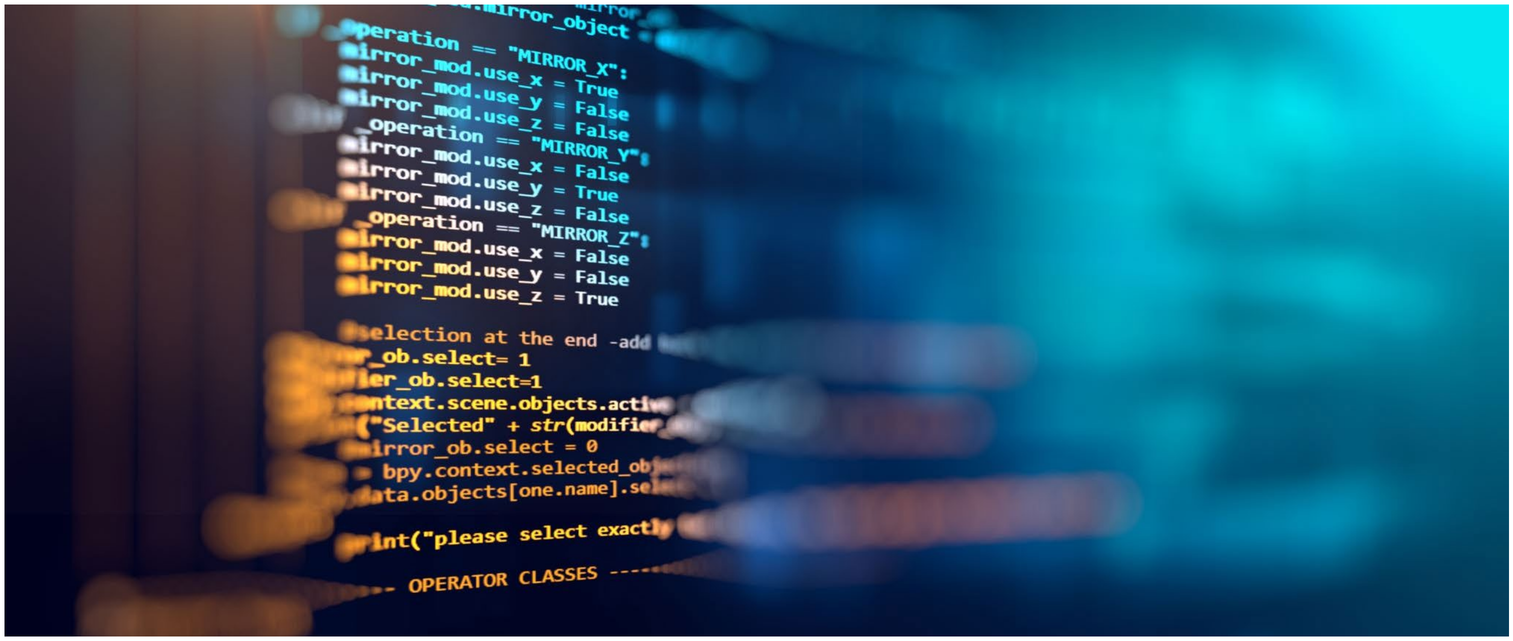


PRT582 Software Engineering: Process and Tools

# Software Unit Testing Report

Prepared by

Abu Saeed Md Shafiqur Rahman (Shafiq Rahman)

Student ID: S386795

Campus: External

***5 September 2025***

## Introduction

The purpose of this project is to design and implement a Hangman game in Python using a **Test-Driven Development (TDD)** approach combined with an automated unit testing framework. The key objectives are to:

1. Develop a modular, sustainable, and testable program.
2. Demonstrate the use of TDD as a software engineering methodology.
3. Implement automated testing to validate requirements.
4. Provide a user-friendly interface through a graphical user interface (GUI).

The program requirements included two difficulty levels (basic words and intermediate phrases), timer functionality with penalties for exceeding time, validation of user guesses, and win/loss conditions.

The programming language chosen for this project is **Python 3**. Python is widely used in both academia and industry due to its readability, flexibility, and strong support for software testing frameworks (Lutz, 2021). The automated unit testing tool used was **unittest**, a standard Python library that supports TDD workflows (Kuhn and MacDonald, 2020). Additionally, the GUI was implemented using **Tkinter**, which provides a lightweight way of building graphical applications in Python.

This report outlines the process of developing the Hangman game through TDD, presents supporting evidence in the form of screenshots (placeholders provided), and concludes with lessons learned from the implementation.

## Process

### TDD and Automated Testing

Test-Driven Development (TDD) is a software engineering methodology that emphasises writing tests before writing the functional code (Beck, 2003). The cycle typically follows three steps:

1. Write a **failing test** (Red).
2. Write the **minimal code** to pass the test (Green).
3. **Refactor** for clarity and maintainability.

Using this approach ensures that every feature in the Hangman game is verified against predefined requirements, reducing the risk of defects and ensuring modular, reusable code (Janzen and Saiedian, 2005).

The **unittest** framework was chosen because it is:

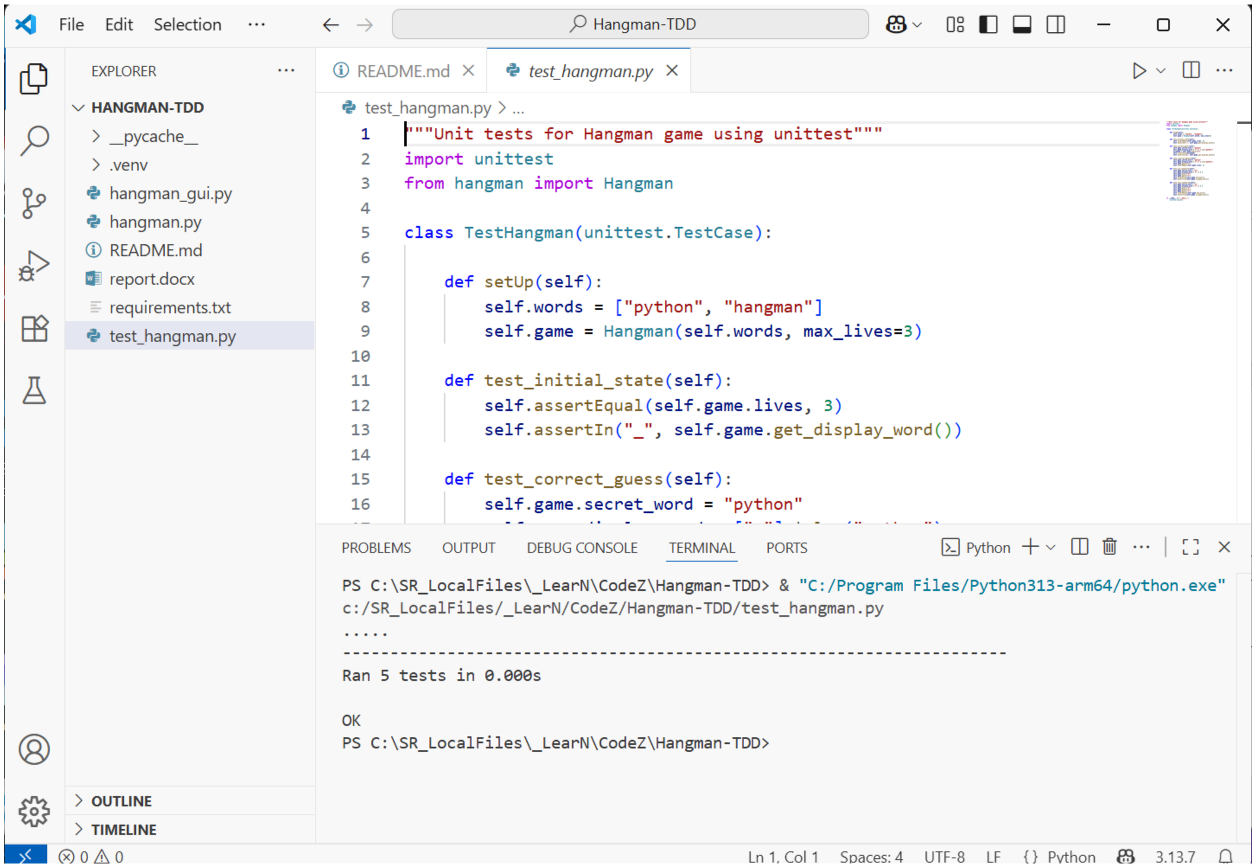
- Integrated with Python by default.
- Provides automated test discovery and execution.
- Produces structured test reports.
- Supports both positive and negative test cases.

This was especially suitable for verifying core logic (word generation, guesses, timeouts, win/loss conditions) independently of the GUI.

## Requirement-by-Requirement Implementation Using TDD

### Requirement 1: Two levels (basic and intermediate)

- **Test**: Unit tests were written to confirm that random words and random phrases could be generated. The phrase-level test ensured that spaces were correctly displayed as blanks.
- **Code**: Implemented in Hangman.random_word() and Hangman.random_phrase(). The constructor accepted either word or phrase.



```python
"""Unit tests for Hangman game using unittest"""
import unittest
from hangman import Hangman

class TestHangman(unittest.TestCase):

    def setUp(self):
        self.words = ["python", "hangman"]
        self.game = Hangman(self.words, max_lives=3)

    def test_initial_state(self):
        self.assertEqual(self.game.lives, 3)
        self.assertIn("_", self.game.get_display_word())

    def test_correct_guess(self):
        self.game.secret_word = "python"
```

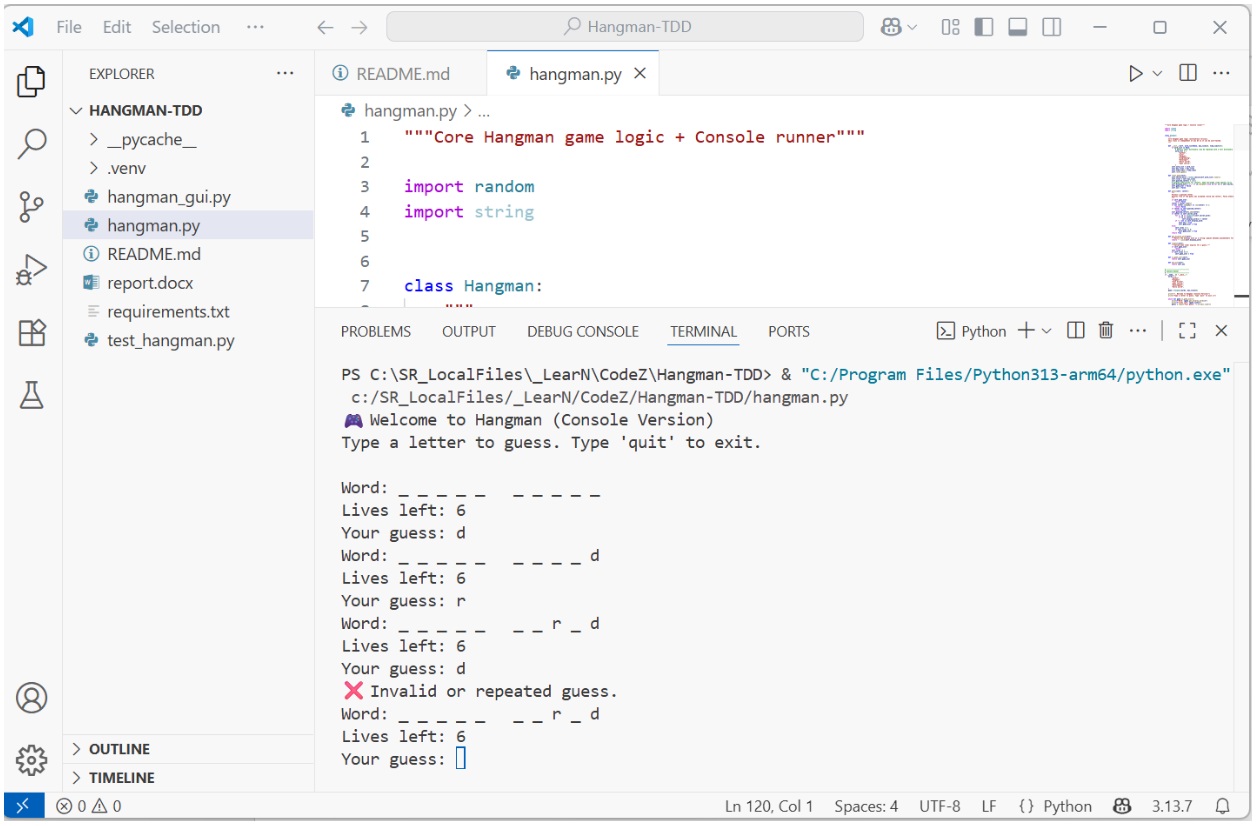
```
PS C:\SR_LocalFiles\_LearN\CodeZ\Hangman-TDD> & "C:/Program Files/Python313-arm64/python.exe" c:/SR_LocalFiles/_LearN/CodeZ/Hangman-TDD/test_hangman.py
.....
----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
PS C:\SR_LocalFiles\_LearN\CodeZ\Hangman-TDD>
```

*Figure 1 – Passing unit tests for word and phrase generation.*

## Requirement 2: Dictionary-based validity

- **Test**: Confirmed that generated words/phrases belonged to predefined dictionaries.
- **Code**: Both random_word and random_phrase methods draw from curated lists. These lists could later be replaced with an external dictionary API.



```python
"""Core Hangman game logic + Console runner"""

import random
import string


class Hangman:
```

```
PS C:\SR_LocalFiles\_LearN\CodeZ\Hangman-TDD> & "C:/Program Files/Python313-arm64/python.exe"
 c:/SR_LocalFiles/_LearN/CodeZ/Hangman-TDD/hangman.py
🎮 Welcome to Hangman (Console Version)
Type a letter to guess. Type 'quit' to exit.

Word: _ _ _ _ _  _ _ _ _ _
Lives left: 6
Your guess: d
Word: _ _ _ _ _  _ _ _ _ d
Lives left: 6
Your guess: r
Word: _ _ _ _ _  _ _ r _ d
Lives left: 6
Your guess: d
❌ Invalid or repeated guess.
Word: _ _ _ _ _  _ _ r _ d
Lives left: 6
Your guess:
```

*Figure 2 – Console output showing randomly selected valid words/phrases.*

## Requirement 3: Displaying underscores for unguessed letters

- **Test**: Asserted that the word "python" initially displayed as ______. For phrases, spaces were preserved (hello world → _____ _____).
- **Code**: Implemented in Hangman.display_word().

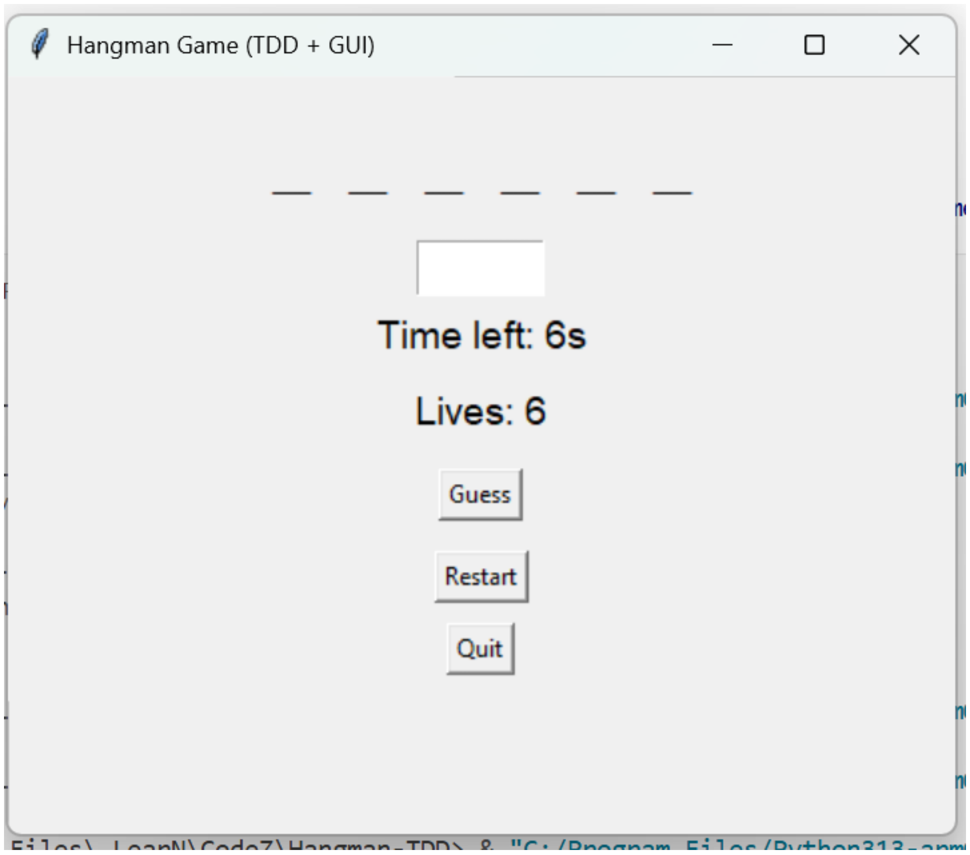
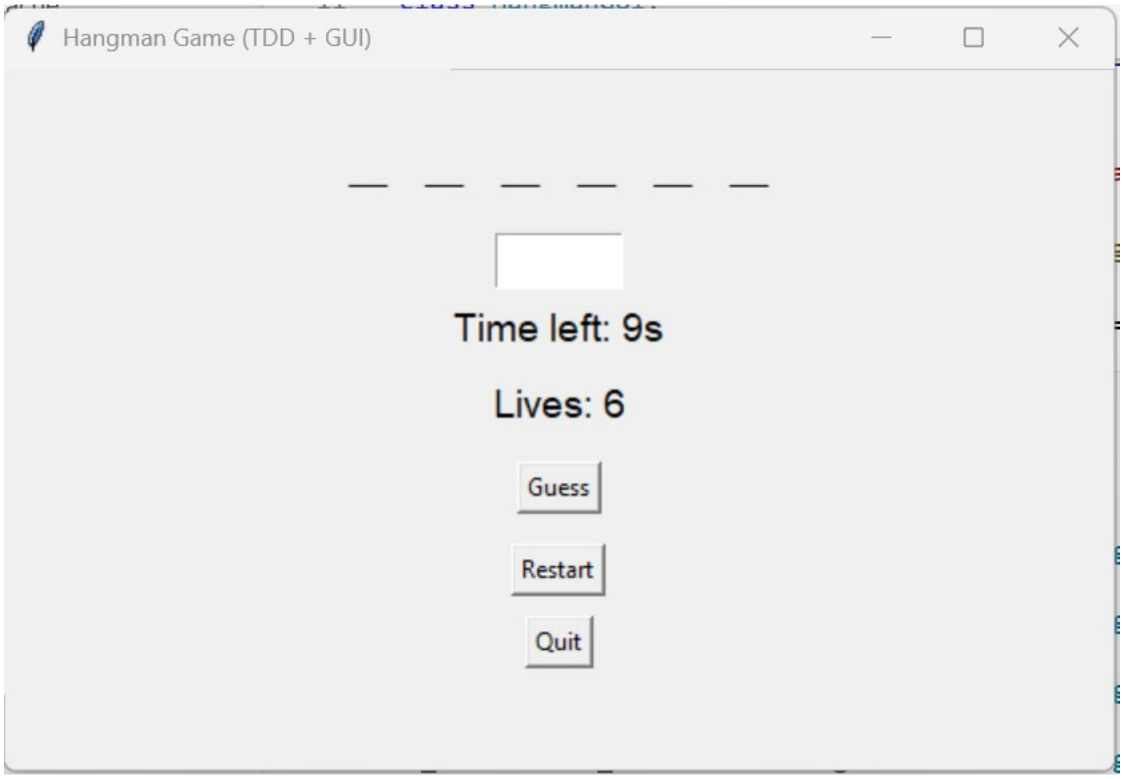


*Figure 3 – GUI display showing underscores at game start.*

## Requirement 4: Timer functionality (15 seconds)

- **Test**: Simulated timeout by calling game.timeout(), reducing lives by one.
- **Code**: Timeout implemented in logic; GUI countdown timer in hangman_gui.py triggers timeout when time expires.



```
Time left: 9s
Lives: 6
```

*Figure 4 – Timer countdown label in GUI.*

## Requirement 5: Revealing all occurrences of guessed letter

- **Test**: Guessed p in "python" → displayed p_____. Guessed o in "hello world" → revealed both os.
- **Code**: guess() method updates guessed set and reveals multiple positions.

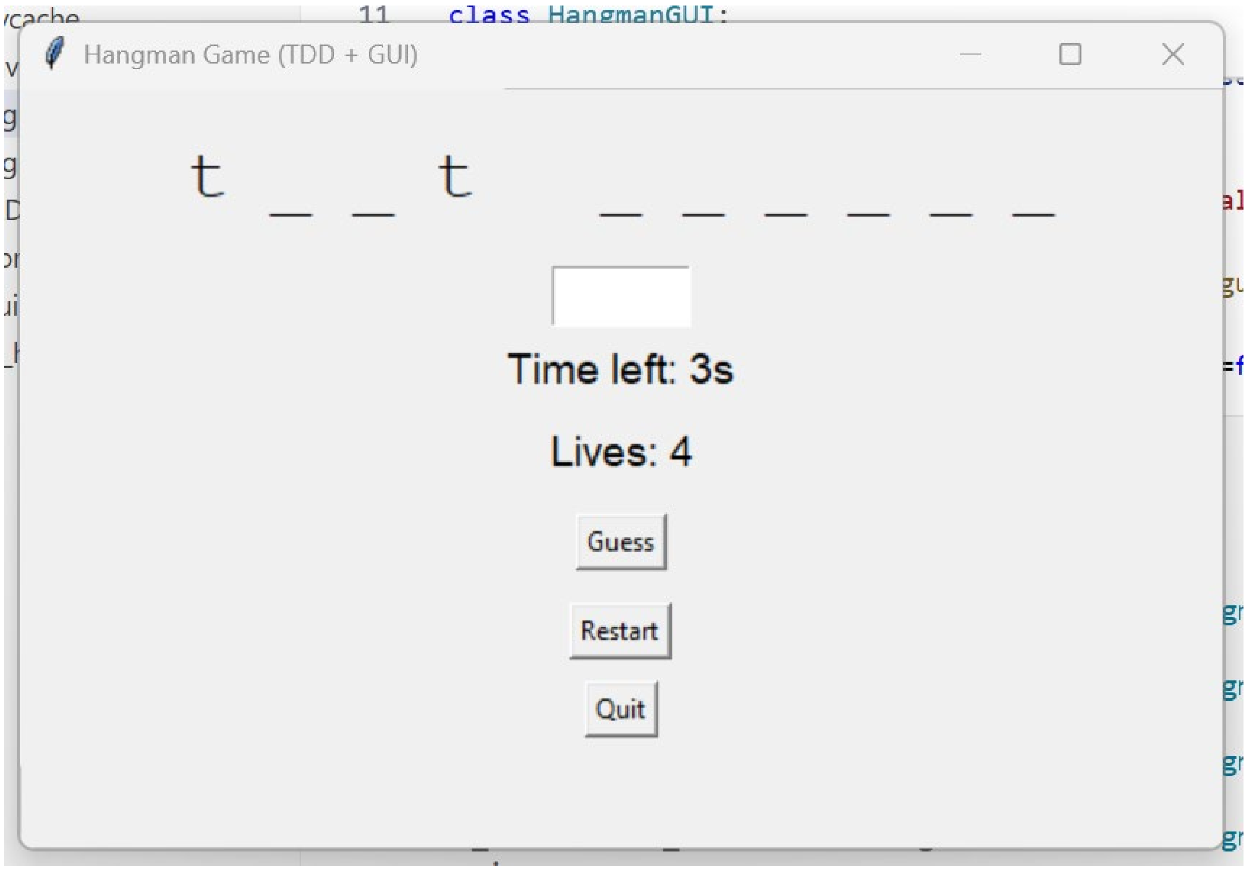
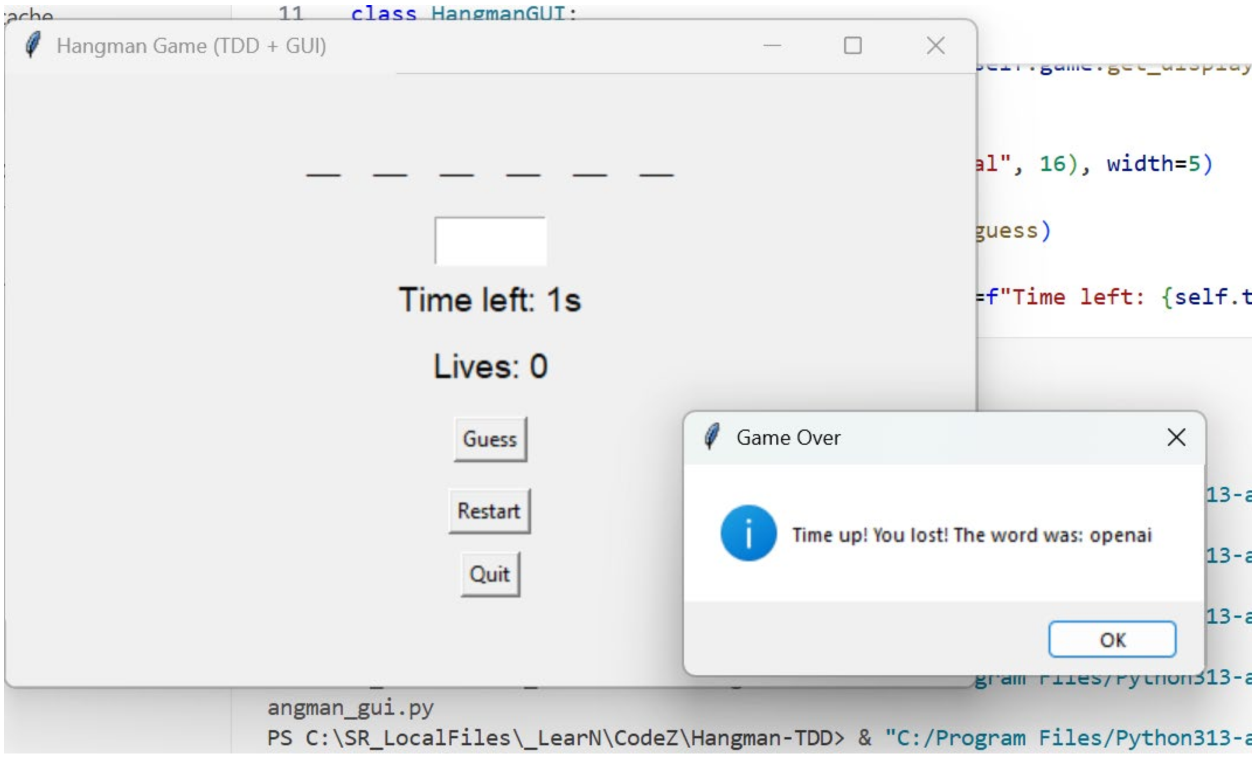


*Figure 5 – GUI after correct guess.*

## Requirement 6: Deducting life on wrong guess

- **Test**: Guessing x reduced attempts from 6 to 5.
- **Code**: Deduction implemented in guess() method.



Hangman Game (TDD + GUI)

_ _ _ _ _ _

Time left: 1s

Lives: 0

Game Over

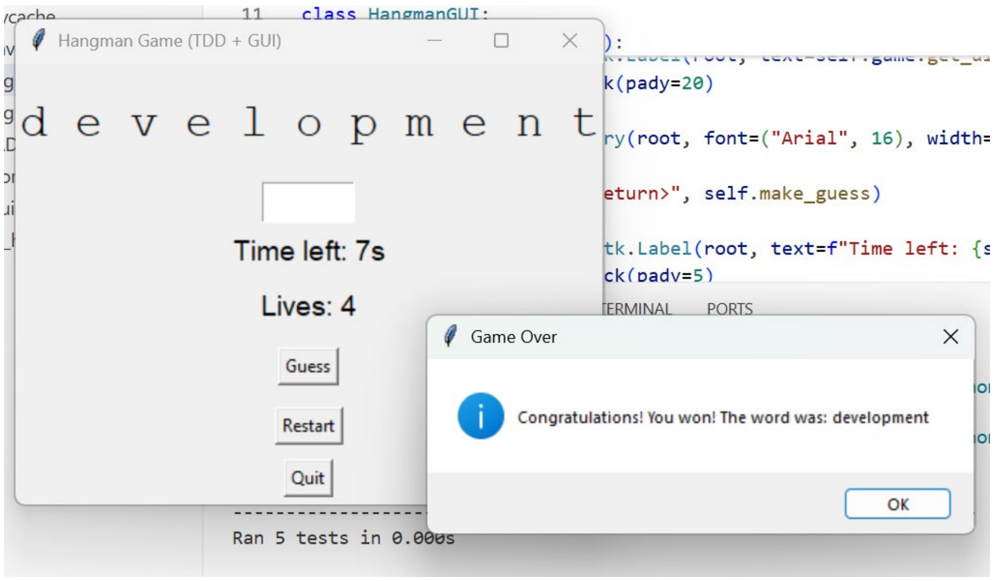
Time up! You lost! The word was: openai

```
class HangmanGUI:
al", 16), width=5)
guess)
=f"Time left: {self.t
angman_gui.py
PS C:\SR_LocalFiles\_LearN\CodeZ\Hangman-TDD> & "C:/Program Files/Python313-a
```

*Figure 6 – Attempts counter decreasing in GUI.*

## Requirement 7: Win/Loss conditions

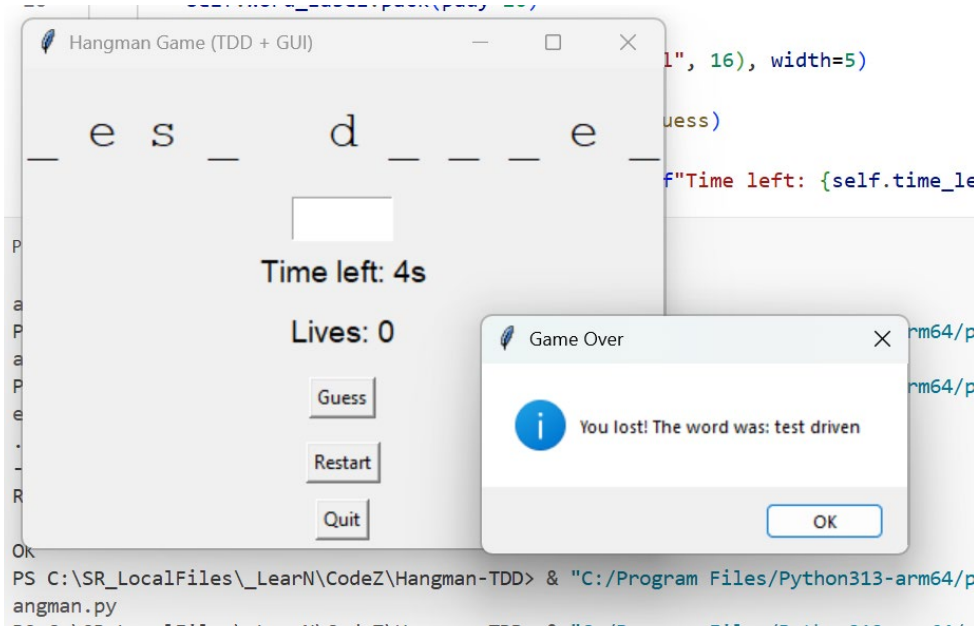
- **Test**: Guessed all letters correctly → is_won() returns True. Wrong guesses exceeding limit → is_lost() returns True.
- **Code**: Implemented in is_won() and is_lost().



Congratulations! You won! The word was: development

```
class HangmanGUI:
k(pady=20)
ry(root, font=("Arial", 16), width=
eturn>", self.make_guess)
tk.Label(root, text=f"Time left: {s
ck(pady=5)
Ran 5 tests in 0.000s
```

*Figure 7 – GUI popup for win message;*



You lost! The word was: test driven

```
l", 16), width=5)
uess)
f"Time left: {self.time_le
OK
PS C:\SR_LocalFiles\_LearN\CodeZ\Hangman-TDD> & "C:/Program Files/Python313-arm64/p
angman.py
```

*Figure 8 – GUI popup for loss message.*

### Requirement 8: Game continues until quit/win/loss

- **Test**: Verified restart functionality resets attempts and generates new word/phrase.
- **Code**: Restart button in GUI calls start_game().

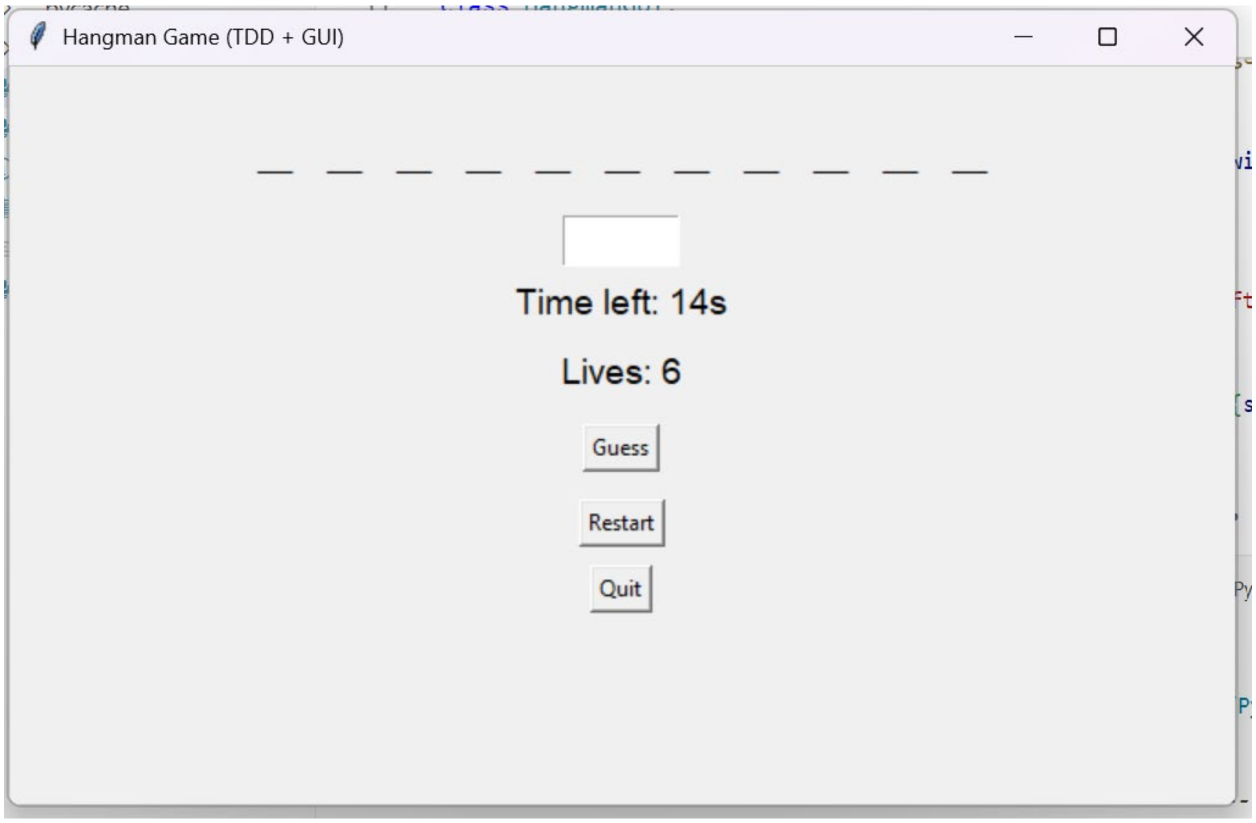


*Figure 9 – GUI showing restart functionality.*

## Code Quality, Style, and Sustainability

- **Modular Design**: Core logic (hangman.py) is separated from interface (hangman_gui.py), supporting reusability and testing.
- **Linting**: Code was checked with **flake8** and **pylint**, with issues addressed (style, spacing, naming conventions).
- **Documentation**: Docstrings and inline comments explain methods.
- **Error Handling**: Input validation prevents invalid entries (e.g., multiple characters, digits).

## Conclusion

This project demonstrated the design and development of a Hangman game in Python using TDD and automated testing. The **unittest** framework ensured requirement coverage, while the **Tkinter GUI** provided a user-friendly interface.

**Lessons learned:**

- TDD ensures discipline in coding by requiring clear expectations before implementation.
- Automated unit testing improves reliability and enables quick regression checks.
- Separation of logic and interface results in more maintainable code.
- Time constraints and GUI interaction require careful handling of concurrency (e.g., Tkinter timer threads).

**Areas for improvement:**

- Expand dictionary sources to improve word variety (e.g., API integration).
- Enhance GUI with graphics for a traditional "hangman" drawing.
- Add multiplayer or scoring features for extended play.

The complete codebase and report are available in the accompanying GitHub repository:

https://github.com/shafiqsaeed/Hangman-TDD

## References

Beck, K. (2003). Test Driven Development: By Example. Addison-Wesley.

Janzen, D. and Saiedian, H. (2005). Test-driven development: Concepts, taxonomy, and future direction. Computer, 38(9), pp.43–50.

Kuhn, B. and MacDonald, D. (2020). Python Testing with unittest. O'Reilly Media.

Lutz, M. (2021). Learning Python, 5th Edition. O'Reilly Media.