UNIT CODE: PRT 582
SOFTWARE ENGINEERING: PROCESS AND TOOLS

Software Unit Testing Report

Submitted by:
Kar Keat Koh (S394886)

Table of Contents

The full version of the game can be obtained from GitHub repository with the link below:

GitHub link: https://github.com/triplekkoh/hangman_tdd

SSH: git@github.com:triplekkoh/hangman_tdd.git

Repository set to private visibility with access granted to Charles.Yeo@cdu.edu.au

Introduction

The objective of this task is to try to create a classic Hangman game, which is a word-guessing game using Test Driven Development (TDD) in Python. It is expected to develop along with an automated unit testing tool available in Python. A list of requirements must be fulfilled for the development of this game. Players will have the choice between Basic and Intermediate levels, where they are tasked to find all the characters in a word for the Basic level and find all the characters in a phrase for the Intermediate level. Each play will grant the player a fixed amount of life, and it will be deducted when a wrong guess is made or the 15-second timer times out. All the words and phrases are valid words from a dictionary, and they will be hidden at the start of the play. Until a correct guess from the player, all characters will be represented by underscores. When the player correctly guesses a character, all the places in the answer that match the character will be revealed. In the case where the lives become zero or the player has correctly guessed all the characters, the answer will be revealed. Apart from the requirements, this game has been designed to give the players 6 lives for each play.

The Python "*unittest*" module has been selected as the automated unit testing tool for the development of this game. This module provides a large list of tools that can be used for creating and running the test. The constructed test cases can be easily executed in any command-line interface. A full documentation for the details of this tool is also available as a useful reference during the test case construction. This tool helps to ensure that the expected outcome for each of the functions is correctly identified before creating the actual functions.

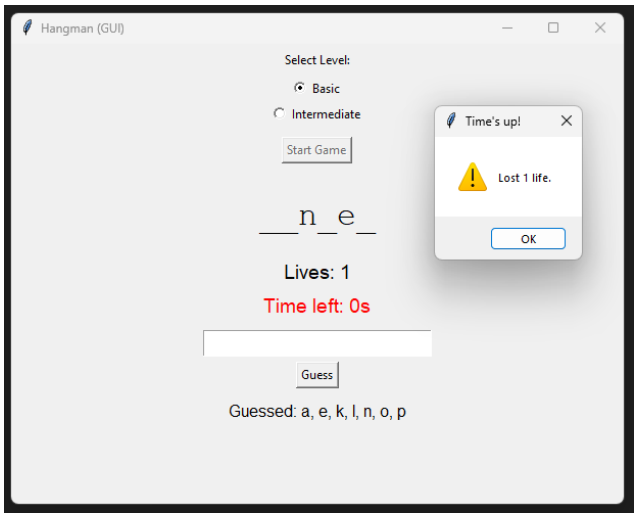


Figure 1 Screenshot of the game with Basic Level, and the timer times out. Answer: singer

Process

To create the Hangman game using TDD with an automated unit testing tool, a high-level process plan was created to ensure that the project development meets all the requirements. The flowchart below shows all 9 processes that took place when developing the game.

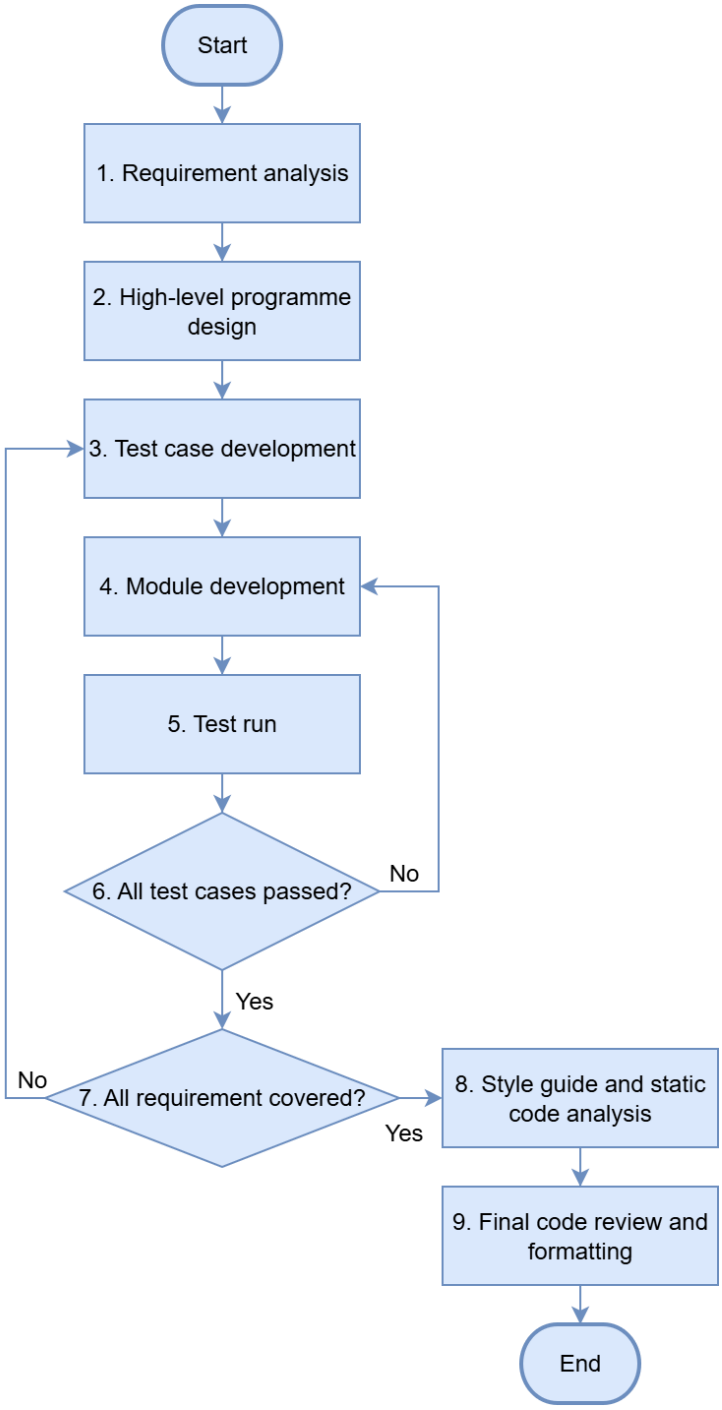


Figure 2 Flowchart of Hangman TDD

1. Requirement analysis

For the first process of the development, requirement analysis was done to identify the scope as well as looking for any missing information that will be needed for the game. 9 requirements had been identified, and a few assumptions were made.

Requirement	Details
1	Two levels of game difficulty are needed. Basic (Single word) and Intermediate (Phrases).
2	All words and phrases should be a valid word from dictionary.
3	Unguessed characters need to be shown as underscore "_"
4	A 15-second timer is needed for each guess.
5	The timer's timeout will cost the player one life.
6	Correct character guess reveals the matching characters in the answer.
7	An incorrect guess will cost the player one life.
8	The game ends when the player has zero life. Display answer.
9	The game ends when the player finds all the characters.

Table 1 Requirement table

The following assumptions are made, in addition to the requirements, to enhance the overall gaming experience.

Assumption	Details
1	Each play will grant the player 6 lives.
2	Players will not be allowed to make repeated characters guess.
3	No life deduction for repeated guesses.
4	Players will not be allowed to start a new game before the current game ends.
5	No character limitation for words and phrases.
6	Phrases can be created and stored locally to the program.
7	A pop-up is shown when the timer times out or to display the answer.
8	Only accepts alphabet characters. Other characters are considered invalid.

Table 2 Assumption table

2. High-level programme design

With requirements and assumptions identified, a high-level programme design was created to give a better understanding of the overall design of the programme. Decision was made to create 3 modules to cover the scope.

Selector module will be used to generate the words or phrases based on the selected level. A list of words will be extracted from a web page as dictionary for words. Phrases will be stored created manually and stored as a List.

Game module is responsible for masking all the unguessed characters, checking the player's guess and determine whether the game is won or lost. There will be 4 outcomes for each guess, which are Correct, Incorrect, Repeated and Invalid.

GUI module will help to create the interface where the player can interact with, using *tkinter*. Player will be able to choose the difficulty level, Basic or Intermediate and click the "Start Game" button to begin playing. There will be a word placeholder for the play to enter their guess. Timer should be visible to players to tell them how much time is left until they have to make a guess in order not to lose a life. This module will also have functions to start, reset and handle timer timeouts.

3. Test case development

After the design of the programme had been finalized, test cases were created to check that the modules were created according to their expectations. A total of 37 test cases were created by the end of development. They can be separated into 3 different sections corresponding to 3 different modules. All requirements were checked to ensure that there are sufficient test cases to cover them.

Section 1: Word Selection and difficulty level

1. test_check_default_level: Test default level values (Basic).
2. test_check_change_inter: Test changing Level values to intermediate.
3. test_level_change_basic: Test changing Level values back to Basic.
4. test_basic_pick_word: Test whether a word can be picked from the Basic level.
5. test_basic_pick_word_different: Test whether different words can be randomly picked from the Basic level.
6. test_inter_pick_phrase: Test whether a phrase can be picked from the Intermediate level.
7. test_inter_pick_phrase_different: Test whether different phrases can be picked from the Intermediate level.
8. test_invalid_level: Test that the invalid level handling .

Section 2: Hangman Game Logic

1. test_default_game_state: Test initial game state.
2. test_masked_initial: Test masked function returns underscores for unguessed letters.
3. test_masked_after_some_guesses: Test masked function after some correct guesses.
4. test_valid_guess: Test valid_guess function.
5. test_guess_letter_correct: Test guess_letter returns CORRECT for a correct guess.
6. test_guess_letter_incorrect: Test guess_letter returns INCORRECT for a wrong guess and deducts a life.
7. test_guess_letter_repeated: Test guess_letter returns REPEATED for a repeated guess.
8. test_guess_letter_invalid: Test guess_letter returns INVALID for invalid input.
9. test_guess_letter_case_insensitivity: Test guess_letter is case-insensitive.
10. test_masked_all_letters_guessed: Test masked function when all letters are guessed.
11. test_masked_with_repeated_letters: Test masked function with repeated letters in the answer.
12. test_is_won_true: Test is_won returns True when all letters are guessed.
13. test_is_won_false: Test is_won returns False when not all letters are guessed.

14. test_is_lost_true: Test is_lost returns True when lives reach zero and the game is not won.
15. test_is_lost_false: Test is_lost returns False when lives remain or the game is won.
16. test_time_out_deducts_life: Test time_out deducts one life.

Section 3: GUI-Related Logic

1. test_gui_setup: Set up a GUI instance with a dummy game for testing.
2. test_update_display: Test that the GUI display updates correctly.
3. test_start_game: Test that the GUI starts the game correctly.
4. test_reset_game: Test that the GUI resets the game correctly.
5. test_update_timer: Test that the timer updates correctly.
6. test_timer_expired: Test that the timer expiration is handled correctly.
7. test_reset_timer: Test that the timer resets correctly.
8. test_make_guess: Test that making a guess updates the game state.
9. test_make_repeated_guess: Test that making a repeated guess is handled correctly.
10. test_make_invalid_guess: Test that making an invalid guess is handled correctly.
11. test_timer_reset_after_guess: Test that the timer resets after a valid guess.
12. test_check_endgame_win: Test that the endgame is detected correctly on win.
13. test_check_endgame_loss: Test that the endgame is detected correctly on loss.

4. Module development

At this stage, functions in modules will be developed according to the expectations set in the test cases for each section. Development is done using Visual Studio Code with Python plugin installed.

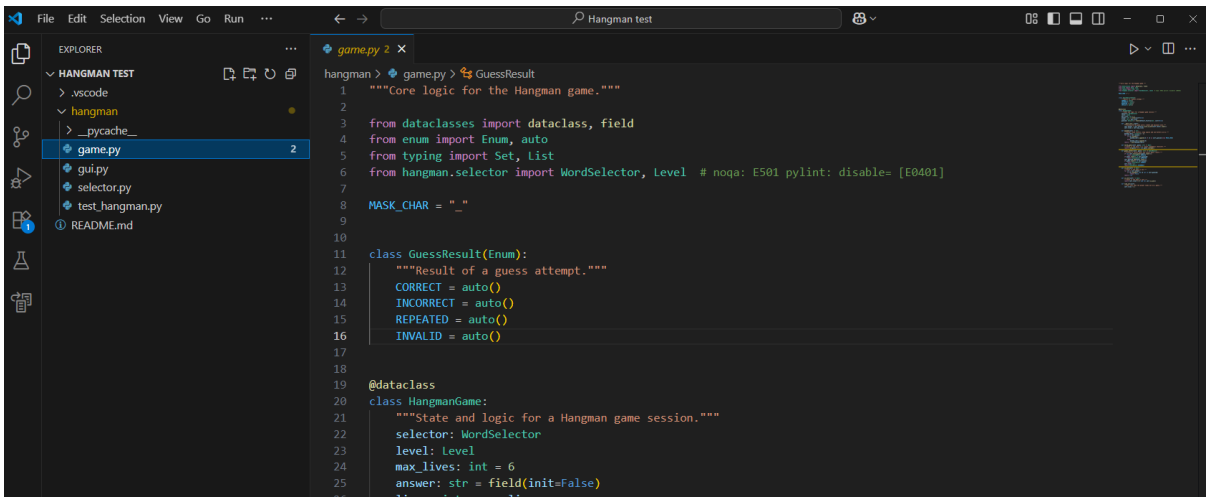


Figure 3 Visual Studio Code python development

5. Test run

Once the module function is created, a test case for that specific function will be executed to check whether the function fulfilled the requirements. The table below shows that the requirements can be linked to each of the test cases. Some of the test cases will cover multiple requirements. Details of the test cases will be available at the provided GitHub repository.

Requirement	Requirement	Test Case(s)
1	Two levels of game difficulty are needed. Basic (Single word) and Intermediate (Phrases).	test_check_default_level test_check_change_inter test_level_change_basic test_basic_pick_word test_inter_pick_phrase
2	All words and phrases should be a valid word from dictionary.	test_basic_pick_word test_basic_pick_word_different test_inter_pick_phrase test_inter_pick_phrase_different
3	Unguessed characters need to be shown as underscore "_".	test_masked_initial test_masked_after_some_guesses test_masked_all_letters_guessed test_masked_with_repeated_letters
4	15-second timer needed for each guess.	test_update_timer test_reset_timer
5	Timer time out will cost the player one life.	test_timer_expired test_time_out_deducts_life
6	Correct character guess reveals the matching characters in the answer.	test_guess_letter_correct test_masked_after_some_guesses test_masked_all_letters_guessed
7	Incorrect guess will cost the player one life.	test_guess_letter_incorrect test_guess_letter_case_insensitivity
8	Game ends when player has zero life. Display answer.	test_is_lost_true test_is_lost_false test_check_endgame_loss
9	Game ends when player finds all the characters.	test_is_won_true test_is_won_false test_check_endgame_win

Table 3 Test cases can be linked to specific requirement

6. Test result

Test cases will be executed at each function development in VS Code, and test results will be obtained after each run. Some diagnostic messages are printed when the test cases are executed in order to verify the randomness of the words and phrases generated. If the test result fails, the function will be reviewed to identify the root cause. However, there will be times when test cases have to be changed due to new or better approaches found during development that can achieve the same expectation.

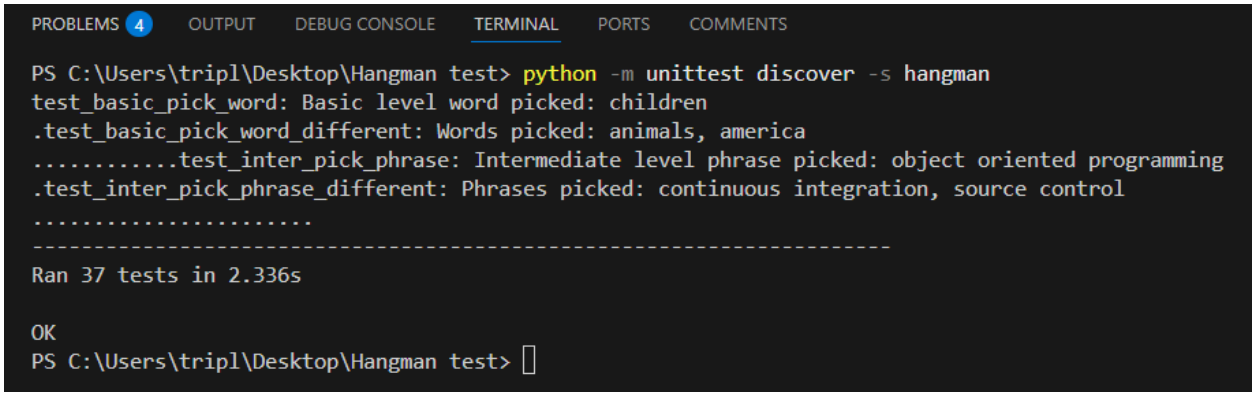
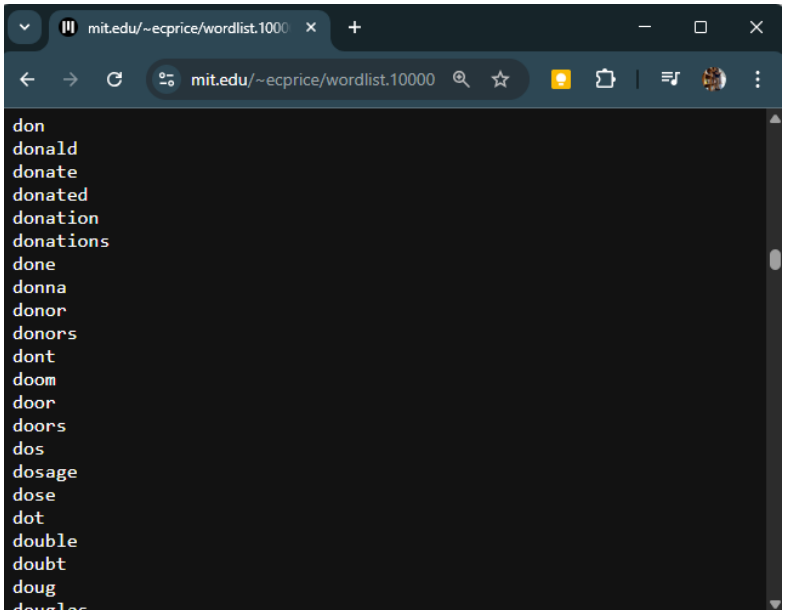


Figure 4 Test run results in VS Code

7. Requirement coverage check

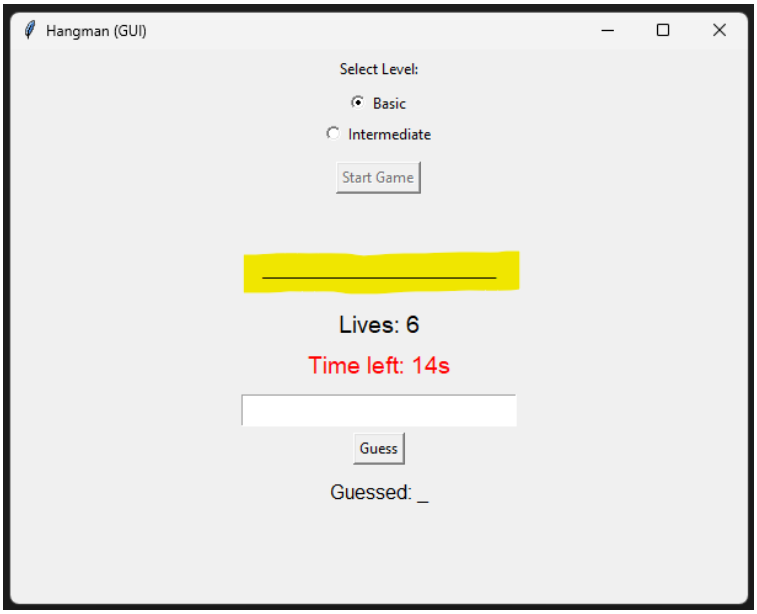
Once all the test cases were passed, requirement coverage checks were done once again to make sure the developed game fulfilled all the requirements. Below are the screenshots from the game that match the requirement.

Requirement 1: Two levels of game difficulty are needed. Basic (Single word) and Intermediate (Phrases). Two radio buttons are available for game level selection.



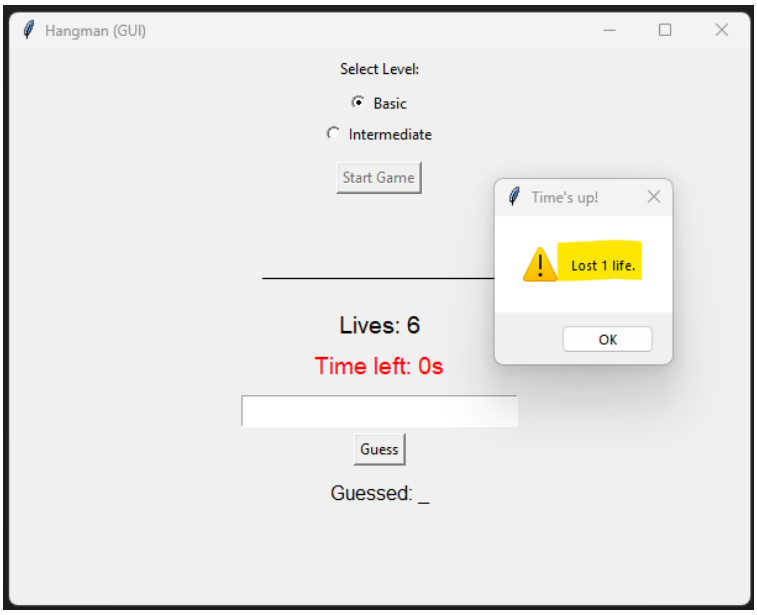

```python
# A list of intermediate phrases for intermediate players.
INTERMEDIATE_PHRASES: List[str] = [
    "agile methodology",
    "artificial intelligence",
    "continuous integration",
    "espresso machine",
    "data science",
    "northern territory",
    "object oriented programming",
    "personal computer",
    "test driven development",
    "user experience",
    "machine learning",
    "cloud computing",
    "source control",
    "information system",
    "network security",
    "virtual reality",
    "resource management",
    "debug mode",
    "memory leak",
    "syntax error"
]
```

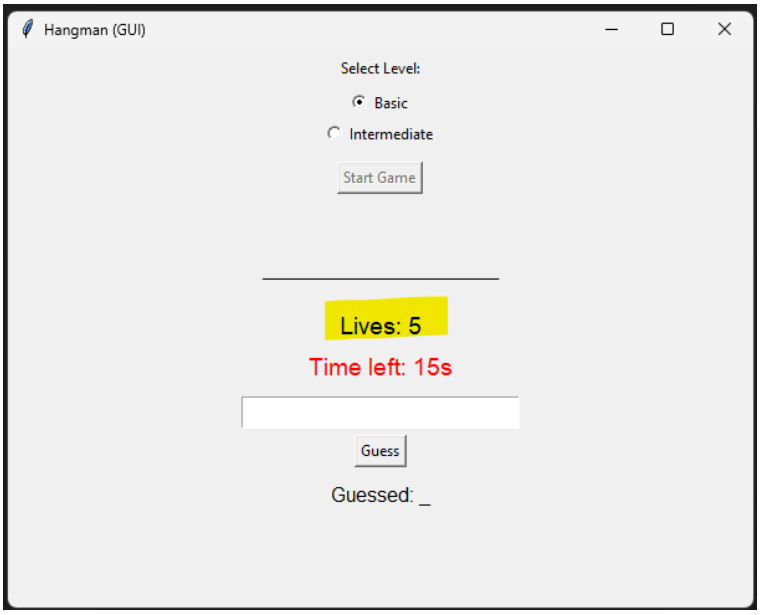
Requirement 2: All words and phrases should be valid words from the dictionary. A list of 10000 different words is obtained from an online source, and phrases are created locally.



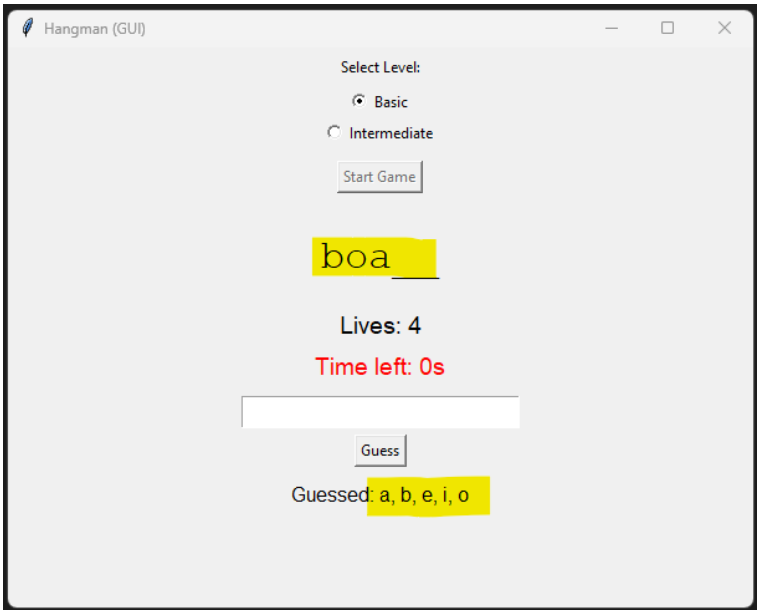
Requirement 3: Unguessed characters need to be shown as underscore "_". Since no guess has been made yet, only underscores are shown.

Requirement 4: 15-second timer needed for each guess.

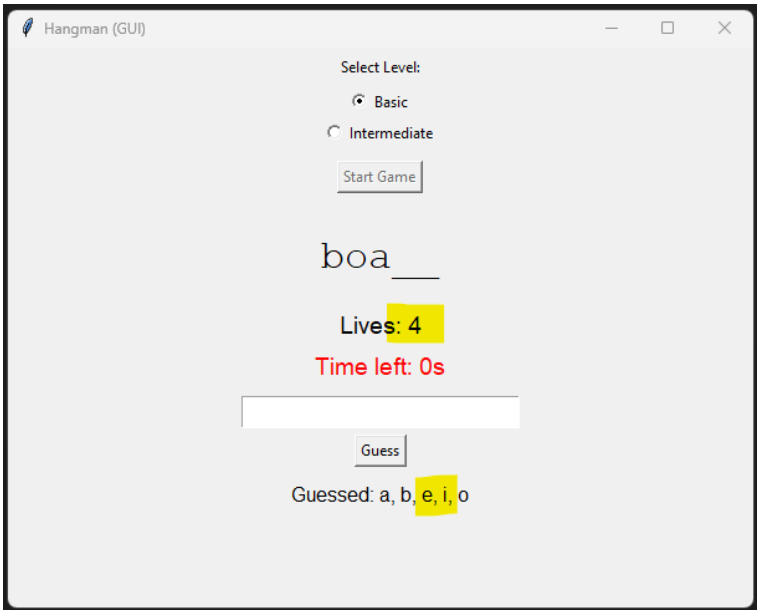




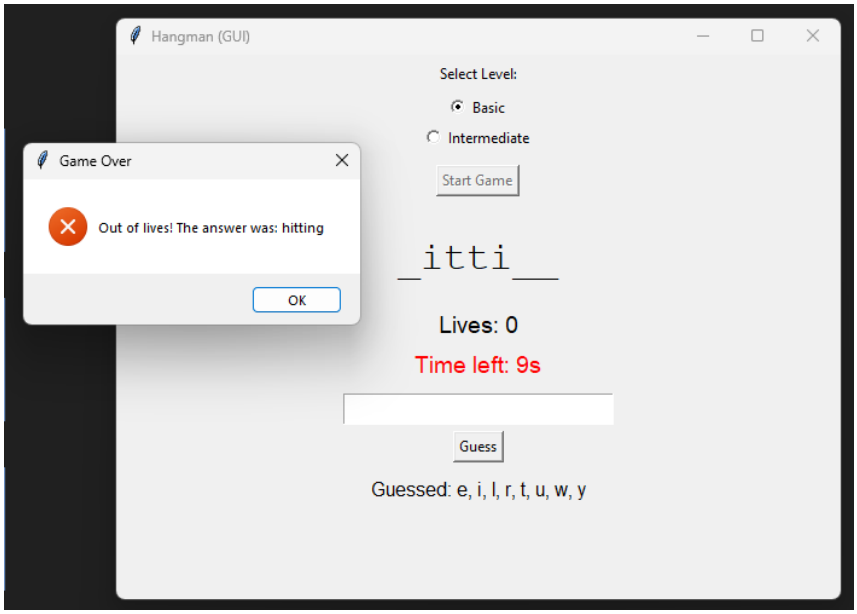
Requirement 5: Timer time-out will cost the player one life. A pop-up is shown to indicate that a life is lost due to the timer timing out.



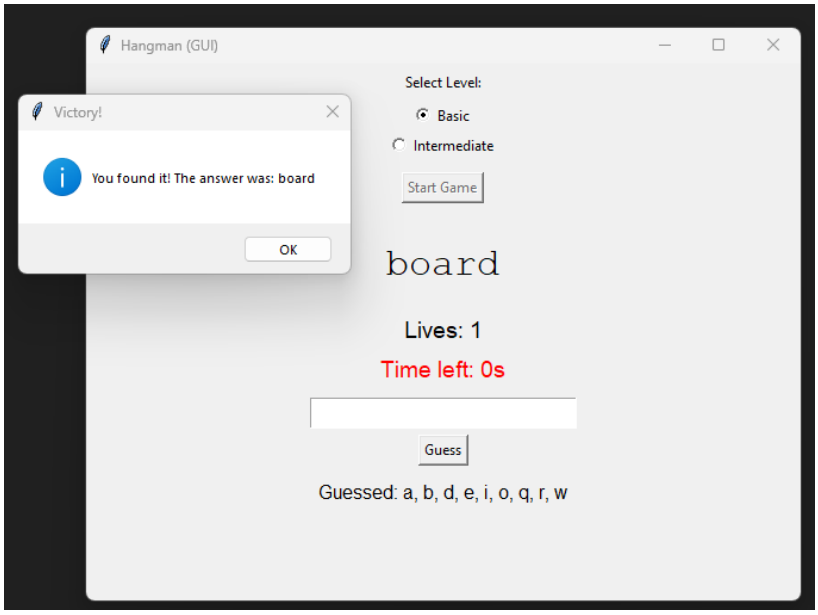
Requirement 6: Correct character guess reveals the matching characters in the answer.



Requirement 7: An Incorrect guess will cost the player one life. (Two lives were deducted due to two wrong guesses.)



Requirement 8: Game ends when the player has zero life, and the answer will be displayed.



Requirement 9: Game ends when the player finds all the characters.

8. Style guide and static code analysis

Pylint, as a static code analysis tool, has been used throughout the development and at the end of development to make the source code follow the correct format and correct Python logic. *Flake8* was also used to make sure that the developed code follows the standard style guide to ensure better readability.

```
PS C:\Users\tripl\Desktop\Hangman test> python -m pylint ./hangman

--------------------------------------------------------------------
Your code has been rated at 10.00/10 (previous run: 10.00/10, +0.00)
```

Figure 5 Pylint score.

```
PS C:\Users\tripl\Desktop\Hangman test> python -m flake8 ./hangman
./hangman\gui.py:147:11: W292 no newline at end of file
PS C:\Users\tripl\Desktop\Hangman test> python -m flake8 ./hangman
PS C:\Users\tripl\Desktop\Hangman test>
```

Figure 6 Example of flake8 fix.

9. Final code review and formatting

At the final process, a final code review was done to ensure the game works correctly without any runtime errors. Code arrangement and format were reviewed again to ensure readability and understandability with the use of *Pylint* and *Flake8*.

Conclusion

Test-Driven Development (TDD) has demonstrated its effectiveness in maintaining alignment between the game's design and its implementation throughout the development process. Most of the challenges encountered were successfully addressed through independent research and the use of community forums. Although some test cases were modified during the development, the initial planning and test case design definitely provided a strong guideline for the development phase. After applying TDD, I can safely say that incremental development helps the developer to focus on breaking down a complex problem into small targets, and this also helps to avoid the risk of major bugs. Nonetheless, there is still room for multiple improvements in the future in terms of better overall graphics design, immersive sound effects, theme-based riddles and richer animation.