**PRT582 - SOFTWARE ENGINEERING: PROCESS AND TOOLS**

**Name – Krijal Khadka**

**Student ID – S385427**

**Danala campus,CDU**

**Introduction**  
The project delivers a console-based Hangman game that fulfills the brief: specific two levels of difficulty (single word for Basic and multi-word phrase for Intermediate), a visible 15-second countdown per guess (timing out deducts a life), and the following behaviours: correct guesses reveal letters, wrong guesses deduct lives, detailed closing states (win, loss, or quit). A very simple hint system lets the player reveal one random, unrevealed letter at the expense of a life, along with a notional useable hint (counts of letters, vowels, and words) to improve usability. Word answers are returned randomly at runtime from a lightweight dependency to have no static word list. The core rules were first implemented using Test-Driven Development (TDD) with automated unit tests against them; a thin command-line interface (CLI) provides timing and user interaction.

**Technology Selection and Justification**

Python was my language of choice as it allows fast iteration, displays readable code that suits a TDD workflow well, and requires little to no additional effort to get up and running. The standard unittest framework is built-in, requires no additional installation if using a code editor like VS Code, and makes it easy to run tests again as your feature evolves. To keep things satisfying the need for randomly generated, real words, while not adding additional work to maintain a local dictionary, the use of the wonderwords library supports single-word and short phrase generation at runtime. The architecture developed for the project separates a pure game engine (logic only) from the command-line interface (CLI) (I/O and timer) to provide predictable, deterministic tests, and to allow changing the user interface (UI) with little to no effort.

**Process (TDD and Automation)**

I followed my red-green-refactor loop. I wrote a failing test for a small rule, wrote the minimal code to pass, refactored while maintaining green tests. The tests focused on masking using underscores (spaces/punctuation visible), the difference between correct and wrong guesses, guessing a letter multiple times, time-out penalties, invalid input, winning/losing states, and hint behaviour (reveal exactly one new letter, cost a life, do nothing with everything has been revealed). Once the engine was stabilised, I added the CLI that selects the level, prints out the masked board and state, applies the 15 second timer (Windows uses msvcrt for a live countdown and macOS/Linux uses select), applies time-out penalties, applies the h (hint) and q (quit) commands. I validated it with unit tests (all passing) and played for screenshots.

**Requirements Mapping**

**Two levels (Basic & Intermediate)**

Player chooses **1** for a random single word or **2** for a random multi-word phrase generated at runtime.

# hangman\_cli.py

def choose\_level() -> str:

while True:

print("Choose level: [1] Basic (single word) [2] Intermediate (phrase)")

choice = input("> ").strip()

if choice in {"1","2"}: return choice

print("Invalid choice. Enter 1 or 2.\n")

def choose\_answer(level: str) -> str:

return random\_word() if level == "1" else random\_phrase()

# words\_loader.py

from wonderwords import RandomWord

import random

\_rw = RandomWord()

def random\_word() -> str:

while True:

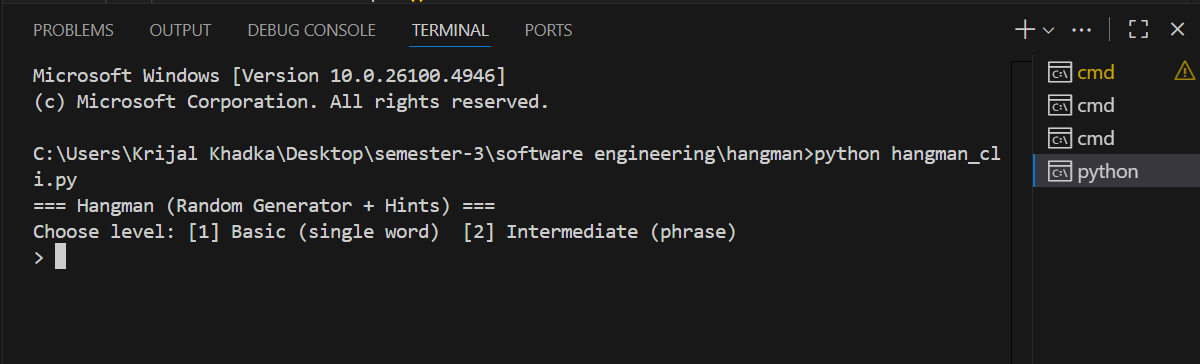
w = \_rw.word(word\_min\_length=3, word\_max\_length=10)

if w and w.isalpha(): return w.lower()

def random\_phrase() -> str:

n = random.randint(2, 3)

return " ".join(random\_word() for \_ in range(n))



**Valid, randomly generated words/phrases**

Answers come from the wonderwords dictionary at runtime so each game uses real, varied words without any hard-coded list.

# words\_loader.py (shown above) uses RandomWord() every time the game starts

A screenshot of a computer program

AI-generated content may be incorrect.

**Underscores for hidden letters; spaces/punctuation shown**

The masked board shows \_ for unknown letters while keeping spaces and punctuation visible for readability.

# hangman\_engine.py

MASK\_CHAR = "\_"

@property

def masked(self) -> str:

out: List[str] = []

for ch in self.answer:

if ch == " ":

out.append(" ")

elif ch.isalpha():

out.append(ch if ch in self.guessed else MASK\_CHAR)

else:

out.append(ch) # punctuation visible

return "".join(out)

A screenshot of a computer program

AI-generated content may be incorrect.

**15-second timer per guess; timeout deducts one life**

If the player doesn’t enter input within 15 seconds, the game prints a timeout message and subtracts one life.

# hangman\_cli.py

user = timed\_input(f"Guess a letter within {TIME\_LIMIT}s (h/q): ", TIME\_LIMIT)

if user is None or user.strip() == "":

print("⏰ Time's up! -1 life.")

engine.timeout\_penalty()

continue

# hangman\_engine.py

def timeout\_penalty(self):

if self.is\_won() or self.is\_lost(): return

self.timeouts += 1

self.lives -= 1

A screenshot of a computer

AI-generated content may be incorrect.

**Correct guess reveals all matching positions**

A correct letter adds to guessed and the mask instantly reveals that letter everywhere it appears.

# hangman\_engine.py

def guess(self, ch: str) -> bool:

if self.is\_won() or self.is\_lost(): return False

if not ch or len(ch) != 1: return False

ch = ch.lower()

if not ch.isalpha(): return False

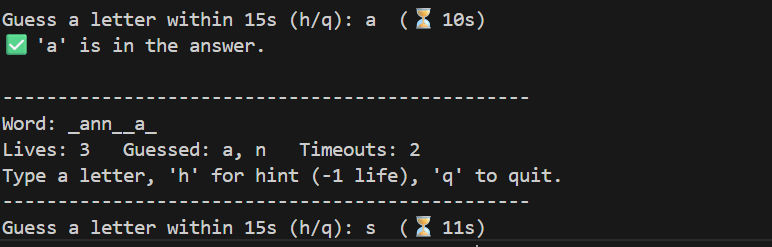
if ch in self.guessed: return ch in self.answer # no penalty on repeat

self.guessed.add(ch)

if ch in self.answer: return True

self.lives -= 1

return False



**Wrong guess deducts one life**

An incorrect alphabetical guess immediately reduces the player’s lives by one.

# hangman\_engine.py (see above)

if ch not in self.answer:

self.lives -= 1

return False

A screenshot of a computer program

AI-generated content may be incorrect.

**Must solve before lives reach zero (win/lose)**

The player wins when all letters are revealed and loses if lives hit zero before that.

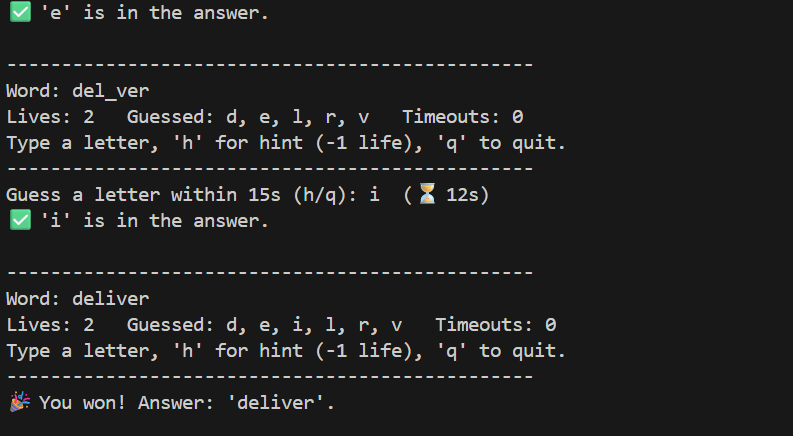
# hangman\_engine.py

def is\_won(self) -> bool:

return all((not ch.isalpha()) or (ch in self.guessed) for ch in self.answer)

def is\_lost(self) -> bool:

return self.lives <= 0 and not self.is\_won()



**Continue until quit or life = 0; always show the answer**

The main loop runs until win/lose or the player presses q, and the final answer is printed in all cases.

# hangman\_cli.py

if user in {"q", "quit"}:

print("You quit the game.")

break

# after loop

if engine.is\_won():

print(f"🎉 You won! Answer: '{engine.answer}'.")

elif engine.is\_lost():

print(f"💀 Game over. Answer: '{engine.answer}'.")

else:

print(f"👋 Goodbye! Answer was: '{engine.answer}'.")

A screenshot of a computer program

AI-generated content may be incorrect.

**Hint feature (extra)**

Pressing h reveals one random unrevealed letter and costs one life only when a new letter is actually revealed.

# hangman\_engine.py

def reveal\_hint\_letter(self) -> str | None:

remaining = sorted({ch for ch in self.answer if ch.isalpha()} - self.guessed)

if not remaining: return None

letter = random.choice(remaining)

self.guessed.add(letter)

self.lives -= 1

return letter

# hangman\_cli.py

if user in {"h", "hint"}:

letter = engine.reveal\_hint\_letter()

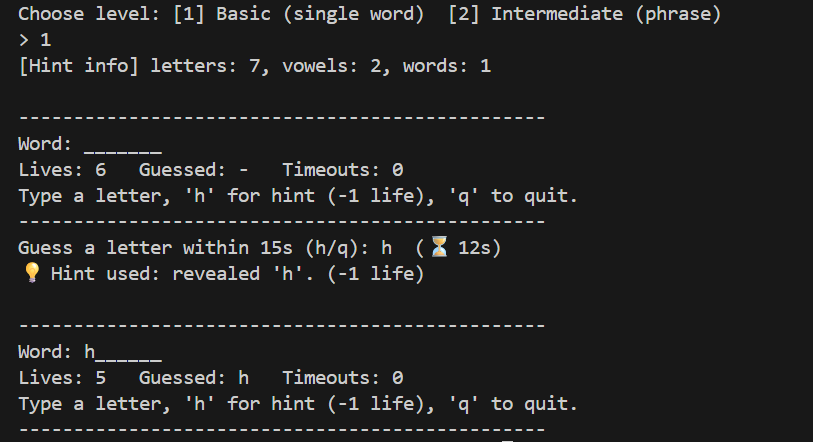
if letter is None:

print("ℹ️ No hint available—all letters already revealed.")

else:

print(f"💡 Hint used: revealed '{letter}'. (-1 life)")

continue



**Automated tests (TDD evidence)**

A unittest suite verifies masking, guesses, repeats, timeouts, hints, and win/lose logic so the engine stays correct as you change code.

# test\_hangman.py (examples)

def test\_wrong\_guess\_costs\_life(self):

eng = HangmanEngine("test", lives=2)

self.assertFalse(eng.guess('x'))

self.assertEqual(eng.lives, 1)

def test\_hint\_reveals\_letter\_and\_costs\_life(self):

eng = HangmanEngine("test", lives=3)

before = eng.lives

revealed = eng.reveal\_hint\_letter()

self.assertIsNotNone(revealed)

self.assertEqual(eng.lives, before - 1)

A screen shot of a computer

AI-generated content may be incorrect.

**Conclusion**

TDD was an invaluable resource in keeping the engine small, deterministic, and relatively easy to reason about, while the CLI elements were able to deal with platform -specific timing issues without infecting the tests. With the I/O logic separated from the rest of the logic it kept flaky behaviour at bay and also gave us the ability to refactor with confidence. Using wonderwords (a wonderful library via pypi.org) satisfied conditions under “random, real words” and prevented us from curating a local asset list. The outcome is a simple game, driven by tests, that is on brief with a straightforward structure to make changes going forward.

**GitHub repository:** [**khadkakrijal/hangman: Hangman TDD with timer & hints**](https://github.com/khadkakrijal/hangman)