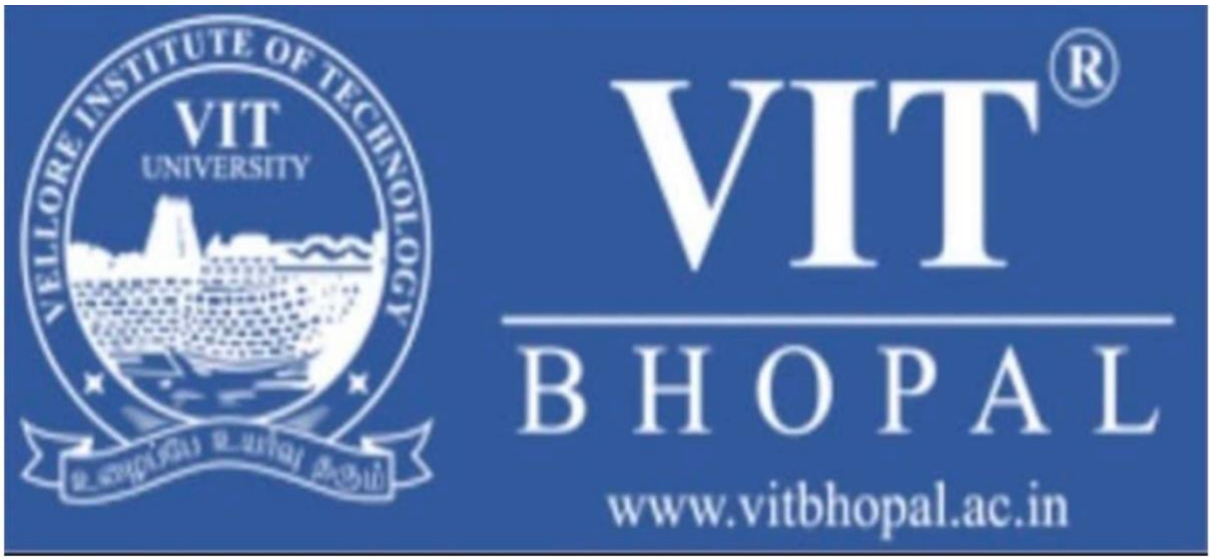

PROJECT TITLE - CLASSIC HANGMAN GAME

SUBMITTED BY – NIVESH MALIK

REG. NO – 25BCE11066

PROGRAMME – B.TECH

COURCE CODE – CSE1021

FACULTY NAME – M K JAYANTHI

INSTITUTION – VIT BHOPAL

1 Introduction

This report details the implementation and design of a classic text-based Hangman game developed in Python. The game selects a random word from a predefined list and challenges the user to guess the word letter by letter, with a limited number of incorrect attempts (lives). The game utilizes external modules (hangman_words for the word bank and hangman_art for visual representation of the game stages).

2 Functional Requirements

1. Python should be installed on system.
2. All the .py files should be inside same folder

3 Problem Statement

The goal is to develop a functional, interactive console application that accurately simulates the traditional Hangman game logic. This includes managing player lives, tracking correctly and incorrectly guessed letters, updating the displayed word progress, and providing visual feedback (the hangman stages) corresponding to the remaining lives.

4 System Architecture

The Hangman game uses a simple **Modular Architecture** with three primary components:

1. **MainLogic(<script>):** Contains the game loop, state management (lives, game_over), input handling, and the core detection/comparison logic.
2. **Data Module (hangman_words.py):** Stores the list of possible words.
3. **Visual Module (hangman_art.py):** Stores the visual assets (logo and stages array) that represent the state of the game.

The flow is linear, driven by the main loop: Initialization → Loop → Check Win/Loss → Output.

5 Design Diagrams

5.1 Use Case Diagram

- **Actor:** Player - **Use Cases:**
- Start Game
- Guess Letter
- View Game State
- Win Game
- Lose Game

Conceptual Diagram Description:

5.2 Workflow Diagram

1. **Start:** Load modules and initialize variables (lives=6, chosen_word, placeholder).
2. **Loop Start:** Check if game_over is True (If Yes → End).
3. **Input:** Prompt user for a letter guess.
4. **Process Guess:** Check if the guess is in correct_letters (Handle duplicate).
5. **Update Display:** Iterate through chosen_word to construct the display string.
6. **Check Miss:** Is the guess in chosen_word? (If No →lives decrements).
7. **Check Loss:** Has lives reached 0? (If Yes →game_over=True, declare loss).
8. **Check Win:** Is '_' not in display? (If Yes →game_over=True, declare win).
9. **Output:** Print current Hangman stage (stages[lives]).
10. **Loop Back:** Return to Loop Start.

Conceptual Diagram Description:

5.3 Sequence Diagram

- **Actors:** Player, MainScript, WordModule, ArtModule
- **Sequence:**
    1. Player initiates script.
    2. MainScript imports from WordModule (gets word_list).
    3. MainScript selects chosen_word.
    4. MainScript imports from ArtModule (gets logo, stages).
    5. MainScript prints logo and initial placeholder.
    6. **Loop:**
        (a) MainScript prompts Player for guess.
        (b) Player provides guess.
        (c) MainScript processes guess (correctly or incorrectly).

(d) MainScript updates lives and correct_letters.
(e) MainScript prints updated word display and stage from ArtModule.

7. **End Loop (Win/Loss):** MainScript prints final result.

Conceptual Diagram Description:

5.4 Class/Component Diagram

Table 3: System Components

Component	Description
Main Script	Central logic, manages game state (lives, game_over), I/O, and win/loss checks.
random Module	Used to select the initial chosen_word.
hangman_words	External module providing the word_list data (String Array).
hangman_art	External module providing the visual assets (logo and stages array).
Variables (lives,	Internal state data for the current game instance. **chosen_word, display)**

Conceptual Diagram Description:

5.5 ER Diagram (if storage used)

N/A. No persistent storage or database is utilized in this console game. The state is only maintained in memory during execution.

6 Design Decisions & Rationale

Table 4: Design Decisions

Decision	Rationale
External Modules	Separation of concerns: keeping the core logic clean by isolating word data (hangman_words) and visual presentation (hangman_art). This makes the game easier to modify (e.g., adding more words or changing the art).
Lives and Stages Array	The use of an integer lives variable (starting at 6) directly maps to the index of the stages list, allowing for simple, efficient retrieval of the correct hangman image without complex conditional logic.
correct_letters Array	Instead of checking the full word against the current guess every time, the correct_letters array efficiently tracks all successful guesses, ensuring that letters are only

revealed if they match the current guess OR are already known.

Lowercase Input input("Guess a letter: ").lower() ensures that the comparison logic works regardless of whether the user enters an uppercase or lowercase letter, making the game more user-friendly.

7 Implementation Details

The core game logic resides within the while not game_over: loop.

- **Initialization:** The code initializes the word display with underscores:

```
placeholder = "" # ...
for position in range(word_length): placeholder += "_"
```

- **Guess Handling:** The logic for updating the display handles two conditions:
  1. if letter == guess: (The current guess is found).
  2. elif letter in correct_letters: (A previous, successful guess is found).
- **Win/Loss Condition Checks (End of Loop):**
- Loss check: if guess not in chosen_word: triggers the life loss and checks if lives == 0:.
- Win check: if "_" not in display: confirms the word is complete.

8 Screenshots / Results

As this is a console application, screenshots cannot be provided, but a typical successful game state sequence would look like this:

Example Game Flow Segment (Conceptual):

1. Word to guess: _____e
2. Guess a letter: t
3. You guessed t, that's not in the word. You lose a life.
4. [Art: Stage 5]
5. Guess a letter: h
6. Word to guess: h__ _e
7. [Art: Stage 5]
8. Guess a letter: r
9. Word to guess: h__re
10. [Art: Stage 5]

9 Testing Approach

The game requires manual testing focusing on the core functional requirements:

1. **Initialization Test:** Verify lives starts at 6, the logo is shown, and the display is all underscores.
2. **CorrectGuessTest(FR4):** Guess a letter known to be in the chosen word and confirm the display updates correctly and lives remains unchanged.
3. **Incorrect Guess Test (FR5):** Guess a letter known *not* to be in the word and confirm lives decreases by 1 and the Hangman stage updates visually.
4. **Win Test (FR7):** Complete the word and ensure the game loop terminates with the "*YOU WIN*" message.
5. **Loss Test (FR8):** Make 6 incorrect guesses and ensure the game loop terminates, the correct word is revealed, and the "*YOU LOSE*" message is displayed alongside the final Hangman stage (stages[0]).
6. **Duplicate Guess Test (FR6):** Attempt to guess a letter that was already successfully guessed and confirm the duplicate message is printed without affecting lives.

10 Challenges Faced

1. **HandlingDuplicateCorrectGuesses:** A challenge in Hangman is ensuring the guess logic distinguishes between a letter being present in the word and a letter already having been guessed. The current implementation slightly simplifies this by relying on the loop that constructs the display string.
2. **Modular Imports:** Relying on external, simple Python files (.py) for assets, which is a design decision (NFR2), requires those files to be correctly placed in the execution environment.

11 Learnings & Key Takeaways

- **Iterative String Construction:** Building the display string character by character within the loop is an effective way to update the word's status based on both the current guess and all previous correct guesses.
- **State Machine Mapping:** The direct relationship between the lives variable (0 to 6) and the stages array index is a simple, effective example of mapping a game state variable to a visual output array.
- **Code Clarity:** The use of clear boolean flags like game_over simplifies loop control and makes the exit conditions explicit.

12 Future Enhancements

1. **Input Validation:** Add robust checks to ensure the user only inputs a single alphabetical character and prevent inputting previously *incorrect* letters (currently only prevents previously *correct* ones).
2. **Tracking Guessed Letters:** Maintain a separate set for *all* guessed letters (correct and incorrect) and display this set to the user to aid their guessing process.
3. **Word Hints:** Implement a difficulty system where lives/stages change, or provide a hint feature if the user loses a few lives.
4. **Graphical Interface:** Upgrade the game from the console to a simple GUI (e.g., using Tkinter or PyQt) to better showcase the visual Hangman stages.

13 References

1. **Python Standard Library:** Used for core data types and the `random` module.
2. **External Modules (Assumed):** `hangman_words.py` (for word list) and `hangman_art.py` (for ASCII art stages and logo).