MAPÚA UNIVERSITY
SCHOOL OF ELECTRICAL, ELECTRONICS, AND COMPUTER ENGINEERING

Experiment 4:
Design Patterns and Unit Testing

CPE106L (Software Design Laboratory)

Danikka Villaron
Jessa Abigaile Tongol
Mark Vincent De Villa
Matthew Benedict Nepomuceno

Group No.: **2**
Section: **E03**

PreLab

Readings, Insights, and Reflection

Insights and Reflections
Python Projects 1st Edition
9781118909195

<Villaron> (Chapter 4: Building Desktop Applications)

This chapter discusses the basic parts you need to build desktop applications, like the user interface (UI), the core logic, and the data layers. It talks about why organizing your app properly is important, so it's easier to maintain and can grow if needed.

It also introduces some tools, like PyQt and PyGTK, which are frameworks that make creating graphical user interfaces (GUIs) much simpler. The chapter then goes over how to save app data and user preferences, which is super important for making the app feel personalized for the user.

Also, it talks about the best ways to log errors, which helps in debugging and makes the app more reliable overall.

Professional Python
9781119070832

<De Villa> (Chapter 11: Unit Testing)

Chapter 11 of the book "Professional Python," 1st edition, mentioned the two testing scenarios. The two scenarios may mimic a live environment or isolate specific codes and their dependencies. System tests are automated tests that run on an ecosystem that mimics a live environment to test specific codes and monitor its hypothetical performance in a live environment. Contrarily, the Unit test isolates select blocks of codes from the environment or the program itself to test their interactions with the requirements and dependencies of the code. Both scenarios have their advantages and disadvantages. Unit tests perform better when finding and testing changes in a function or blocks of codes, whereas System tests perform better when debugging or testing an application's performance when publicized. Additionally, the unit testing utilizes the assert keyword, testing whether an expression is true or false, and runs an assertion error when false, which may contain a custom error message.

PostLab

Programming Problems

1. Convert the oxo-logic.py module to reflect OOP design by creating a Game class.

```
''' This is the main logic for a Tic-tac-toe game.
It is not optimised for a quality game it simply
generates random moves and checks the results of
a move for a winning line. Exposed functions are:
newGame()
saveGame()
restoreGame()
userMove()
computerMove()
'''

import os, random
import oxo_data

class Game(object):

    def newGame():
        ' return new empty game'
        return list(" " * 9)

    def saveGame(game):
        ' save game to disk '
        oxo_data.saveGame(game)

    def restoreGame():
        ''' restore previously saved game.
        If game not restored successfully return new game'''
        try:
            game = oxo_data.restoreGame()
            if len(game) == 9:
                return game
            else: return newGame()
        except IOError:
            return newGame()

    def _generateMove(game):
        ''' generate a random cell from thiose available.
            If all cells are used return -1'''
        options = [i for i in range(len(game)) if  game[i] == " "]
        if options:
            return random.choice(options)
        else: return -1

    def _isWinningMove(game):
        wins = ((0,1,2), (3,4,5), (6,7,8),
```

Figure 1.1. Code of oxo_logic.py – Part 1

```
                (0,3,6), (1,4,7), (2,5,8),
                (0,4,8), (2,4,6))

        for a,b,c in wins:
            chars = game[a] + game[b] + game[c]
            if chars == 'XXX' or chars == 'OOO':
                return True
        return False

    def userMove(game,cell):
        if game[cell] != ' ':
            raise ValueError('Invalid cell')
        else:
            game[cell] = 'X'
        if Game._isWinningMove(game):
            return 'X'
        else:
            return ""

    def computerMove(game):
        cell = Game._generateMove(game)
        if cell == -1:
            return 'D'
        game[cell] = 'O'
        if Game._isWinningMove(game):
            return 'O'
        else:
            return ""

def test():
    result = ""
    game = Game.newGame()
    while not result:
        print(game)
        try:
            result = Game.userMove(game, Game._generateMove(game))
        except ValueError:
            print("Oops, that shouldn't happen")
        if not result:
            result = Game.computerMove(game)

        if not result: continue
        elif result == 'D':
            print("Its a draw")
        else:
```

Figure 1.2. Code of oxo_logic.py – Part 2

```
            print("Winner is:", result)
        print(game)

if __name__ == "__main__":
    test()
```

Figure 1.3. Code of oxo_logic.py – Part 3

Figures 1.1, 1.2, and 1.3 present the modified version of the oxo_logic.py codes, wherein the exposed functions (newGame(), saveGame(), restoreGame(), userMove(), and computerMove()) become methods of the newly defined Game class. Apart from redefining functions to become methods of a class and the test function's slight modifications for calling the redefined functions, the rest of the code remained the same, resulting in similar outputs as the original file.

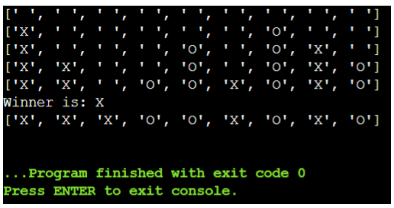

Figure 1.4. Output 1 of Originaloxo_logic.py

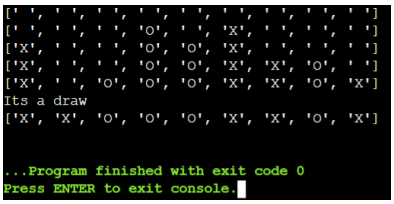

Figure 1.5. Output 1 of Originaloxo_logic.py

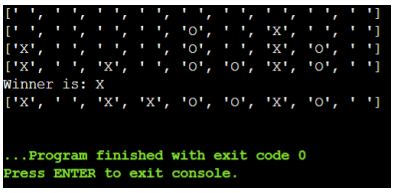

Figure 1.6. Output 1 of Originaloxo_logic.py

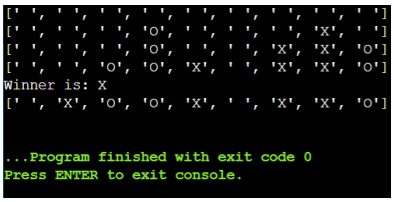

Figure 1.7. Output 1 of Modified oxo_logic.py

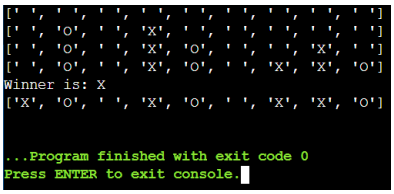

Figure 1.8. Output 2 of Modified oxo_logic.py

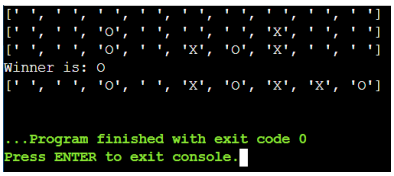

Figure 1.9. Output 3 of Modified oxo_logic.py

Figures 1.4, 1.5, and 1.6 present the sample outputs of the original codes, whereas 1.7, 1.8, and 1.9 present sample outputs of the modified code. Although the test() function randomly obtains values for the tic-tac-toe slots, the outputs of the original and altered are inherently similar.

2. Explore the Tkinter.filedialog module to get the name of a text file

```python
import tkinter as tk
import os
from tkinter import filedialog

def open_file():
    file_path = filedialog.askopenfilename(
        title="Open Text File",
        filetypes=[("Text Files", "*.txt"), ("All Files", "*.*")]
    )
    if file_path:
        print(f"Selected file: {os.path.basename(file_path)}")
        root.title(os.path.basename(file_path))

# main window
root = tk.Tk()
root.title("File")
root.geometry("300x150")

# Button to open file location
open_button = tk.Button(root, text="Open Text File", command=open_file)
open_button.pack(pady=50)

root.mainloop()
```

Figure 2.1. Tkinter File Dialogue Module Code

Figure 2.2. Newly Opened Dialogue Box

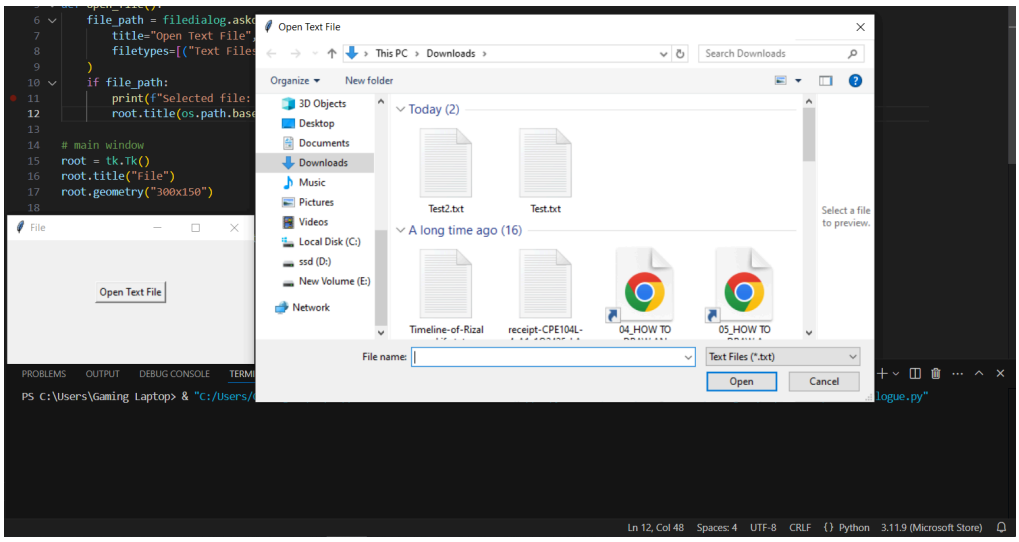

Figure 2.3. File Selection After Pressing Button

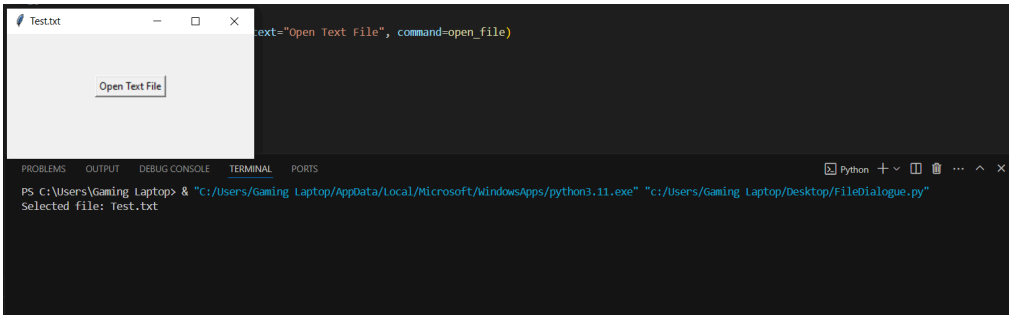

Figure 2.4. Changing of Window Name After File Selection

The above show a project of python that uses Tkinter to access a computer's files (specifically text (txt) files) and extracts their respective file names.I imported the os library so the program is able to access and read the names of selected files. The line of code that is

os.path.basename(file_path) is the module that is used to extract the file name of the selected file using file_path. After that, the file name is printed into the terminal and the root title of the main window is also changed into the file name to show that the name was taken properly.
This is done using print(f"Selected file: {os.path.basename(file_path)}") and

root.title(os.path.basename(file_path))

3. Create a unit test program for testing the Tic Tac Toe Console App

```python
import unittest
import os
from oxo_data import saveGame, restoreGame, _getPath

class TestOXOData(unittest.TestCase):
    def setUp(self):
        """Set up test environment: clean up existing game file if present."""
        self.game_file_path = os.path.join(_getPath(), "oxogame.dat")
        if os.path.exists(self.game_file_path):
            os.remove(self.game_file_path)

    def tearDown(self):
        """Clean up test environment: remove game file after test."""
        if os.path.exists(self.game_file_path):
            os.remove(self.game_file_path)

    def test_save_and_restore_game(self):
        """Test that saveGame and restoreGame work correctly."""
        game_data = list("XOXOXOXOX")  # Sample game board data
        saveGame(game_data)  # Save the game
        restored_game = restoreGame()  # Restore the game

        # Check if the restored game matches the saved game
        self.assertEqual(restored_game, game_data, "Restored game does not match saved game!")

    def test_save_creates_file(self):
        """Test that saveGame creates the game file."""
        game_data = list("XO XO XO ")  # Another sample game board data
        saveGame(game_data)

        # Check if the file exists
        self.assertTrue(os.path.exists(self.game_file_path), "Game file was not created!")

    def test_restore_empty_file(self):
        """Test restoreGame behavior with an empty file."""
        # Create an empty game file
        open(self.game_file_path, 'w').close()

        # Restore the game and check if it returns an empty list
        restored_game = restoreGame()
        self.assertEqual(restored_game, [], "Restored game from an empty file should be an empty list!")

    def test_restore_file_not_found(self):
        """Test restoreGame raises FileNotFoundError if no game file exists."""
        with self.assertRaises(FileNotFoundError, msg="restoreGame did not raise FileNotFoundError for missing file!"):
            restoreGame()

if __name__ == "__main__":
    unittest.main()
```

Figure 3.1. Code of test_oxo_data.py

```
....
----------------------------------------------------------------------
Ran 4 tests in 0.001s

OK


...Program finished with exit code 0
Press ENTER to exit console.
```

Figure 3.2. Output of test_oxo_data.py

The image above outlines a series of Python unit tests designed to evaluate the functionality of two key functions within the oxo_data Tic Tac Toe game application. The tests specifically target the saveGame function, which is responsible for storing the current game state,

and the restoreGame function, which retrieves this saved data. These tests encompass four distinct scenarios: verifying data consistency between saving and restoring, confirming successful file creation during saving, handling empty game files, and appropriately managing situations where the game file is missing. To execute these tests, users are instructed to save the test script as test_oxo_data.py alongside the oxo_data.py file and then run them using the command python -m unittest test_oxo_data.py. The test outcomes will provide valuable insights into the correct functioning of the oxo_data.py functions and aid in identifying any potential implementation issues.