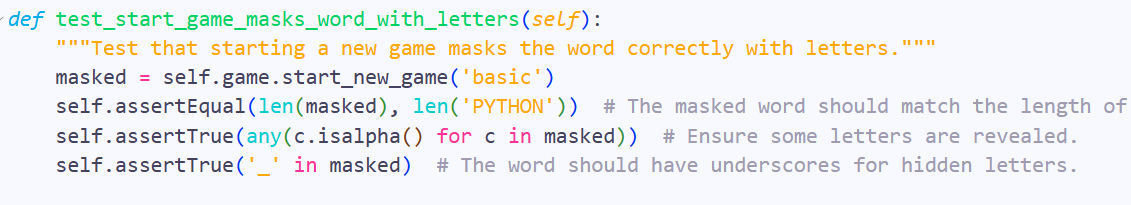
***Below is the explanation how each requirement of this game has been implemented using Test Driven Development Methodology*:**

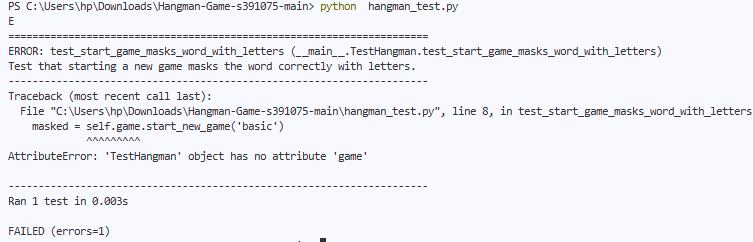
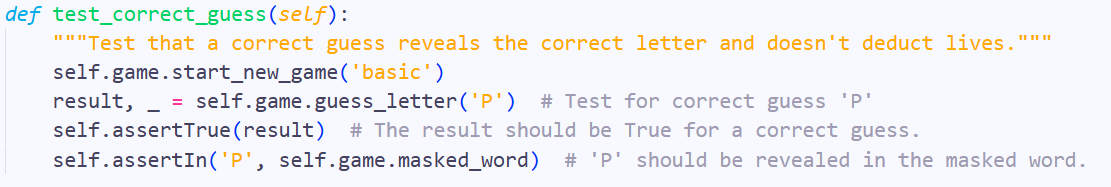
### Requirement 1: Masking the Word

**Requirement**: The game should display underscores for hidden letters, and reveal a few letters at random at the start.

**TDD Approach**:

1. **Write the Test**:
   * Ensure at least 1 letter is revealed.
   * Ensure all other letters are underscores.



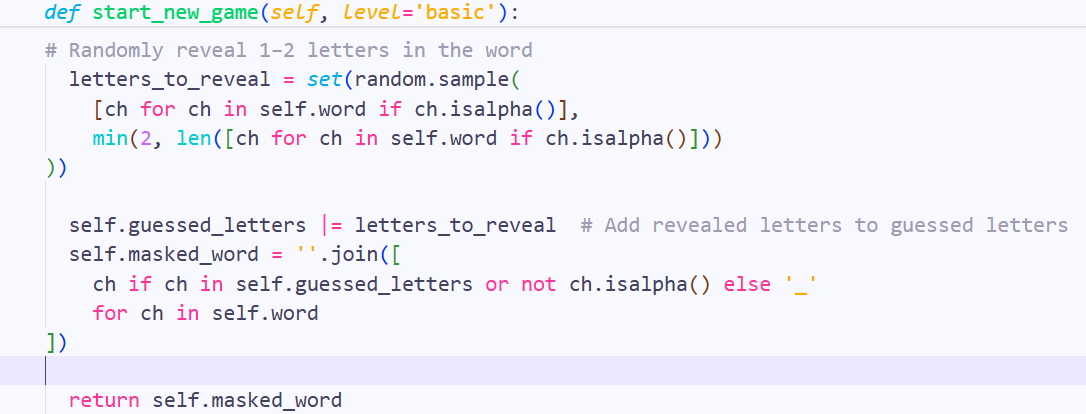
1. **Run the Test**:
   * Fails at first because masking logic isn’t implemented.
2. **Write the Code**:
   * Add logic to generate masked\_word with underscores and reveal 1–2 letters.
3. **Refactor**:
   * Move masking logic to a helper function (\_mask\_word) for readability.

### Requirement 2: Correct Guess Handling

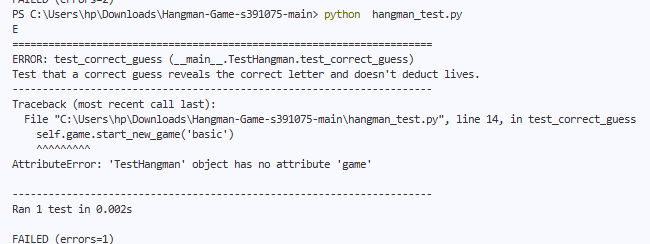
**Requirement**: The game should reveal letters in the masked word if the player guesses correctly.

**TDD Approach**:

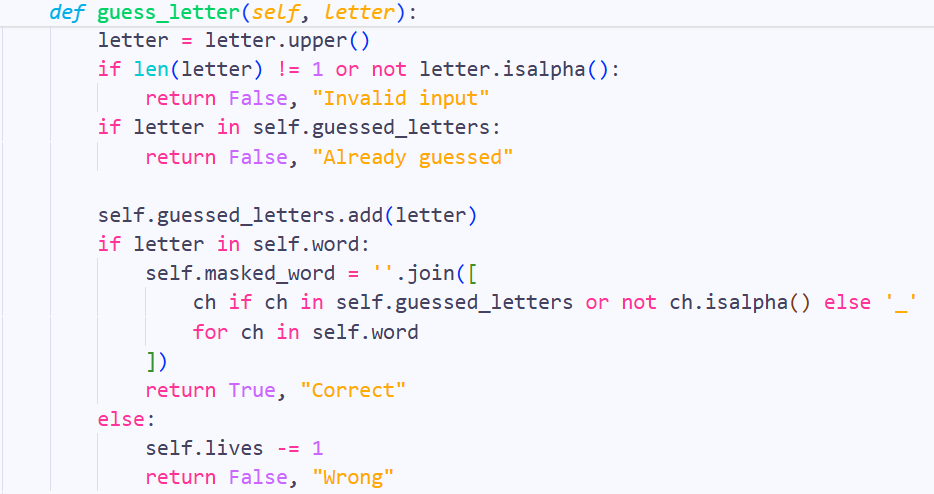
1. **Write the Test**:
   * Check that guessing a correct letter updates masked\_word.



1. **Run the Test**:
   * Fails because guess\_letter is not yet implemented.



1. **Write the Code**:
   * Implement guess\_letter to update masked\_word for correct guesses.

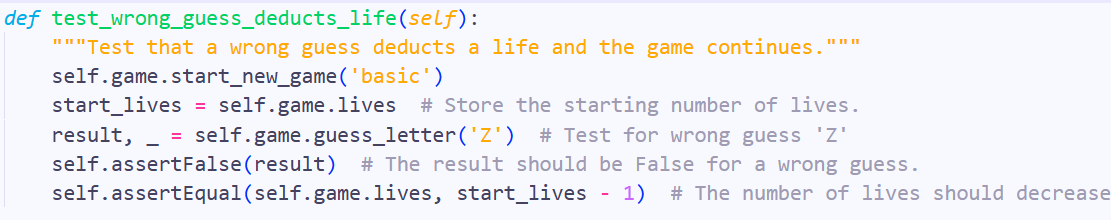


1. **Refactor**:
   * Clean up logic and ensure repeated guesses don’t break the game.

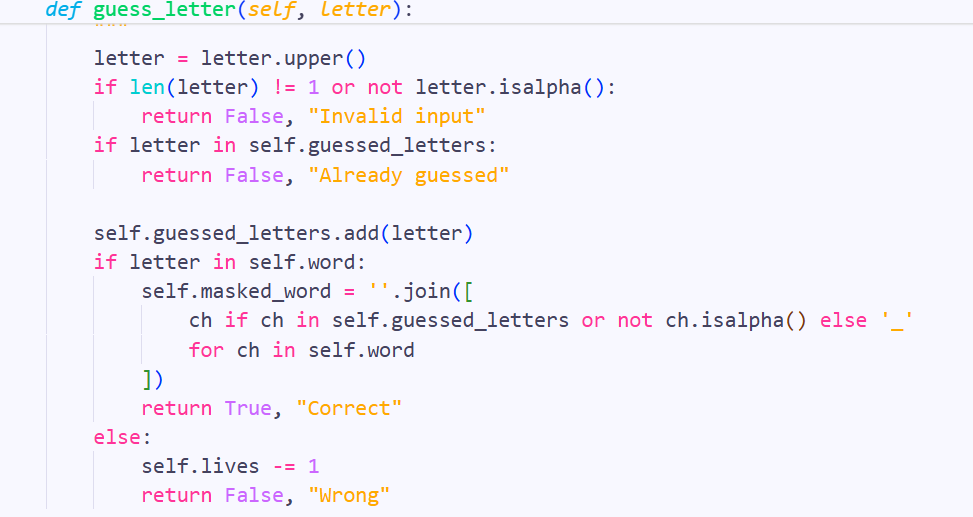
### Requirement 3: Wrong Guess Handling

**Requirement**: Deduct a life if the player guesses incorrectly.

**TDD Approach**:

1. **Write the Test**:
   * Check that lives decrease after a wrong guess.

1. **Run the Test**:
   * Fails initially.
2. **Write the Code**:
   * Implement logic in guess\_letter to deduct a life for incorrect guesses.

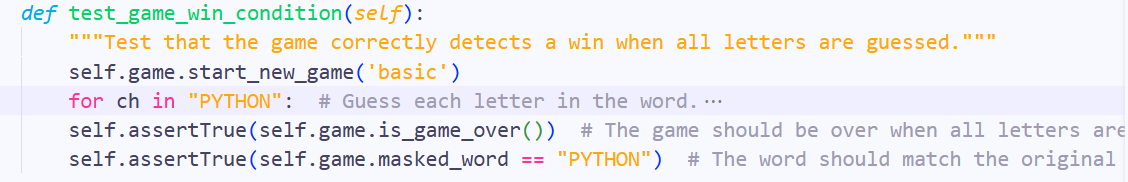


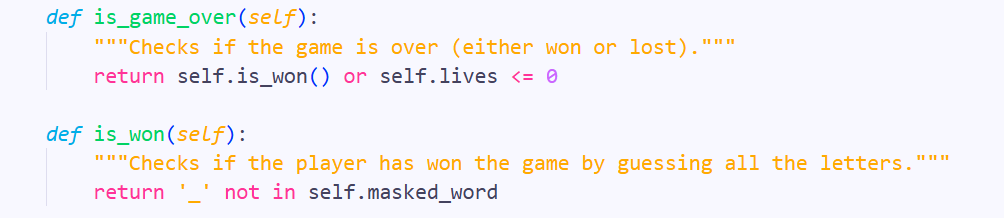
1. **Refactor**:
   * Ensure the method is clean and prevents negative lives.

### Requirement 4: Win and Lose Conditions

**Requirement**: End the game when the player guesses all letters (win) or runs out of lives (lose).

**TDD Approach**:

1. **Write the Test**:
   * Check that the game recognizes a win when all letters are revealed.
2. **Run the Test**:
   * Fails at first since is\_game\_over isn’t implemented.
3. **Write the Code**:
   * Implement is\_won, and is\_game\_over methods.

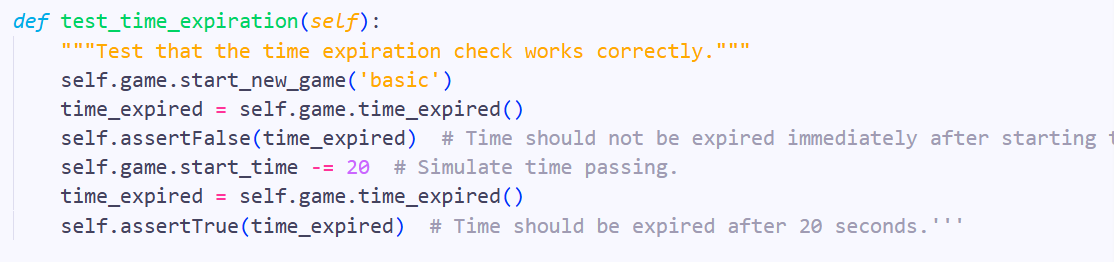


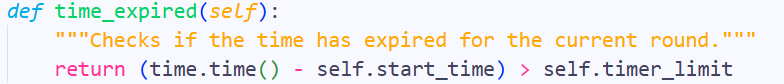
1. **Refactor**:
   * Combine win/lose logic in is\_game\_over for simplicity.

### Requirement 5: Timer Functionality

**Requirement**: Deduct a life if the player exceeds the time limit for a guess.

**TDD Approach**:

1. **Write the Test**:
   * Simulate passing time and check if a life is deducted.
2. **Run the Test**:
   * Fails if time\_expired is not implemented.
3. **Write the Code**:
   * Implement time\_expired method to check time and handle deductions.



1. **Refactor**:
   * Ensure timer resets properly for the next guess.

.

## Automated Unit Testing Tool

In this section, I describe the **unit testing** process used to validate the functionality of the **Hangman game**. The **unittes**t is a built-in python library used to check whether the game acts as it should. I concentrate on how to use automated tests to validate important game functionality, including word selection, processing of player clues, loss of lives count, and timing. The tests were executed according to the **Test-Driven Development (TDD),** and they make sure the logic of the game is confirmed at any step of the development process..

**1. Overview of the unittest Framework:**

The **unittest** framework is part of Python’s standard library and provides tools for writing and running tests. It enables the creation of **test cases**, running those tests, and checking if the code works as expected.

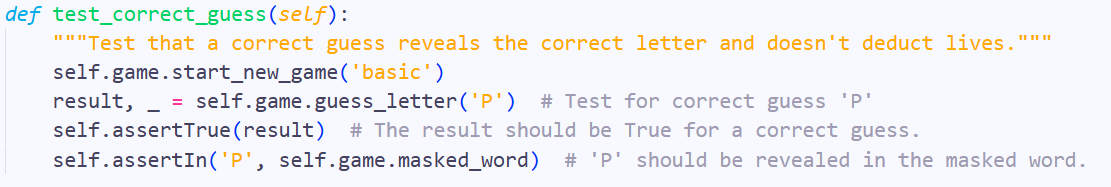
Key Features of **unittest**:

* **Test Cases**: You may write test cases that test a particular feature or functionality of the game (e.g., check that word is obscured well).
* **Assertions**: Such automated tests employ assertions to test the correctness of the game behavior. This can be as simple as making certain that when there is a wrong guess, lives are taken away.
* **Automated Execution**: Once the tests are written, you can run them automatically without checking every part of the game manually.
* **Test Reports**: After executing the tests, the framework generates a test report, showing which test is passed and which failed.

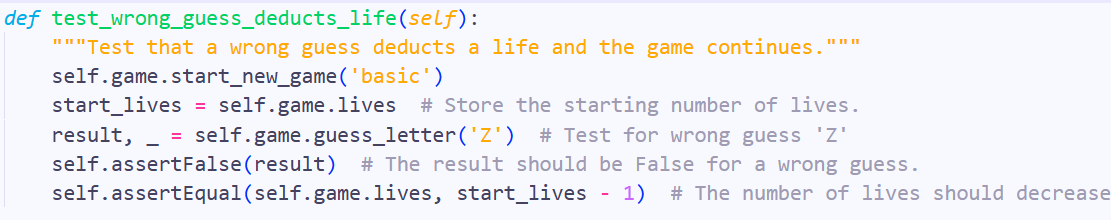
**2. How unittest is Used in the Hangman Game:**

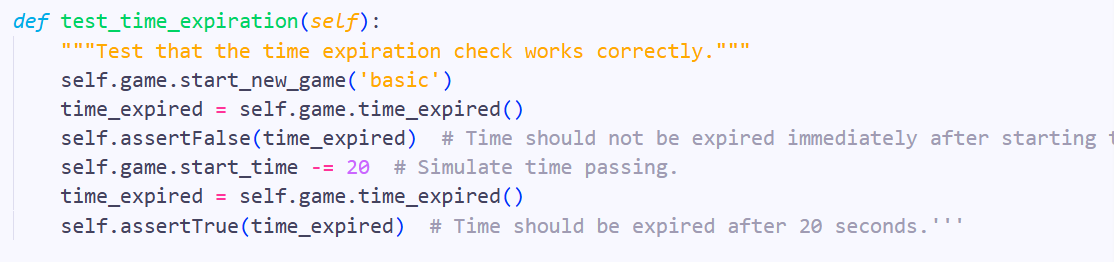
I utilized **unittest** to validate critical features of the **Hangman game**, ensuring that all parts of the game were functioning correctly. Below are some of the key areas that were tested:

* **Correct Guess**:  
  I tested that when a player guesses correctly, the corresponding letter is revealed in the word.  
  Example:



* **Incorrect Guess**:  
  I validated that a life is deducted when a player makes an incorrect guess.  
  Example:



* **Timer Functionality**:  
  We verified that the timer works correctly and a life is deducted if the time runs out.  
  Example:

**3. Running the Tests:**

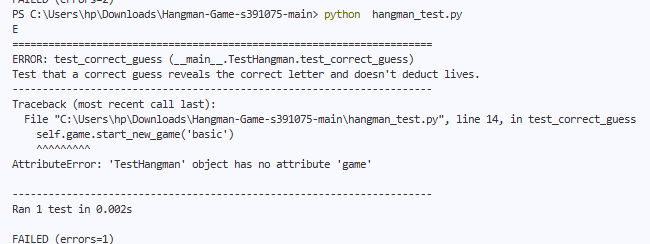
To run the tests, you can execute the following command in the terminal:

***python -m unittest test\_hangman.py***

This command will automatically run all test methods (those starting with test\_) in the **test\_hangman.py** file. Here's a breakdown of what happens during execution:

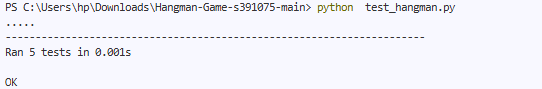
* **Test Failure (Before Code Implementation)**:  
  When running the tests before implementing any functionality, they are expected to fail because the code is not yet written. For example, the test for **word selection** might fail because the logic for selecting a word hasn't been implemented.

**Example Output** (Test Failure):



* **Test Success (After Code Implementation)**:  
  Once the necessary game logic (e.g., word selection, handling guesses) is implemented, the tests should pass.

**Example Output** (Test Success):



This indicates that all tests have passed successfully, confirming that the functionality is working as expected.

# Conclusion

**Lessons Learned:**

1. **What Went Well:**

* **Test-Driven Development (TDD)** methodology proved to be very useful in verifying the game logic prior to writing. Writing the tests before writing the actual game code allowed me to clearly defined the expected behavior, and identify bugs at an early stage.
* The **automatic unit testing tool (unittest)** permitted protracted testing during development. It assisted to detect the problems with the functionality of selecting words, guessing, and making correct/incorrect guesses, as well as with the timer mechanism. Frequent checks validated the stability and correctness of the game.
* The main Hangman gameplay (word selection, guess processing, lives counting, and the clock) was concluded according to the plan, and it has been successfully tested with the help of automated tests which gave an assurance of the logic of the game as a whole.

1. **Areas for Improvement:**
   * **Testing Coverage**: The testing coverage could have been extended to include more edge cases, such as:
     + **Invalid Inputs**: Handling cases where the user enters multiple characters at one go or non-alphabetical characters.
     + **Repeated Guesses**: Making sure the game works correctly when a letter has been guessed more than once.
     + **Time Overflow**: Handling cases of the game timer containing overflow errors or unpredictable behavior when the game itself runs long.
   * **Error Handling**: The error handling in the game can be improved, especially when the unexpected inputs are triggered by the user. It would improve user experience to implement the more specific error messages of edge cases.
   * **UI/UX Improvements**: Albeit the simple interface of **Tkinter** is perfectly workable, it can be enhanced to make the interface more interactive and pleasing to the eye. For example:
     + Adding animations or sound effects when the player guesses correctly or incorrectly.
     + Improving the **hangman drawing** by making it more visually distinct for each incorrect guess.
     + A more polished **end-game screen** that displays clear options to restart the game, view a scoreboard, or exit.

GitHub Link:  
<https://github.com/amanparteek/Hangman-Game-s391075.git>

# References

* Beck, K. (2002). *Test-driven development: By example*. Addison-Wesley Professional.
* Qiang, B. (2016, October 27). *The absolute beginner’s guide to test driven development, with a practical example*. Medium. <https://medium.com/@bethqiang/the-absolute-beginners-guide-to-test-driven-development-with-a-practical-example-c39e73a11631>
* Martin, R. C. (2003). *Agile software development, principles, patterns, and practices*. Prentice Hall.
* Python Software Foundation. (n.d.). *Python documentation*. <https://docs.python.org/3/>
* Python Software Foundation. (n.d.). *Tkinter documentation*. <https://docs.python.org/3/library/tkinter.html>
* Redfern, M. (2020). *Tkinter GUI programming with Python*. Packt Publishing.
* Ramos, L. P. (2025, July 29). *Python's unittest: Writing unit tests for your code*. Real Python. <https://realpython.com/python-unittest/>
* EnglishClub. (n.d.). *Hangman game*. <https://www.englishclub.com/esl-games/hangman/>