QLC-4) – Using TDD to design production code

QL-4.1: Phase 1

In this exercise, you'll apply **Test-Driven Development (TDD)** to implement a simple file loader class in Python.

Instructions:

1. **Create a new Test Class**
   - Name it **TestLiveFileLoader** following Python's unittest naming convention.
   - This test class will drive the development of your production class **FileLoader**.
2. **Write the first Unit Test**
   - Begin by writing a test for the class-under-test (CUT), **FileLoader**.
   - The FileLoader should have **one public method**:
3. **Develop the Production Code**
   - Write only the code necessary in FileLoader to make your test pass.
   - Follow the red-green-refactor cycle of TDD.

```
def load_file(self, fname: str) -> int:
    """
    Loads the file specified by fname and returns the total number of
characters.
```

The function above is responsible for loading a text file from disk and returning the **total number of characters** contained in the file.

In your test:

- Specify the **input filename**, including the full path.
- Define the **expected character count** for the given file.
- Use an assertion to compare the actual number of characters read from the file against the expected value.

In the production class, you may use a method such as readlines() to load the file content and then calculate the character count by summing the lengths of each line.

Reflective Question:

"What is the weakness of this design?"

Once your test passes, consider the design implications:

- Is the class tightly coupled to external resources (e.g., the file system)?
- How might this affect testability, reliability, or reuse?

Be ready to discuss how this approach could be improved in the next phase.

A solution can be found in QL-4.1) TDD FileLoader v1.1 file

QL-4.2: Phase 2 – Using TDD to Design a Flexible FileLoader

In this phase, you'll continue to apply Test-Driven Development (TDD) by enhancing the design of your FileLoader class.

1. **Create a new test class** called TestFileLoader.
2. Write a unit test **first**, before implementing the actual code.
3. Your Code Under Test (CUT) is the FileLoader class, which should now include a new public method that accepts a strategy or function for how the file should be read.

Design Hint:

To make your design more flexible and testable, consider passing the file-loading logic into the method as a function argument. In Python, this could look something like:

```
# Act
bytes_read = cut.load_file(lambda fname:  open(fname, encoding="utf-8").readlines()
  if fname else []
)
```

This approach allows you to decouple the file-reading strategy from the FileLoader implementation, making it easier to mock or change the behaviour during testing.

```
import unittest
from app.file_loader import FileLoader

class TestFileLoader(unittest.TestCase):

  def test_load_all_of_file_using_inbuilt_files_type_as_lambda(self):
    # Arrange
    file_to_load = "sample.txt"
    cut = FileLoader(file_to_load)

    # Calculate expected number of characters
    with open(file_to_load, encoding="utf-8") as f:
      expected_bytes_read = sum(len(line) for line in f.readlines())

    # Act
    bytes_read = cut.load_file_with_func(lambda fname: open(fname,
        encoding="utf-8").readlines())

    # Assert
    self.assertEqual(expected_bytes_read, bytes_read)

if __name__ == '__main__':
```

Modify the FileLoader class to include the following functionality:

```
def get_lines(self):
  """Returns the list of lines read from the file."""
  return self.lines

def load_file_with_func(self, func):
  """
  Accepts a file loading function to inject lines, used primarily for testing.

  This avoids direct I/O operations and makes the method more testable by passing
  a mock or simulated version of file loading logic.
  """
  self.lines = func(self.file_to_load)
  return self._calculate_file_size()
```

Reflection Question

"What is the weakness of this design?"

Discussion:

Originally, FileLoader handled both how the file was read and what was done with the contents. This tight coupling makes the class difficult to test in isolation—especially if it relies on reading actual files from disk.

To address this, we refactor the design so that the file-loading mechanism is injected as a function (or lambda). This separates concerns, improves testability, and decouples FileLoader from the filesystem.

This approach forces us to clarify the true responsibility of FileLoader. Instead of just reading from disk, FileLoader now becomes a reusable component capable of working with any data source provided via the injected loader function. We have broadened its design so its more flexible and less brittle.

A solution can be found in QL-4.2) TDD FileLoader v1.2 file

QL-4.3: Phase 3 – Working with stubs

In Phase 2, the test had a significant weakness—it violated a core principle of good unit testing:

- A unit test should not perform any I/O operations.

Although we had introduced flexibility by allowing a lambda to be passed into FileLoader, the test still relied on the file system.

In this phase, you'll redesign the test to avoid real file access and instead use canned (predefined) data.

Instructions

1. Create a new test that specifically targets the use of canned data.
2. In the # Act section of the test, pass a lambda function that returns a fixed list of strings (e.g., ["Hello", "World"]) instead of reading from a file.
3. Do not change the FileLoader implementation. This confirms that your class is now decoupled from file system dependencies.
4. Assert that the returned value (e.g., total character count) matches the expected value for the canned data.

Reflection

Once you've completed the test, consider this question:

"Can we further improve the test?"

A possible solution or improvement can be found in the next section.

A solution can be found in QL-4.3) TDD FileLoader v1.3 file

QL-4.4: Phase 4 – Working with mocks

In this phase, we will explore how mocking works in Python and how we can use it to decouple our tests from the file system.

We aim to:

- Test the behaviour of our FileLoader class.
- Use a mock object to simulate reading lines from a file.
- Decouple the logic so our test does not perform real I/O.

Instructions

1. **Create a mock interface:**
   Instead of mocking built-in file reading methods directly, we'll define a dummy callable that mimics a file reading mechanism.
2. **Add to your test class:**

```
def test_load_file_with_mocked_file_reader(self):
    # Arrange
    file_to_load = "c:/tmp/KeyboardHandler.java.txt"
    cut = FileLoader(file_to_load)
    expected_bytes_read = 10
    pretend_file_content = ["Hello", "world"]

    # Create a mock callable for reading files
    mock_file_reader = Mock()
    mock_file_reader.return_value = pretend_file_content

    # Act
    bytes_read = cut.load_file_with_func(lambda fname: mock_file_reader(fname))

    # Assert
    self.assertEqual(bytes_read, expected_bytes_read)
```

3. **Run the test** and observe that:
   - It doesn't rely on any actual files.
   - The mocked reader returns canned data.
   - The test checks the logic in isolation.

Key Concepts

- **Mocks vs Stubs:**
  - A stub returns hard-coded data.
  - A mock can act as a stub but also checks for interactions (e.g., method calls and arguments).
  - Mocks can enforce expectations — they check what gets passed in.

Deep Dive – Do Expectations Matter?

Try changing the lambda to this:

```
# Act with unexpected value
bytes_read = cut.load_file_with_func(lambda fname: mock_file_reader("XYZ"))
```

Since "XYZ" is not the expected input, the mock may return a different value (or None, depending on your setup), potentially causing the test to fail.

This illustrates an important point:

> When mocking, you're not just specifying what should be returned — you're also enforcing what values the mock is expected to receive.

A solution can be found in QL-4.3) TDD FileLoader v1.3 file

Summary

- Use mocks to simulate external dependencies like file I/O.
- Carefully define the input parameters you expect in your mocks.
- Keep your unit tests focused and free from real file access.
- Mocking helps design cleaner, decoupled, and more flexible code.

Approach	Description	Coupling Level	Flexibility	Test Isolation	External Dependency	Use Case
Tightly Coupled Code	The CUT directly creates or depends on concrete dependencies	High	Low	Poor	Often required (e.g., real files, DBs)	Simple but hard to test in isolation
Loosely Coupled Code	Dependencies are passed in (e.g., via constructor or function)	Medium	Medium to High	Better	Still uses real dependencies	Easier to test than tightly coupled
Using Stubs	Stub classes/functions return canned responses, no logic	Low	High	Good	None	Used to simulate predictable outputs
Using Mocks	Mocks simulate and verify behaviour and expectations	Low	Very High	Excellent	None	Used for testing interactions and behaviours

- **Tight Coupling**: Hard to test, requires real resources, brittle.
- **Loose Coupling**: Better design but may still involve real I/O unless stubs/mocks are used.
- **Stubs**: Fake objects with hard-coded responses, focus on data, not behaviour.
- **Mocks**: Fully programmable fakes that check how they're used (e.g., method calls, parameters).