SOFTWARE ENGINEERING: PROCESS AND TOOLS [PRT582]

SOFTWARE UNIT TESTING REPORT

Submitted By

Mahesh Chandra Regmi [SID: S389242]

Submitted To Charles Yeo

CHARLES DARWIN UNIVERSITY,
FACULTY OF SCIENCE AND TECHNOLOGY,
DANALA | EDUCATION AND COMMUNITY PRECINCT,
AUSTRALIA

3 Sept 2025

Table of Contents

1. Introduction

1.1. Project Overview

Hangman is an old school favourite, a word game where the goal is to find the missing word. In this report, I have developed a simple console-based Hangman game using Python Programming language in a test-driven development (TDD) approach. The game has two difficulty levels (Basic and Intermediate) with a timer system, includes input validation and a lives-based mechanism.

1.2. Project Objectives

The goal of this project is to build a word guessing game using Test-Driven Development (TDD) methodology with automated unit testing. The primary objectives for this project are listed below.

- Apply the Red-Green-Refactor cycle throughout the entire development process.
- Write unit tests to cover all the game functionality.
- Build a fully functional word guessing game that meets all the specified requirements.

1.3. Project Requirements

I have developed the following requirements for the Hangman game.

Requirement	**Description**	**Implementation Status**
R1: Two Difficulty Levels	Basic (Single Words) and Intermediate (Phrases)	COMPLETE
R2: Valid Dictionary Words	Words/Phrases must be from predefined dictionary	COMPLETE
R3: Visual Display	Show underscores for missing letters	COMPLETE
R4: 15-Second Timer	Timer with visible countdown per guess	COMPLETE
R5: Letter Reveal	Reveal all correctly guessed letters	COMPLETE
R6: Lives System	Deduct life for wrong guesses.	COMPLETE
R7: Win/Loss System	Game continues until win or lives = 0	COMPLETE
Game Loop	Continue until quit, show answer then.	COMPLETE

1.4. Technical Overview

1.4.1. Programming Language Justification

I used Python programming language for this project because of the following reasons.

- Python's syntax is very easy to learn and understand. It allows me to focus on the game logic rather than complex language constructs such as those in system level languages like Rust, C etc.
- Python provides the ***unittest*** module as a part of the standard library. This module is very useful and self-sufficient for running tests as well as writing tests. No external dependencies are required for testing.
- The standard library of python is very broad and covers most of the functionalities we need. It provides libraries like ***random*** for word selection, ***time*** for timer functionality and ***enum*** for state management.

Apart from these benefits, Python is a cross-platform programming language. So, our game will run consistently across Windows, macOS and Linux systems.

1.4.2. Automated Unit Testing Tool Justification

I selected Python's ***unittest*** framework as the automated testing tool for this project. This choice was made because of the following reasons.

- ***unittest*** is a built-in framework which doesn't require installing any extra dependencies and ensures consistent testing environment.
- It has standard testing features such as fixtures, assertions, test discovery and detailed reporting.
- It is widely used in industry for Python development.

This framework enabled creation of 21 test cases achieving 60% test-to-code ratio.

2. Process

I used the red-green-refactor cycle throughout the development phase of all the components in this project.

Red Phase: Write failing test
Green: Write minimal code to pass the test
Refactor: Improve code quality.

2.1. TDD Methodology

2.1.1. Red Phase Implementation

In the red phase, every test failed. This ensures the test was testing the desired functionality. Tests are written using the ***unittest*** framework as mentioned earlier, with clear descriptive names following the pattern *test_<functionality>_<behaviour>*. Each test only focused on one part of the program. This ensures clarity and isolation of the tests. We utilized mock objects to isolate units that are under tests from external dependencies.

2.1.2. Green Phase Implementation

In the green phase, I wrote minimal implementation to get the tests go from red to green (failing to passing). No additional functionality was implemented beyond what was the actual need to get the tests passing. Each implementation was verified by running the test suite to ensure there are no regressions.

2.1.3. Refactor Phase Implementation

In this phase, I refactored the code to make it suitable for the new requirements. It includes naming methods appropriately, extracting functionalities and enhancing readability of the code. Performance optimizations are applied as well in the refactor phase but existing functionality is kept intact.

2.2. Implementation

2.2.1. Requirement R1 and R2

For the first two requirements, I wrote two simple tests to verify if we can construct a object of the Hangman class first. Then after, I checked if the game level stored in that class matches the level I set. I also added one more test to ensure the target of the game is a single word for basic game and multi-word for intermediate game.

Here is the code screenshot of the test. I saved this test in a file *test_hangman.py*.

```python
import unittest
import time
from unittest.mock import patch
from hangman import HangmanGame, GameLevel, GameState


class TestHangmanGame(unittest.TestCase):

    def setUp(self):
        self.basic_game = HangmanGame(GameLevel.BASIC)
        self.intermediate_game = HangmanGame(GameLevel.INTERMEDIATE)

    def test_game_initialization_basic(self): # Requirement: R1 & R2
        game = HangmanGame(GameLevel.BASIC)
        self.assertEqual(game.level, GameLevel.BASIC)
        self.assertEqual(game.lives, 6)
        self.assertEqual(game.state, GameState.PLAYING)
        self.assertIsNotNone(game.target)
        self.assertFalse(' ' in game.target)

    def test_game_initialization_intermediate(self): # Requirement: R1 & R2
        """Test that intermediate game initializes correctly."""
        game = HangmanGame(GameLevel.INTERMEDIATE)
        self.assertEqual(game.level, GameLevel.INTERMEDIATE)
        self.assertEqual(game.lives, 6)
        self.assertEqual(game.state, GameState.PLAYING)
        self.assertIsNotNone(game.target)
```

Figure 1: R1 and R2 Tests

These tests failed as expected because hangman module didn't exist.

```
→  assignment git:(main) ✗ python -m unittest test_hangman.py
E                                                                    Add to Chat ⌘L
======================================================================
ERROR: test_hangman (unittest.loader._FailedTest)
----------------------------------------------------------------------
ImportError: Failed to import test module: test_hangman
Traceback (most recent call last):
  File "/Users/regmicmahesh/.pyenv/versions/3.7.17/lib/python3.7/unittest/loader.py", line 154, in lo
adTestsFromName
    module = __import__(module_name)
  File "/Users/regmicmahesh/assignment/test_hangman.py", line 4, in <module>
    from hangman import HangmanGame, GameLevel, GameState
ModuleNotFoundError: No module named 'hangman'


----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (errors=1)
→  assignment git:(main) ✗
```

Figure 2: R1 and R2 Tests Failing Screenshot

Once I could confirm these tests are failing, red phase of the TDD is complete. Now, I implemented this behaviour in the file **hangman.py**.

```python
import random
import time
from enum import Enum
from typing import Set, List, Optional


class GameLevel(Enum):
    BASIC = "basic"
    INTERMEDIATE = "intermediate"

class HangmanGame:

    # Basic words dictionary
    BASIC_WORDS = [
        "PYTHON", "PROGRAMMING", "COMPUTER", "KEYBOARD", "MONITOR",
        "SOFTWARE", "HARDWARE", "INTERNET", "WEBSITE", "DATABASE",
        "FUNCTION", "VARIABLE", "BOOLEAN", "INTEGER", "STRING"
    ]

    # Intermediate phrases dictionary
    INTERMEDIATE_PHRASES = [
        "HELLO WORLD", "COMPUTER SCIENCE", "SOFTWARE DEVELOPMENT",
        "ARTIFICIAL INTELLIGENCE", "MACHINE LEARNING", "DATA STRUCTURE",
        "OBJECT ORIENTED", "VERSION CONTROL", "USER INTERFACE"
    ]

    def __init__(self, level: GameLevel):
        self.level = level
        if level == GameLevel.BASIC:
            self.target = random.choice(self.BASIC_WORDS)
        else:
            self.target = random.choice(self.INTERMEDIATE_PHRASES)
```

Figure 3: R1 and R2 Implementation

Now, I again re-ran the tests and verified that the tests are running successfully.

```
➜  assignment git:(main) ✗ python -m unittest test_hangman.py -v
test_game_initialization_basic (test_hangman.TestHangmanGame) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
➜  assignment git:(main) ✗
```

Figure 4: R1 and R2 Test Output

The cycle is complete for the first requirement and now we can move on to the next requirement.

2.2.2. Requirement R3

I wrote the following tests for the third requirement. I patched the target to **PYTHON** and **HELLO WORLD**. This is required because the word are fetched randomly and we are only checking the display functionality here.

```
def test_display_word_initial(self):
    """Test that display word shows underscores initially."""
    with patch.object(self.basic_game, 'target', 'PYTHON'):
        display = self.basic_game.get_display_word()
        self.assertEqual(display, '_ _ _ _ _ _')

def test_display_phrase_initial(self):
    """Test that display phrase shows underscores and spaces initially."""
    with patch.object(self.intermediate_game, 'target', 'HELLO WORLD'):
        display = self.intermediate_game.get_display_word()
        self.assertEqual(display, '_ _ _ _ _   _ _ _ _ _')
```

Figure 5: R3 Test

After that, I implemented the functionality to replace all the characters with underscore to get the tests passing. I had to add a new property to the game class called guessed_letters so that I can display those characters directly instead of underscore for already guessed letters.

```python
def get_display_word(self) -> str:

    if self.level == GameLevel.BASIC:
        display_chars = []
        for char in self.target:
            if char.upper() in self.guessed_letters:
                display_chars.append(char)
            else:
                display_chars.append('_')
        return ' '.join(display_chars)
    else:

        display_chars = []
        for char in self.target:
            if char == ' ':
                display_chars.append('  ')
            elif char.upper() in self.guessed_letters:
                display_chars.append(char + ' ')
            else:
                display_chars.append('_ ')
        return ''.join(display_chars).rstrip()
```

Figure 6: R3 Implementation

I refactored the code a little bit and enhanced visual formatting after the implementation in refactor phase for better readability. I followed the similar approach for all other requirements.

2.2.3. Requirement R4

For the time requirement, I wrote 3 tests to verify the timer functionality. The remaining time should be between 0 and 15 seconds. The timer should not be up before 15 seconds. I also mocked the timer_start property to expire it and verified that the is_time_up works correctly when the time has completed already.

```python
def test_timer_start(self):
    self.basic_game.start_timer()
    self.assertIsNotNone(self.basic_game.timer_start)
    remaining = self.basic_game.get_remaining_time()
    self.assertTrue(0 < remaining <= 15)

def test_timer_not_started(self):
    remaining = self.basic_game.get_remaining_time()
    self.assertIsNone(remaining)
    self.assertFalse(self.basic_game.is_time_up())

def test_timer_timeout(self):
    with patch.object(self.basic_game, 'timer_start', time.time() - 16):
        self.assertTrue(self.basic_game.is_time_up())
        self.assertEqual(self.basic_game.get_remaining_time(), 0)
```

Figure 7: R4 Test

In the green phase of R4, I implemented this functionality with minimal code. The time is first calculated and set to the timer_start property. It's lazily evaluated later on to verify it it's expired already.

```
def start_timer(self):
    self.timer_start = time.time()

def get_remaining_time(self) -> Optional[int]:
    if self.timer_start is None:
        return None

    elapsed = time.time() - self.timer_start
    remaining = self.timer_duration - elapsed
    return max(0, int(remaining + 0.5))  # Round up

def is_time_up(self) -> bool:
    if self.timer_start is None:
        return False

    elapsed = time.time() - self.timer_start
    return elapsed >= self.timer_duration
```

Figure 8: R4 Implementation

2.2.4. Requirement R5

I wrote a simple test by patching the target to HELLO and checking if the display has accurately displayed the two L characters and other are replaced with underscore.

```
def test_multiple_occurrences_revealed(self):
    with patch.object(self.basic_game, "target", "HELLO"):
        self.basic_game.make_guess("L")
        display = self.basic_game.get_display_word()
        self.assertEqual(display, "_ _ L L _")
```

Figure 9: R5 Test

I implemented this functionality in the make_guess method of the Hangman class. The make_guess will update the guessed letters with the newly typed letter.

```python
def make_guess(self, letter: str) -> bool:
    # Input validation
    if not letter or len(letter) != 1 or not letter.isalpha():
        raise ValueError("Guess must be a single letter")

    # Convert to uppercase for consistency
    letter = letter.upper()

    # Check if already guessed
    if letter in self.guessed_letters:
        return letter in self.target.upper()

    # Add to guessed letters
    self.guessed_letters.add(letter)

    # Check if letter is in target
    is_correct = letter in self.target.upper()

    if not is_correct:
        self.lives -= 1

    # Update game state
    self._update_game_state()

    return is_correct
```

Figure 10: R5 Implementation

Requirement R6-R8 were also implemented in similar way.

2.2.5. Final Test Output

After completing the implementation of the hangman game, I ran all the test suite and we had 21 tests in total. All of the tests were passing which validated all the requirements we had originally listed down.

```
→ assignment git:(main) ✗ python -m unittest test_hangman.py -v
test_game_level_enum (test_hangman.TestGameEnums)
Test GameLevel enum values. ... ok
test_game_state_enum (test_hangman.TestGameEnums)
Test GameState enum values. ... ok
test_case_insensitive_guessing (test_hangman.TestHangmanGame)
Test that guessing is case insensitive. ... ok
test_display_phrase_initial (test_hangman.TestHangmanGame)
Test that display phrase shows underscores and spaces initially. ... ok
test_display_word_initial (test_hangman.TestHangmanGame)
Test that display word shows underscores initially. ... ok
test_game_initialization_basic (test_hangman.TestHangmanGame) ... ok
test_game_initialization_intermediate (test_hangman.TestHangmanGame)
Test that intermediate game initializes correctly. ... ok
test_game_lost (test_hangman.TestHangmanGame)
Test game state changes to lost when lives reach zero. ... ok
test_game_won (test_hangman.TestHangmanGame)
Test game state changes to won when word is guessed. ... ok
test_get_guessed_letters (test_hangman.TestHangmanGame)
Test tracking of guessed letters. ... ok
test_get_target_answer (test_hangman.TestHangmanGame)
Test getting the target answer. ... ok
test_handle_timeout_reduces_lives (test_hangman.TestHangmanGame)
Test that handling timeout reduces lives. ... ok
test_invalid_guess_letter (test_hangman.TestHangmanGame)
Test making an invalid letter guess. ... ok
test_invalid_input_handling (test_hangman.TestHangmanGame)
Test handling of invalid input. ... ok
test_multiple_occurrences_revealed (test_hangman.TestHangmanGame)
Test that all occurrences of a letter are revealed. ... ok
test_repeated_guess_same_result (test_hangman.TestHangmanGame)
Test that repeated guesses return same result. ... ok
test_timeout_can_end_game (test_hangman.TestHangmanGame)
Test that timeout can end the game. ... ok
test_timer_not_started (test_hangman.TestHangmanGame) ... ok
test_timer_start (test_hangman.TestHangmanGame) ... ok
test_timer_timeout (test_hangman.TestHangmanGame) ... ok
test_valid_guess_letter (test_hangman.TestHangmanGame)
Test making a valid letter guess. ... ok

----------------------------------------------------------------------
Ran 21 tests in 0.001s

OK
→ assignment git:(main) ✗
```

Figure 11: Final Test Output

2.2.6. Linter Implementation

I used ***flake8***, ***black*** and ***pylint*** for linting the code after each implementation.

```
● → assignment git:(main) ✗ python3 -m flake8 hangman.py main.py  test_hangman.py
● → assignment git:(main) ✗ python3 -m black hangman.py main.py test_hangman.py --check
All done! ✨ 🍰 ✨
3 files would be left unchanged.
○ → assignment git:(main) ✗
```

Originally it had lot of linting errors because of missing comments but I resolved all of those issues by adding docstrings and comments to each and every module, method and class.

```
⊗ → assignment git:(main) pylint main.py
********** Module main
main.py:1:0: C0114: Missing module docstring (missing-module-docstring)
main.py:8:0: C0116: Missing function or method docstring (missing-function-docstring)
main.py:12:0: C0116: Missing function or method docstring (missing-function-docstring)
main.py:26:0: C0116: Missing function or method docstring (missing-function-docstring)
main.py:36:8: R1705: Unnecessary "elif" after "return", remove the leading "el" from "
elif" (no-else-return)
main.py:44:0: C0116: Missing function or method docstring (missing-function-docstring)
main.py:71:0: C0116: Missing function or method docstring (missing-function-docstring)
main.py:93:0: C0116: Missing function or method docstring (missing-function-docstring)
main.py:106:0: C0116: Missing function or method docstring (missing-function-docstring
)
main.py:109:8: R1705: Unnecessary "elif" after "return", remove the leading "el" from
"elif" (no-else-return)
```

There are no linting errors in the program and it satisfied the PEP-8 python standards fully.

```
Problems  Output  Debug Console  Ports  Terminal
● → assignment git:(main) ✗ pylint *.py

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

○ → assignment git:(main) ✗
```

3. Conclusion

3.1. What Went Well

In this unit testing task, I successfully applied red-green-refactor methodology throughout all the development phases which resulted in well tested and maintainable code.

- The functionality of the program is well covered with 21-unit tests, and it has 100% pass rate.
- All the project requirements are successfully implemented with full functionality.
- There are zero linting errors and the formatting is very consistent and up to the PEP8 standards.
- Multiple quick cycles improved the code quality a lot and provided clear understanding of the system.
- I successfully integrated version control and code quality tools.

3.2. Areas of Improvements

- The timer is lazily evaluated which do not provide a great UX to the user. It could be changed to a more interactive approach.
- The errors messages could be more specific and helpful for different types of invalid inputs.
- Some edge cases were discovered later in the program, and I had to do the work which did not fit in any of the requirement cycle.
- More details in the beginning and more clarity could have made the testing a lot easier.

3.3. How Improvements can be implemented

- For the enhanced timer display, we could use a thread-based approach for real time display. The timer can run on the background thread.
- The error messages can be better handled by creating a separate error handler class and more appropriate error messages.
- Detailed documentation should be developed before and after each approach. This should have corrected the cases where requirements were met but functionality not being correct multiple times.

4. Appendix

4.1. GitHub Repository

The repository link is given below.
https://github.com/regmicmahesh/tdd-hangman

It contains the complete source code of the application as well as the development history of the program.