Hangman Word Guessing Game Development Report

1. Introduction

1.1 Project Objectives

This project aims to develop a complete Hangman word guessing game program with the following core functionalities:

- Two game levels: Basic mode (words) and Intermediate mode (phrases).
- 15-second countdown mechanism.
- Life system (6 lives).
- Graphical user interface.
- Comprehensive input validation and error handling.

1.2 Technology Selection Rationale

Programming Language: Python

- **Simple and Readable**: Python's clear syntax makes it suitable for rapid development and maintenance.
- **Rich GUI Libraries**: Built-in tkinter library requires no additional installation.
- **Cross-platform Compatibility**: Supports Windows, macOS, and Linux.
- **Strong Testing Support**: Excellent testing frameworks like pytest.

Automated Unit Testing Tool: pytest

- **Simple and Easy to Use**: Concise syntax with a low learning curve.
- **Powerful Features**: Supports parameterized tests, fixtures, mocks, and other advanced features.
- **Rich Plugin Ecosystem**: Plugins like pytest-cov for code coverage analysis.
- **Detailed Reports**: Provides precise test results and failure information.

2. Process

2.1 Project Initialisation and Environment Setup

First, create a project structure and install necessary dependencies:

```
# Testing dependencies
pytest>=8.0.0
pytest-cov>=4.0.0

# Code quality tools
pylint>=3.0.0
```

2.2 Test-Driven Development (TDD) Implementation

2.2.1 Creating Word and Phrase Dictionary

Requirement: Provide valid English words and phrases as game content.

TDD Process:

1. First, create a `words.py` file, defining basic words and intermediate phrase lists.
2. Include 50 basic words and 50 intermediate phrases.
3. All vocabulary is valid English words/phrases.

```
"""
Hangman Game Word and Phrase Dictionary
Contains vocabulary for basic mode (words) and intermediate mode (phrases)
"""

# Basic Mode - Single Words
BASIC_WORDS = [
    "python", "computer", "programming", "algorithm", "function",
    "variable", "string", "integer", "boolean", "dictionary",
    "list", "tuple", "loop", "condition", "class", "object",
    "method", "inheritance", "polymorphism", "encapsulation",
    "abstraction", "recursion", "iteration", "debugging", "testing",
    "framework", "library", "module", "package", "import",
    "exception", "error", "syntax", "semantic", "compiler",
    "interpreter", "runtime", "memory", "stack", "queue",
    "tree", "graph", "hash", "sort", "search", "binary",
    "linear", "recursive", "iterative", "optimization",
    "hello", "world", "game", "player", "guess", "letter",
    "word", "phrase", "level", "difficulty", "challenge",
    "victory", "defeat", "success", "failure", "attempt"
]

# Intermediate Mode - Phrases
INTERMEDIATE_PHRASES = [
    "artificial intelligence",
    "machine learning",
    "data science",
    "web development",
    "mobile application",
    "user interface",
    "user experience",
    "database management",
    "software engineering",
    "version control",
    "agile methodology",
    "test driven development",
    "object oriented programming",
    "functional programming",
    "cloud computing",
    "cyber security",
    "network administration",
    "system architecture",
    "application programming interface",
    "representational state transfer",
    "structured query language",
    "hypertext markup language",
    "cascading style sheets",
    "javascript object notation",
    "extensible markup language",
    "simple object access protocol",
```

```
    "domain name system",
    "transmission control protocol",
    "internet protocol",
    "file transfer protocol",
    "hypertext transfer protocol",
    "secure sockets layer",
    "transport layer security",
    "public key infrastructure",
    "virtual private network",
    "local area network",
    "wide area network",
    "wireless fidelity",
    "bluetooth technology",
    "near field communication",
    "hello world",
    "good morning",
    "thank you",
    "you are welcome",
    "have a nice day",
    "see you later",
    "take care",
    "best wishes",
    "happy birthday",
    "merry christmas"
]
```

2.2.2 Game Core Logic Development

Requirement: Implement game rules, guess processing, and win/lose determination.

1. Write test cases first (`test_hangman.py`).
2. Implement the `HangmanGame` class to pass the test.
3. Refactor code while maintaining test coverage.

Key Test Cases:

```python
    def test_game_initialization_basic_level(self):
        """Test basic level game initialization"""
        game = HangmanGame(GameLevel.BASIC)
        assert game.level == GameLevel.BASIC
        assert game.lives == 6
        assert game.max_lives == 6
        assert game.word is not None
        assert len(game.word) > 0
        assert game.guessed_letters == set()
        assert game.display_word.count('_') > 0

    def test_correct_letter_guess(self):
        """Test correct letter guess"""
        with patch('hangman_game.random.choice', return_value='python'):
            game = HangmanGame(GameLevel.BASIC)
            result = game.make_guess('p')
            assert result['success'] is True
            assert result['message'] == 'Correct guess!'
            assert 'p' in game.display_word
            assert game.lives == game.max_lives  # Lives should not decrease
```

Implementation Results:

- Created `HangmanGame` class with complete game logic.
- Implemented `GameLevel` enum to define game levels.
- Supports both word and phrase modes.
- Comprehensive input validation and error handling.

2.2.3 Graphical User Interface Development

Requirement: Create a modern GUI interface.

TDD Process:

1. First, create `the HangmanGUI` class.
2. Implement a clean interface design.
3. Integrate game logic.

Interface Features:

- Clear visual hierarchy.
- Real-time game state display.
- User-friendly input validation.
- Keyboard support (Enter key confirmation).

2.2.4 Test Coverage and Verification

Test Statistics:

- Total test cases: 22.
- Test pass rate: 100%.
- Coverage scope: Game logic, integration tests, edge cases.

Test Run Results:

```
PS E:\Python\Hangman> python -m pytest test_hangman.py -v
============================================== test session starts ==============================================
platform win32 -- Python 3.13.7, pytest-8.4.2, pluggy-1.6.0 -- C:\Users\Lonny\AppData\Local\Microsoft\WindowsApps\PythonSoft
wareFoundation.Python.3.13_qbz5n2kfra8p0\python.exe
cachedir: .pytest_cache
rootdir: E:\Python\Hangman
plugins: cov-6.2.1
collected 23 items

test_hangman.py::TestHangmanGame::test_game_initialization_basic_level PASSED                                [  4%]
test_hangman.py::TestHangmanGame::test_game_initialization_intermediate_level PASSED                         [  8%]
test_hangman.py::TestHangmanGame::test_display_word_format PASSED                                            [ 13%]
test_hangman.py::TestHangmanGame::test_display_phrase_format PASSED                                          [ 17%]
test_hangman.py::TestHangmanGame::test_correct_letter_guess PASSED                                           [ 21%]
test_hangman.py::TestHangmanGame::test_incorrect_letter_guess PASSED                                         [ 26%]
test_hangman.py::TestHangmanGame::test_duplicate_letter_guess PASSED                                         [ 30%]
test_hangman.py::TestHangmanGame::test_game_won_condition PASSED                                             [ 34%]
test_hangman.py::TestHangmanGame::test_game_lost_condition PASSED                                            [ 39%]
test_hangman.py::TestHangmanGame::test_timer_functionality PASSED                                            [ 43%]
test_hangman.py::TestHangmanGame::test_phrase_guessing PASSED                                                [ 47%]
test_hangman.py::TestHangmanGame::test_invalid_input_handling PASSED                                         [ 52%]
test_hangman.py::TestHangmanGame::test_get_remaining_time PASSED                                             [ 56%]
test_hangman.py::TestHangmanGame::test_get_wrong_guesses_count PASSED                                        [ 60%]
test_hangman.py::TestGameLevel::test_game_level_values PASSED                                                [ 65%]
test_hangman.py::TestGameLevel::test_game_level_enumeration PASSED                                           [ 69%]
test_hangman.py::TestGameIntegration::test_complete_basic_game_flow PASSED                                   [ 73%]
test_hangman.py::TestGameIntegration::test_complete_intermediate_game_flow PASSED                            [ 78%]
test_hangman.py::TestGameIntegration::test_game_lost_flow PASSED                                             [ 82%]
test_hangman.py::TestEdgeCases::test_empty_word_handling PASSED                                              [ 86%]
test_hangman.py::TestEdgeCases::test_special_characters_in_input PASSED                                      [ 91%]
test_hangman.py::TestEdgeCases::test_case_insensitive_guessing PASSED                                        [ 95%]
test_hangman.py::TestEdgeCases::test_whitespace_handling PASSED                                              [100%]

============================================== 23 passed in 0.20s ===============================================
```

2.3 Functional Requirements Implementation Verification

2.3.1 Game Level Implementation

- **Basic Mode**: Random word generation.
- **Intermediate Mode**: Random phrase generation.
- Implemented level selection interface.

2.3.2 Dictionary Validation

- Created basic dictionary with 50 words.
- Created an intermediate dictionary with 50 phrases.
- All vocabulary is valid English words/phrases.

2.3.3 Underscore Display

- Word display: `_ _ _ _ _ _`
- Phrase display: `_ _ _ _ _ _ _ _ _ _ _`
- Correct guesses reveal letters.

2.3.4 Life System

- Initial six lives.
- Wrong guesses deduct one life.
- Real-time life display.

2.3.5 Win/Lose Conditions

- Win: Guess all letters.
- Lose: Lives exhausted.
- Display the correct answer when the game ends.

2.3.6 Game Loop

- Support continuous gameplay.
- Ask to play again after the game ends.
- Can exit the game anytime.

2.4. Project Optimisation and Cleanup

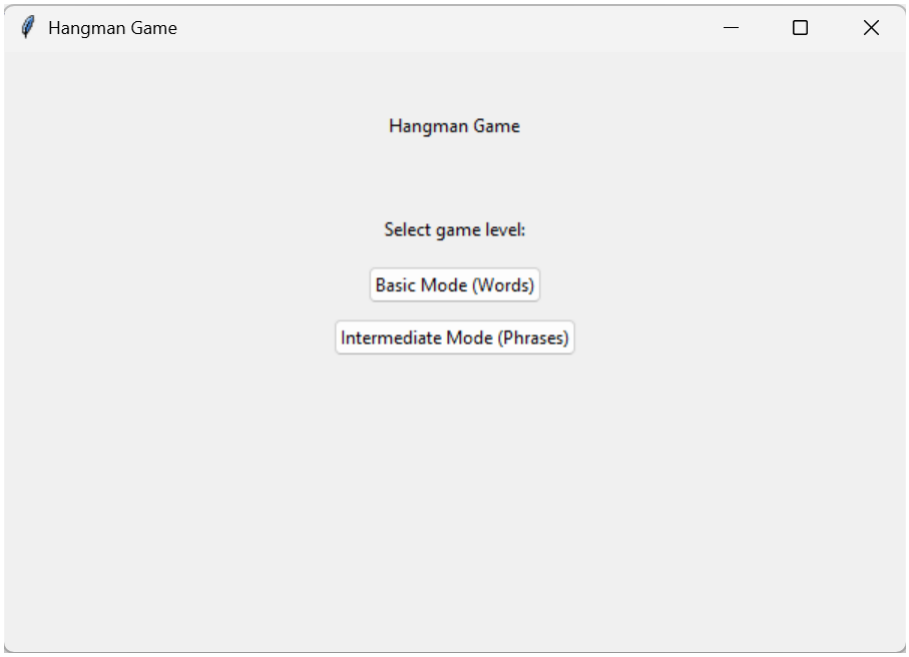
- Checked using pylint, and all warnings were fixed.

```
PS E:\Python\Hangman> python -m pylint .\hangman_game.py .\hangman_gui.py .\test_hangman.py .\words.py

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

PS E:\Python\Hangman>
```

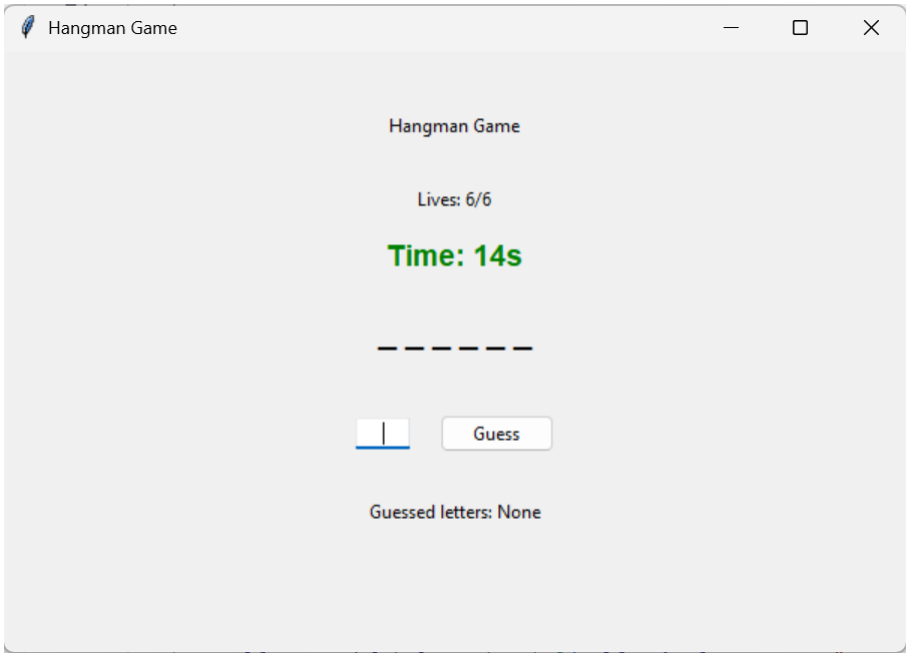
- Simplified GUI interface, focusing on core functionality.
- Cleaned up debug and useless files.

2.5. Screenshots of runtime

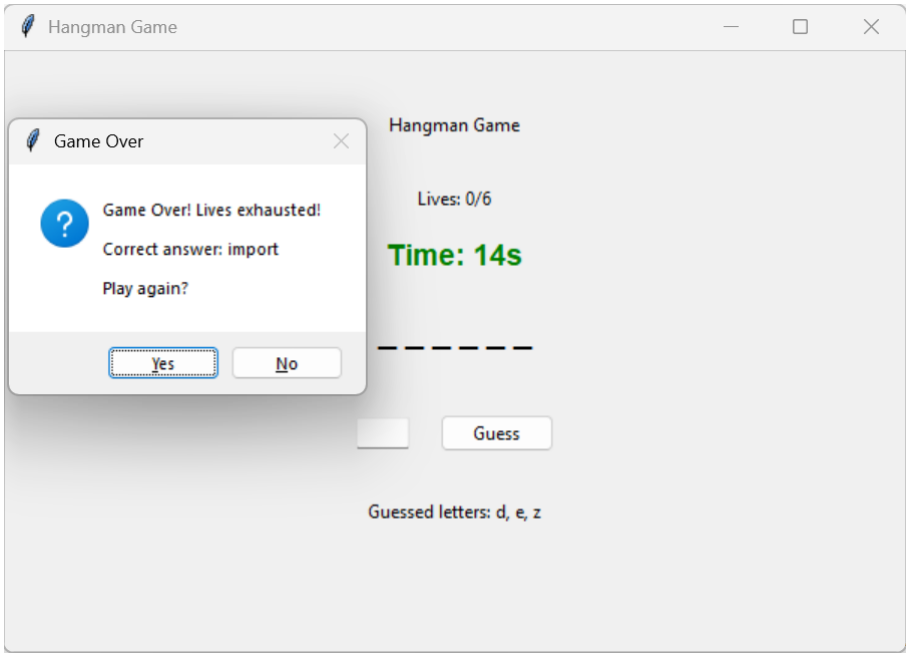
2.5.1 Game Level Selection



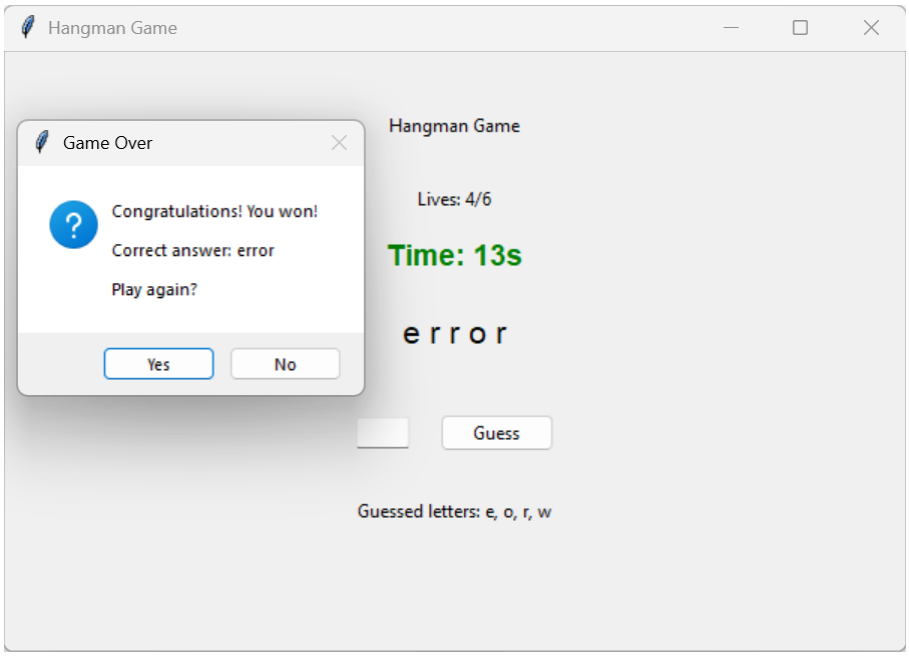
2.5.2 Guessing with Countdown Timer



2.5.3 Game Failed



2.5.4 Game Won



3 Conclusion

Repository: https://github.com/Jeremy0730/Hangman

3.1 Key Achievements

1. **Test-Driven Development (TDD) Practice**
    - Ensured code quality and functional correctness through a test-first approach.
    - 22 test cases with 100% pass rate, proving the effectiveness of the TDD method effectiveness.
    - Test cases not only verify functionality but also serve as code documentation.
2. **Python GUI Programming Experience**
    - Mastered basic usage of the tkinter library.
    - Learned to create responsive user interfaces.
    - Understood event-driven programming patterns for GUI applications.
3. **Software Engineering Best Practices**

- Modular design: Clear separation of responsibilities.
- Error handling: Comprehensive exception handling mechanisms.
- Code standards: Consistent code style and documentation comments.

3.2 Technical Experience

1. **Advantages of the Pytest Testing Framework**
   - Concise syntax, easy to write and maintain test cases.
   - Supports mocks and patches, facilitating unit testing.
   - Provides detailed test reports and failure information.
2. **TDD Development Process Insights**
   - Red-Green-Refactor cycle ensures code quality.
   - Test cases serve as requirement documentation, clarifying functional specifications.
   - The refactoring process is safer with test protection.
3. **Project Management and Code Organisation**
   - A clear project structure facilitates maintenance.
   - Appropriate documentation and comments improve code readability.
   - Importance of version control and dependency management.

3.3 Project Results

- **Complete Functionality**: Implemented all required functional requirements.
- **High Code Quality**: Ensured code correctness through TDD.
- **User-Friendly**: Simple and intuitive graphical interface.
- **Maintainable**: Clear code structure and comprehensive test coverage.