



software Engineering: Process and tools

**Software Unit Testing Report**



**Student Name:** Binod Basnet  
**Student ID:** S388532  
**Campus Location:** Sydney  
**Course Code:** PRT582

Table of Contents

[1. Introduction 3](#_Toc207886571)

[2. Program 3](#_Toc207886572)

[2.1 Requirements Fulfilment 3](#_Toc207886573)

[3. Process 3](#_Toc207886574)

[3.1 Design Choices 3](#_Toc207886575)

[3.2 Workflow 5](#_Toc207886576)

[3.3 Test Cases 5](#_Toc207886577)

[3.4 Output and Discussion 6](#_Toc207886578)

[4. Conclusion 10](#_Toc207886579)

[5. References 11](#_Toc207886580)

# Introduction

This report documents the design, development, and testing of a Hangman game built as part of the *Software Engineering Process and Tools* unit. The project was implemented in Python, with clear separation between core game logic, graphical user interface (GUI), and automated unit tests. The program strictly follows the given requirements: two difficulty levels (basic and intermediate), valid technology-related words/phrases, a timer of 15 seconds per guess, life deduction for wrong or delayed guesses, full reveal of correctly guessed letters, and proper game termination conditions.

Python was chosen for this project due to its readability, wide adoption in both industry and academia, and strong ecosystem for testing (van Rossum & Drake Jr, 2009). The automated unit testing tool used is Python’s built-in “*unittest”*, which integrates seamlessly with the development workflow and supports Test-Driven Development (Beck, 2003).

# Program

The development of the Hangman game followed a Test-Driven Development (TDD) approach. Unit tests were created alongside the implementation of the core logic to ensure correctness, modularity, and maintainability

## Requirements Fulfilment

The Hangman game satisfies all the specified requirements:

* Two levels: Basic (random technology-related word) and Intermediate (random technology-related phrase).
* The answer is displayed as underscores, with spaces/punctuation preserved.
* Players have 15 seconds to guess; if the timer runs out, a life is deducted.
* Correct guesses reveal all occurrences of the letter in the answer.
* Wrong guesses deduct a life.
* Game ends when the word/phrase is fully guessed or lives reach zero.
* A hanging animation progressively draws as lives are lost.
* Game runs in a Tkinter GUI window (800x600) with in-game level selection.

# Process

## Design Choices

The program was designed with simplicity, readability, and modularity in mind. The code was split into three files for clarity:

1. hangman\_core\_logic.py – Contains the entire game logic, dictionary validation, and word/phrase banks. It is independent of any user interface to ensure easy testing.
2. hangman\_visual.py – Implements the Tkinter-based visual interface, including an 800x600 window, in-game level selection, countdown timer, and progressive hangman animation.
3. hangman\_logic\_test.py – Provides automated tests using unittest. Each requirement of the game was verified through unit tests, with print statements to display state transitions.

The automated unit testing done in the file hangman\_logic\_test.py covered the following key requirements:

1. **Dictionary Validation**

**Task:** Ensure that the words and phrases selected by the game are valid technology-related terms.  
**Description:** The test checks that the TECH\_WORDS and TECH\_PHRASES dictionaries are not empty. It also validates that answers chosen in Basic level are from TECH\_WORDS and answers in Intermediate level are from TECH\_PHRASES. This guarantees the random generator never produces invalid content.

1. **Letter Reveal on Correct Guess**

**Task:** Verify that when a player guesses a correct letter, all occurrences of that letter in the answer are revealed.  
**Description:** For example, with the answer "protocol", guessing "o" should reveal positions 2, 4, and 6. The test checks both the positions returned and the updated masked word, confirming multiple-letter handling.

1. **Life Deduction on Incorrect Guess**

**Task:** Ensure that wrong guesses reduce the player’s lives by one.  
**Description:** With an answer like "python", guessing a letter not present (e.g., "e") reduces the life count by one while keeping the masked word unchanged. This test verifies both the decrement in lives and that no letters are revealed incorrectly.

1. **Life Deduction on Timer Expiry (forfeit\_life)**

**Task:** Confirm that when the 15-second timer expires, a life is deducted without requiring a guess.  
**Description:** This functionality is simulated with a forfeit\_life method (GUI calls it when time runs out). The test ensures that this action reduces the life count and can eventually trigger a game loss if repeated.

1. **Win Condition Detection**

**Task:** Verify that the game correctly identifies a win when all letters have been revealed.  
**Description:** For a short word like "ai", after guessing 'a' and 'i', the game should mark is\_won() as True and is\_lost() as False. This proves the game stops correctly when the word is solved.

1. **Loss Condition Detection**

**Task:** Confirm that the game declares a loss when the player runs out of lives.  
**Description:** For example, with answer "cloud", making six incorrect guesses ("a", "b", "e", "f", "g", "h") exhausts the lives and sets is\_lost() to True. This test ensures the failure state is handled correctly.

1. **Invalid Input Handling**

**Task:** Ensure that invalid guesses (non-alphabetic characters or multi-character strings) are rejected safely.  
**Description:** The test attempts invalid inputs such as "1" or "ab" and expects a ValueError. This prevents crashes and enforces correct user input rules.

1. **Repeated Guess Handling**

**Task:** Verify that repeating the same letter does not penalize the player twice.  
**Description:** With answer "debugger", guessing "g" once reveals positions correctly. Guessing "g" again does not deduct lives or reveal anything new. This ensures fairness in gameplay and avoids punishing repeated guesses.

## Workflow

The workflow followed a simplified software engineering process with iterative cycles of requirements understanding, design, coding, and testing. Test-Driven Development (TDD) was applied, where unit tests were written and run alongside the development of core logic.

**Project Workflow Diagram**:

Implementation (Core Logic & GUI)

Design

Requirements

Unit Testing

Integration

Final Testing

## Test Cases

The following table summarizes key unit test cases executed using Python unittest:

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Description | Expected Result | Outcome |
| TC1 | Validate dictionaries not empty | Dictionaries contain tech words/phrases | Pass |
| TC2 | Basic level answer selection | Answer chosen from TECH\_WORDS | Pass |
| TC3 | Intermediate level answer selection | Answer chosen from TECH\_PHRASES | Pass |
| TC4 | Correct guess reveals letters | All positions of guessed letter revealed | Pass |
| TC5 | Wrong guess deducts a life | Life count decreases by 1 | Pass |
| TC6 | Repeated guess no penalty | Lives remain same, no new reveal | Pass |
| TC7 | Win condition check | Game detects victory when all letters guessed | Pass |
| TC8 | Lose condition check | Game detects loss when lives=0 | Pass |
| TC9 | Phrase masking check | Spaces remain visible in masked phrase | Pass |

## Output and Discussion

The pictures of the above discussed test cases of the game are inserted below along with the visual interface of the game.

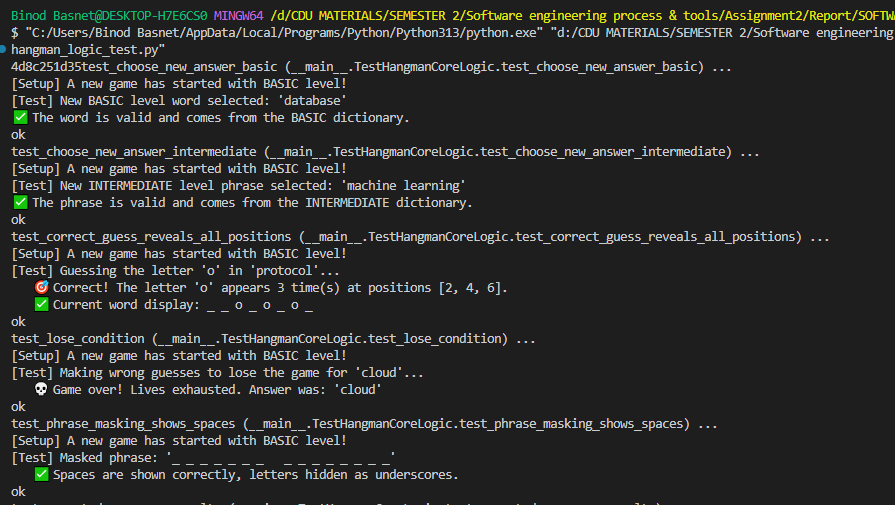


Figure 1 Test cases done on file hangman\_logic\_test.py

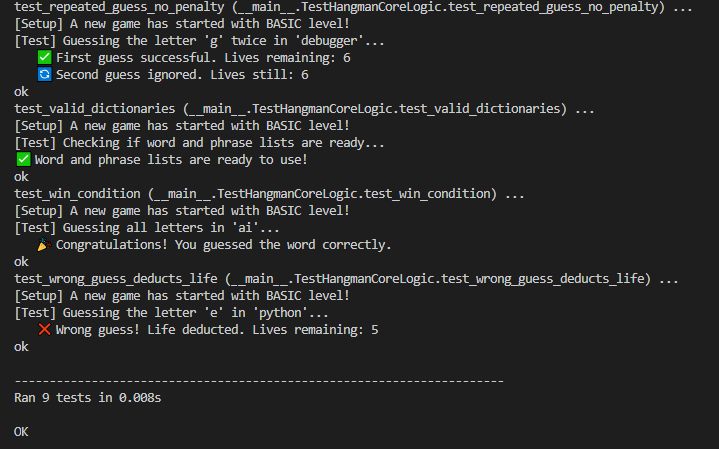


Figure 2 Test cases done on file hangman\_logic\_test.py

From the figure 1 and figure 2, It can be concluded that there are 9 tests executed in 0.008 seconds and all the tests are passed successfully without any errors.

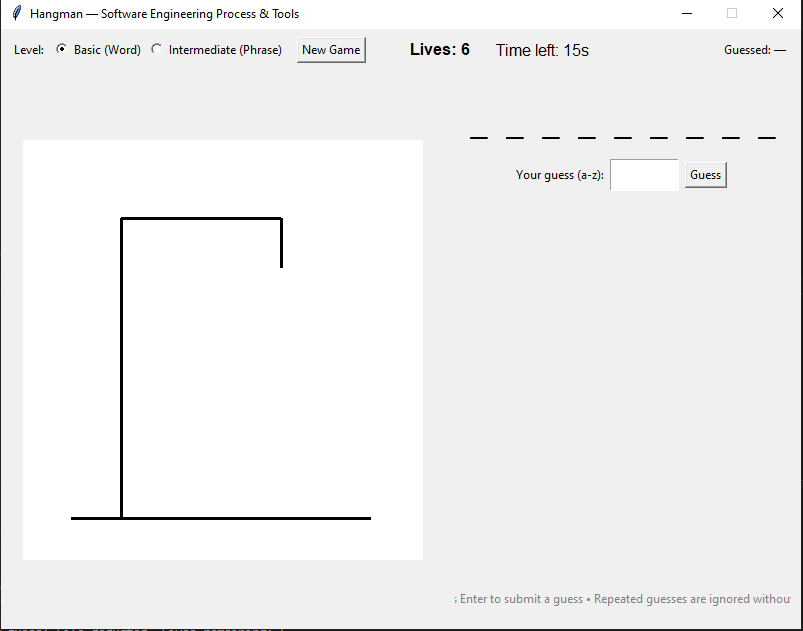


Figure 3 Interface of the game when file hangman\_visual.py is run

The figure 3 shows the interface of the hangman game which includes radio buttons to select level of the game, a button “New Game” to restart. Additionally, the interface displays the lives and time left along with Guessed letters. The answer is displayed as underscores, with spaces/punctuation preserved. The animation of hanging man if user fails to guess the word is simple and minimal.

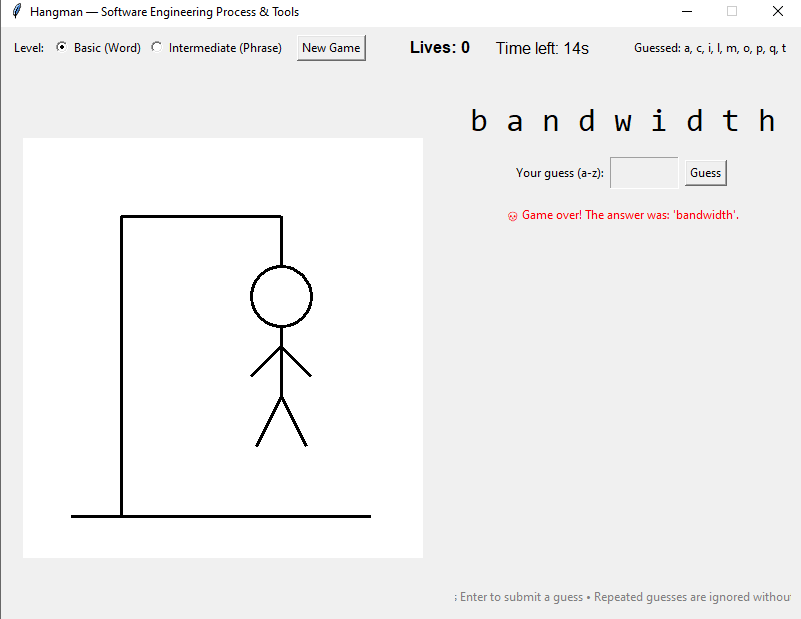


Figure 4 Interface of hanging animation when game is over

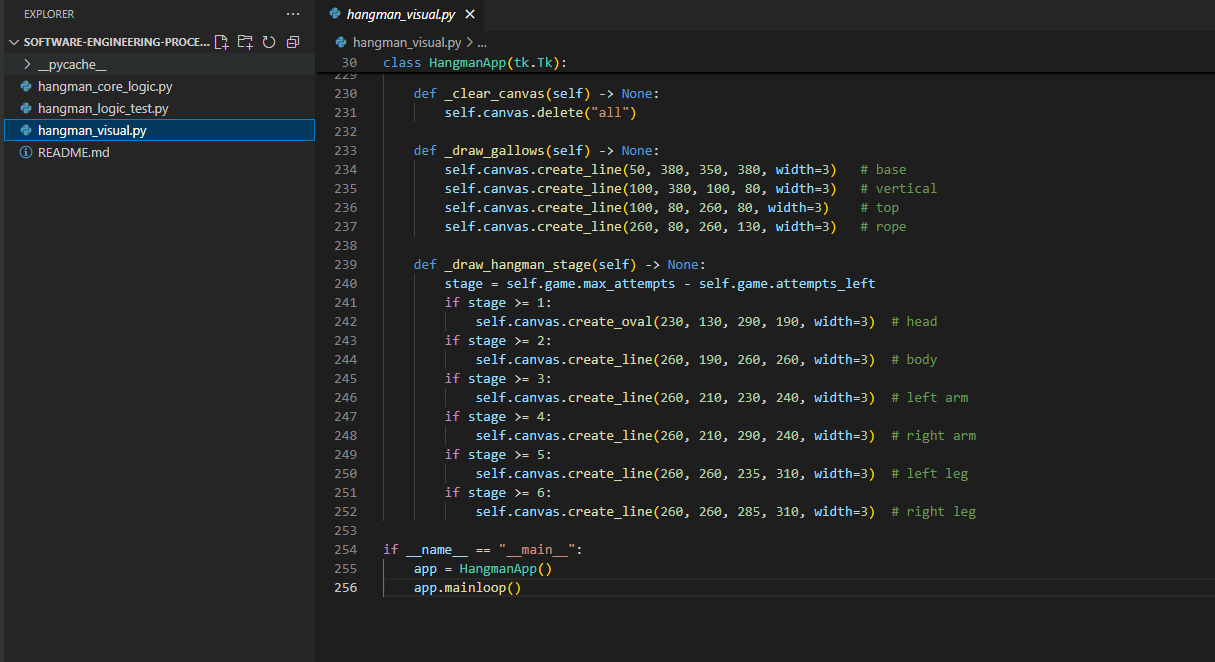


Figure 5 Structure of code, splitted into 3 files

# Conclusion

The Hangman game project successfully demonstrated the use of TDD and automated unit testing in software development. The separation of concerns between logic, visuals, and tests improved modularity and maintainability. Lessons learnt include the importance of isolating core logic from the user interface, handling edge cases early through tests, and the usefulness of linting tools such as flake8 and pylint to maintain code quality.

Future improvements for the Hangman game could focus on enhancing both functionality and user experience. The word and phrase bank could be expanded with a larger collection of technology-related terms, organized by categories such as networking, artificial intelligence, and cybersecurity. Multi-language support could be introduced by maintaining separate dictionary files for each language and allowing players to select their preferred language from the interface. Difficulty levels could be extended beyond the current Basic and Intermediate modes, for example by adjusting the timer length and number of lives to create Easy, Medium, and Hard modes. The game interface itself could be improved with better use of color, typography, and layout, while the hanging animation could be made smoother or replaced with more engaging visuals. Finally, sound effects could be added to provide feedback for correct and incorrect guesses, as well as victory or loss events. These enhancements could be implemented incrementally by leveraging Tkinter styling for visuals, dynamic file loading for dictionaries, and external libraries such as pygame or playsound for audio, thereby creating a more engaging and customizable game experience.  
  
The complete codebase, including this report, is maintained in a GitHub repository:

GitHub Repository Link: <https://github.com/Binod798/SOFTWARE-ENGINEERING-PROCESS-AND-TOOLS>

# References

Beck, K. (2003). *Test-driven development: By exampl*. Retrieved from Addison-Wesley Professional: https://doi.org/10.5555/861287

Hetzel, W. C. (2008). *The complete guide to software testing (3rd ed.)*. Retrieved from https://archive.org/details/completeguidetoso00hetz

van Rossum, G., & Drake Jr, F. (2009). *Python 3 reference manual*. Retrieved from https://docs.python.org/3/reference/