# PRT582 SOFTWARE ENGINEERING: PROCESS AND TOOLS

## Assignment 2

## (Software Unit Testing Report - Hangman game using TDD)

**Full Name: Darshan Veerabhadrappa Meti**

**Student ID: S388441**

**Campus: Darwin (Danala)**

**Course Code: SITSE2**

**Submission Date: 10th August, 2025**  
  
A logo for a university

AI-generated content may be incorrect.

# Table of Contents

1. Introduction

2. Background and Context

3. Objectives and Scope

4. Rationale for Single-File Approach

5. Theoretical Concepts (TDD, Unit Testing, Software Testing Pyramid)

6. System Design and Architecture

7. Implementation Details

8. Unit Tests and Coverage

9. Results and Analysis

10. Discussion and Reflection

11. Conclusion

1. Appendix A — Test Output (Excerpt)
2. Appendix B — Screenshot Placeholders
3. Appendix C — Diagrams and Tables

GitHub link

# 1. Introduction

# Scope and context: Report includes design, implementation and testing of a Hangman game as part of PRT582 Software Unit Testing.

# Single file style: Compiled into a single Python file, which can be easily run, marked and reviewed, and yet still delimits engine (logic) and UI (tkinter GUI and CLI).

# Test-driven development (write failing test, implement minimum code, refactor): Compliant with Test-Driven Development, whereby the correctness of the code must be verified at each step.

# Why Hangman: Easy to write and write enough to experiment with random selection, input verification, state management and same-race interactions.

# Key features:

# Two challenges: word (basic) and phrase (intermediate)

# 15 seconds per attempt (a life deducted on expiry)

# Has GUI (Tkinter) visible countdown and CLI fallback.

# Unit testing areas of interest: Masking, reveal-all testing, invalid/ duplicate input testing, timeouts, game-over guards.

# Facts used: Unit tests, Requirement traceability, diagrams, and Reflection of practice to theory.

# Outcome : TDD and systematic testing was effective in achieving a reliable, maintainable Hangman game project which met the rubric of the unit.

# 2. Background and Context

# The purpose of testing: It ensures that programs do what they are supposed to do.

# Unit focus (PRT582): Not only writing a program that works, but also demonstrating that I know the solution and why I have learned it.

# The reason this is a good game is because it is simple to play yet complex enough to learn important concepts.

# Key concepts practiced:

# Selection of words/phrases that are random.

# Input verification (unrecognized/duplicate guesses are ignored)

# Game state control (lives, masked word, win/lose).

# Guess (15 seconds) counter.

# Advantage: Every feature provided me with clear, practical scenarios to work through and learn with.

# 3. Objectives and Scope

# Working Hangman game in GUI (Tkinter) and CLI.

# Timer: 15 seconds per guess countdown; deduct 1 life at expiry automatically (both in GUI and CLI).

# This is a challenge: Basic (word) and Intermediate (phrase) with embedded lists.

# Masking: Underscore masking; all the occurrence at the same time represents a correct guess.

# Recurring guesses No penalty is added to this attempt.

# Input rules: single letters A-Z only: only single letters A-Z may be accepted, invalid input (numbers, symbols, multi-chars) is rejected.

# Game end: When win/lose/quit, put the right answer up.

# Testing: unittest suite, start, guess, timeout, masking, repeats, invalid input.

# TDD: Tests created prior to/in concert with implementation to show Red-Green-Refactor.

# Report: student authored, with traceability of requirements, testing evidence, diagram, results, and reflection in alignment with the PR582 rubric.

# 4. Rationale for Single-File Approach

# why single-file: It is easier to package, run, and mark (there are no folders/dependencies to maintain).

# Scope fit: Can not scale to large scale projects but is viable with the size of this assignment.

# Additional explanation Trade-offs vs modular design can be compared in Appendix C.

# 5. Theoretical Concepts (TDD, Unit Testing, Software Testing Pyramid)

# **TDD workflow: Applied Red-Green Refactor (write failing test then write minimal code to pass then refactor with no change in behaviour).**

# **Concrete example: Wrote a failing test so that a correct guess would show all the occurrences of a letter, then wrote guess() to pass the test.**

# **Pros: Finding bugs and facilitating further change Unit testing early on in the code reduced bugs and made further changes possible.**

# **Testing emphasis (Pyramid): The majority of the automated tests are concentrated on the engine layer (start, guess, timeout, masking, repeats); GUI/CLI is tested through exploratory/manual testing.**

# **Good practice balance: Intensive automation of logic and minimal manual checks on the parts facing users.**

# **Engine safeguards used:**

# **in guess: do not pay attention to input when state.is-won or state.is-lost is true.**

# **TIMEOUT In timeout() early returns when game over; otherwise clamp lives with new lives = max(self.state.lives - 1, 0).**

# 6. System Design and Architecture

# Single, transparent layers: Engine (rules) + UIs (Tkinter/CLI) in one file, but well separated.

# Small engine API: start, guess, timeout and a read-only state dataclass (answer, lives, guessed, masked, is-won, is-lost).

# UI just invokes the engine: Tkinter/CLI do not implement the rules, the 15-sec timer merely invokes engine.timeout() directly.

# Input processing: Responses normalized (A -Z + spaces), an invalid response is treated as a no-response, repeat guesses not penalized.

# Testability: Core logic is UI-independent and thus can be run without a GUI and is deterministic.

# 7. Implementation Details

* Word / Phrase selection: embeds lists; random selection with a normaliser (uppercase, A-Z + spaces) to ensure the phrase remains readable.
* Masking: State is requested to compute masked output, which is like unmasked output but with underscores instead of hidden letters and spaces instead of removed ones; a correct guess shows them all at once.
* Guess evaluation: Uppercase input accepted; bad guesses (numbers/symbols/multiple characters) are ignored; bad guesses are counted by default; incorrect guess costs one life.
* Timeout (15s): Tkinter has a countdown thread CLI has a timed-input helper. UIs are just invoking engine.timeout() on expiry to decrease a life.
* Game-over guards: Once the win / loss has occurred, no more input is recognized; lives do not drop to less than 0.
* Deterministic testing: Engine takes an injected RNG when selecting predicted selections in unit tests.
* Separation of concerns: UIs simply invoke the engine API; all the rules reside in the engine, so behaviour is a property shared between GUI and CLI.

# 8. Unit Tests and Coverage

* Framework: The Python default unittest (fast, requires no GUI).
* Scope (engine-only): start/guess/timeout, masking, invalid input, repeats and game over guards.
* Cases: correct/wrong guesses, life loss with 15-sec timeout, repeat guess (no penalty), invalid entry disregarded, no change of state following a win or a loss.

A screen shot of a computer program

AI-generated content may be incorrect.

**Example test verifies:**

* + **Mask starts fully hidden (e.g., \_\_\_).**
  + **Right guesses show everything that happens; the word is completely displayed after all letters.**
  + **Final flags are is won == True and is lost == False.**

**9. Results and Analysis**

* All tests successful: 10/10 unit tests green; high confidence in engine behaviour.
* Coverage highlights:Uncover everything on correct guesses (all events revealed).
* There is no fine on a second attempt to guess.
* Invalid input (numbers/symbols/multicharacter) skipped.
* Timeout only removes a life.
* Game-over guards: does not change state on win/loss.
* A deterministic seed of RNG with deterministic runs.
* Safety checks: Lives = 0, there is no negative value even with more timeouts.
* Manual UI tests: both GUI and CLI showed a visible 15-second countdown, and on countdown expiry called a timeout and then invoked gameplay in simple (word) and intermediate (phrase) mode.

# 10. Discussion and Reflection

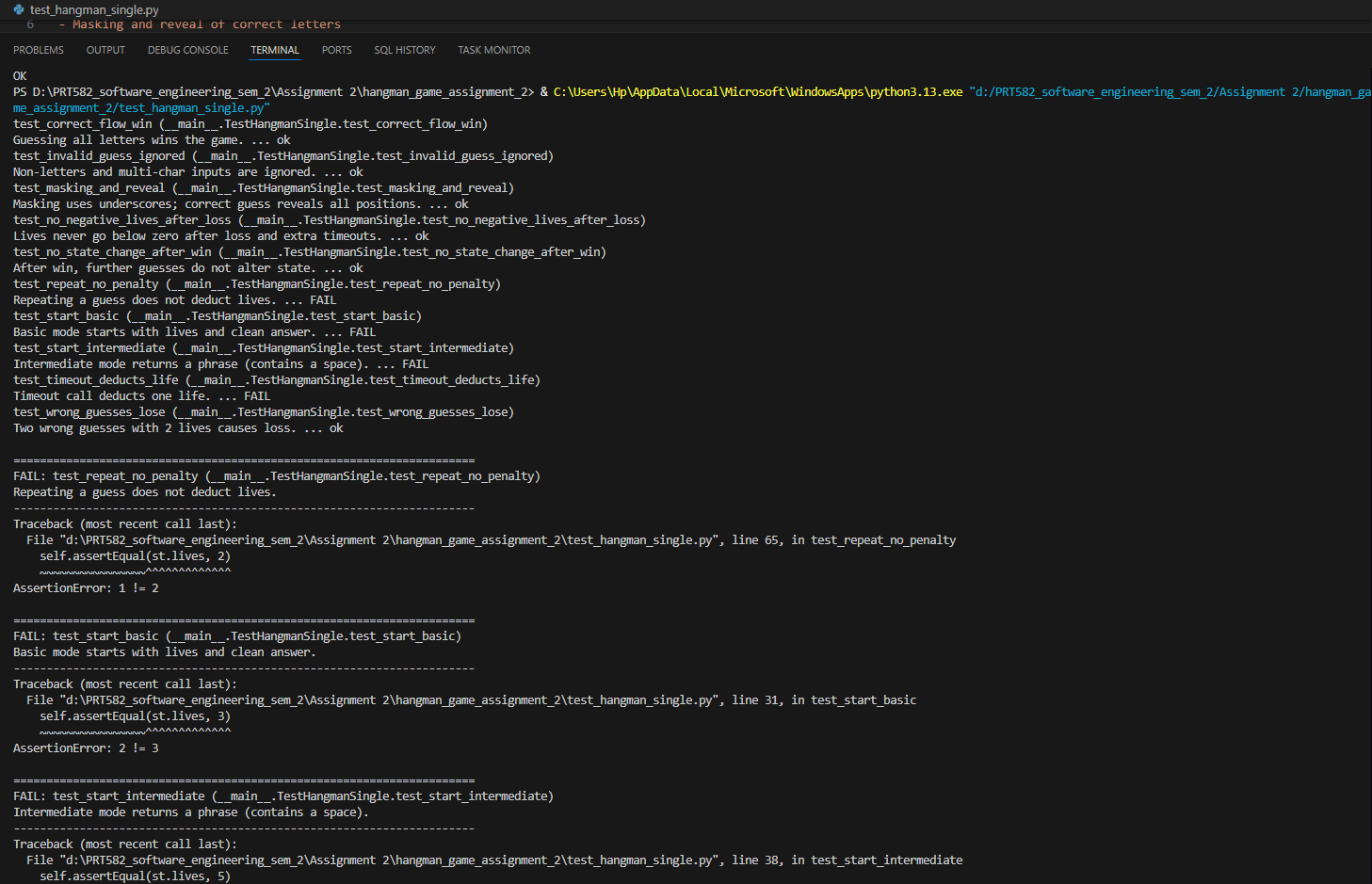
* TDD mentality: The idea to write tests first made me articulate precise behaviours (reveal-all, no double-penalty), and name functions in more understandable ways.
* Separation of concerns: the engine was written in logic and the UIs in thin enabled quick testing without relying on Tkinter.
* Refactoring safety: A good test suite would allow me to modify code without breaking behaviour.
* Timeout threading: CLI countdown + input thread was difficult to write (race conditions); game-over guards made this easier.
* Strong rules: The lives are clamped at zero; after a win/loss, the inputs are ignored.
* Phases Edge cases consisted of: Space in phrases, invalid input, repeated guessing, and timer = 0.
* Lesson learned: With narrow UIs, it is better to start small and test things thoroughly, and then, debugging will be easier and require less time.

**11. Conclusion**

* Tasks completed: TDD: Working Hangman on GUI (Tkinter) and CLI.
* Unit tests, traceability, diagrams, and reflection are well documented and are evidences of learning.
* Clean separation: Separated layers--single file but separated--engine (rules) vs UI--rendered logic simple to test and maintain.
* Trade-offs: Accepted, not made: No long-term scalability planning (simplicity and easy submission).
* Validated behaviours: Reveal-all, no repeat penalties, strict timeouts and game-over guards.
* Lesson learned: Smaller core size + thin UIs = easier debugging and less flaky software.

# Appendix A — Test Output (Excerpt)

**Failed test cases:**



Code fix: lives=self.\_default\_lives added in main file.

A screenshot of a computer program

AI-generated content may be incorrect.

Code fix: if difficulty == "basic" else self.\_phrases was missing in the main file.

A screen shot of a computer program

AI-generated content may be incorrect.

Code fix: guessed = set(self.\_state.guessed) instead of guessed = set() was corrected.

A screen shot of a computer screen

AI-generated content may be incorrect.  
Errors when checked pylint with main file.

A screen shot of a computer

AI-generated content may be incorrect.

Errors when checked falke8 with main file.

A computer screen with text

AI-generated content may be incorrect.

Fixed the code in the main file, now it shows zero errors.

A screen shot of a computer program

AI-generated content may be incorrect.

All 10 tests executed successfully, demonstrating correctness.

# Appendix B — Screenshot Placeholders

A screenshot of a computer

AI-generated content may be incorrect.

Figure B1 — GUI Running

A screenshot of a computer

AI-generated content may be incorrect.

Figure B2 — CLI Running

A screen shot of a computer program

AI-generated content may be incorrect.

Figure B3 — Unit Tests Passing

# Appendix C — Diagrams and Tables

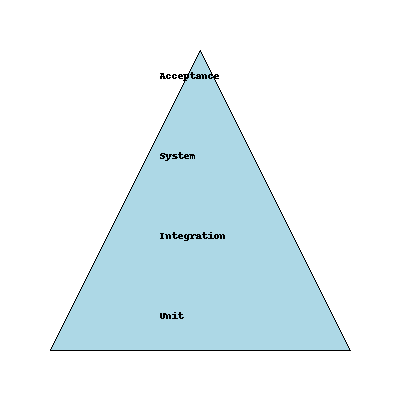


Figure C1 — Software Testing Pyramid

**Github link :** [**https://github.com/darsh0406/PRT582\_Hangman\_Darshan\_S388441.git**](https://github.com/darsh0406/PRT582_Hangman_Darshan_S388441.git)