QLC-3) – Topic Score Writer, a guided solution

Objective

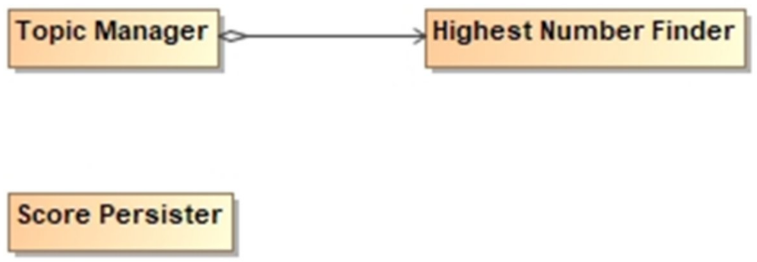
In this lab, you will develop a simple Python class called TopicScoreWriter using Test-Driven Development (TDD). You will write tests first, followed by the minimal production code needed to make those tests pass. Each step reflects the RED → GREEN → REFACTOR cycle of TDD. We will also follow a common pattern for structuring tests called Arrange-Act-Assert (AAA) that helps make tests more organised and readable.

Case Scenario: Storing Top Scores by Topic

An organisation delivers several topics (subjects). Students are graded on each topic. You are required to store the top score for each topic.

We've designed the application so that it comprises three core classes:

- A class to find the highest number from an array of integers. [DONE]
- A class to find the highest score for a topic. [DONE]
- A class to write the topic and score to a file on the disk.



You are going to follow a **TDD** approach to write the scores to a file.

Given the following specification

If the input is:

- ["Physics", 56, 67, 45, 89], the result should be ["Physics", 89]
- [] the result should be []
- [["Physics", [56, 67, 45, 89]], ["Art", [87, 66, 78]], the result should be [["Physics", 89], ["Art", 87]]
- [["Physics", [56, 67, 45, 89]], ["Art", [87, 66, 78]], ["Comp Sci", [45, 88, 97, 56]]], the result should be [["Physics", 89],["Art", 87],["Comp Sci", 97]]

Steps

1. **Create your first test file**
   Create a test script named test_topic_score_writer.py inside the tests folder.
   In this file:
   - Define a class called TestTopicScoreWriter (following PEP 8 naming conventions for test classes).
   - Ensure it inherits from unittest.TestCase, which provides the framework for writing unit tests in Python.
2. **Write your 1st test (RED)**
   Write your first test that verifies TopicScoreWriter correctly delegates the task of writing a topic score to a file-like object, using a mock to isolate file I/O. The test should:
   - Use a mock file writer to avoid real file I/O.
   - Prepare a single topic score ("Physics", 89) as test input.
   - Invoke the write_scores method to write this score.
   - Asserts that the mock writer's write_line method was called exactly once using the assert_called_with method.
   - Your first test method should be named something like:

     def test_verify_topic_score_details_written_out_once

```
import unittest
from unittest.mock import Mock
from app.topic_score_writer import TopicScoreWriter
from app.topic_top_score import TopicTopScore

class TestTopicScoreWriter(unittest.TestCase):
  def test_verify_topic_score_details_written_out_once(self):
    # Arrange
    physics = "Physics"
    expected_result = "Physics, 89"
    top_scores = [TopicTopScore(physics, 89)]

    mock_file_writer = Mock()
    writer = TopicScoreWriter(mock_file_writer)

    # Act
    writer.write_scores(top_scores)

    # Assert
    mock_file_writer.write_line.assert_called_once_with(expected_result)
```

3. **Write minimal Production Code (GREEN)**
   Create a new file called topic_score_writer.py in the app folder and define a new class called TopicScoreWriter:
   - Accepts a file writer object when initialised
   - Has a method write_scores that always writes "Physics, 89" to file writer regardless of input

   This is temporary production code as TDD is in progress and this stub is used to get an initial test to pass – to verify a single line was written. In a later step, this would be replaced with code that loops over top_scores and writes each one dynamically.

```
from app.file_writer import FileWriter

class TopicScoreWriter:
  def __init__(self, file_writer):
    self._file_writer = file_writer

  def write_scores(self, top_scores):
    self._file_writer.write_line("Physics, 89")
```

4. **Write Application dependencies**
   We now need to complete the initial parts of the implementation code to pass the test. Create a new file flle_writer.py in the app folder. Define the class FileWriter with a single method write_line() that accepts the line to write and the filename to write to.
   - Remember we are mocking this class to avoid writing to the real filesystem.
   - This class defines the interface that TopicScoreWriter depends on – file_writer could an instance of filewriter or a mock object pretending to be one.

```
class FileWriter:
  def write_line(self, line, filename="output.txt"):
    with open(filename, 'w') as file:
      file.write(line)
```

   - Run tests and ensure they have passed.
   - Commit code.

5. **REFACTOR re-write write_scores()**
   Re-write the write_scores() method so that it writes the data being passed.

```
from app.file_writer import FileWriter

class TopicScoreWriter:
  def __init__(self, file_writer):
    self._file_writer = file_writer

  def write_scores(self, top_scores, filename="output.txt"):
    if top_scores:
      tts = top_scores[0]
      data_to_write = f"{tts.get_topic_name()}, {tts.get_top_score()}"
      self._file_writer.write_line(data_to_write, filename)
```

   - Re-Run ALL tests to confirm no Regression.
   - Commit your refactored code.
   - The if statement introduces a new logic path in the code – we should have a test for this!

6. **Write your 2nd Test - Verify topic scores details not written (RED)**
   Write a new test that finds the highest score with one topic using a stub.
   - Name the test:
     def test_verify_topic_score_details_not_written
   - Use assert_not_called

```
def test_verify_topic_score_details_not_written(self):
  # Arrange
  top_scores = [] # Empty list simulating no scores
  mock_file_writer = Mock()
  cut = TopicScoreWriter(mock_file_writer)

  # Act
  cut.write_scores(top_scores)

  # Assert
  mock_file_writer.write_line.assert_not_called()
```

   - Ensure test passes
   - Commit code to Git

7. **Write minimal Production Code (GREEN)**
   There should be no need to modify the application production code, from the previous REFACTOR, for the tests to pass.

- Ensure test passes
- Commit code to Git

8. **Refactor Code - Optional.**
   No refactoring required at this stage.
   - Re-Run ALL tests to confirm no Regression.
   - Commit code to Git, if not done in previous step.

9. **Write your 3rd Test - Writes multiple scores to file (RED)**
   Write a new test that writes multiple scores to file.
   - Name the test:
     def test_verify_topic_score_details_written_out_multiple_times

```
def test_verify_topic_score_details_written_out_multiple_times(self):
    # Arrange
    physics = "Physics"
    art = "Art"
    comp_sci = "Comp Sci"

    physics_result = "Physics, 89"
    art_result = "Art, 87"
    comp_sci_result = "Comp Sci, 97"

    top_scores = [
        TopicTopScore(physics, 89),
        TopicTopScore(art, 87),
        TopicTopScore(comp_sci, 97)
    ]

    mock_file_writer = Mock()
    cut = TopicScoreWriter(mock_file_writer)

    # Act
    cut.write_scores(top_scores)

    # Assert: verify each call
    expected_calls = [
        call.write_line(physics_result),
        call.write_line(art_result),
        call.write_line(comp_sci_result)
    ]
    mock_file_writer.write_line.assert_has_calls(expected_calls, any_order=False)
    self.assertEqual(mock_file_writer.write_line.call_count, 3)
```

- Run all tests
- Commit code to Git

10. Write minimal Production Code (GREEN)

Modify the production code for write_scores() to write all top scores for each topic.

```
from app.file_writer import FileWriter

class TopicScoreWriter:
    def __init__(self, file_writer):
        self._file_writer = file_writer

    def write_scores(self, top_scores, filename="output.txt"):
        for tts in top_scores:
            data_to_write = f"{tts.get_topic_name()}, {tts.get_top_score()}"
            self._file_writer.write_line(data_to_write)
```

- Ensure all tests pass
- Commit code to Git