QL-5) – Working with Untestable Code

Consider this small but innocuous piece of code:

```
from datetime import datetime

class FileLog:
  def clear_the_log(self):
    # Simulated method that would do something to files in the log
    print("Log cleared")  # Optional placeholder behaviour

class DataClerk:
  def __init__(self):
    self._file_log = FileLog()

  def process_data(self):
    now = datetime.now().time()
    stop_time = datetime.strptime("20:00", "%H:%M").time()

    if now < stop_time:
      print("Ready to process the data")
      fl = FileLog() # A NEW Instance of FileLog is created here!
      fl.clear_the_log() # Method clear_the_log is called.
```

Why is this untestable?

- The **FileLog** instance is created inside the process_data() method, so you can't inject a fake or mock version from a unit test.
- This hard-coded dependency makes **clear_the_log()** difficult to observe or verify from a test - unless you refactor to inject the dependency (via constructor or parameter).

And here is the test for the code above:

```
import unittest
from app.data_clerk import DataClerk

class TestDataClerk(unittest.TestCase):

  def test_process_data(self):
    # Arrange
    cut = DataClerk()

    # Act
    cut.process_data()

    # Assert
    # There is no assertion possible - we don't know if clear_the_log() was called
    pass

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

What's wrong with this test?

- **No Observable Effect:** The method process_data() prints a message and calls a method clear_the_log() on a new instance of FileLog, but none of that affects the testable state.
- **Hard-Coded Dependency:** Because FileLog is instantiated inside process_data(), there's no way for the test to replace it with a fake, stub, or mock.
- **Void Method / No Return:** process_data() doesn't return a value and doesn't change state in a way the test can observe.
- **No Assertion:** The test has no meaningful assert, because nothing measurable or verifiable happens that the test can detect.

Observations on DataClerk:

- **Hardcoded Time Constraint**

  The line that checks **if now < stop_time:** tightly couples the method's behaviour to the system clock. This makes it hard to reliably test both branches of the if statement, since tests will only pass or fail depending on the real system time when they are run.

- **Hardcoded Dependency on FileLog**

  The instantiation of FileLog directly inside the process_data() method makes it impossible to inject a fake or mock version. If FileLog.clear_the_log() includes file I/O, network, or database operations, it will run in every test - potentially causing side effects and making the test brittle or slow.

- **Suggested Refactor for FileLog**

  The issue with direct instantiation can be resolved by refactoring the code so that FileLog is passed in via the constructor or as a parameter to process_data(). This makes the class easier to test by allowing the injection of a mock or stub.

- **Suggested Refactor for Time Logic**

  The time-check logic should be isolated into a separate method, such as get_current_time(). This method can then be overridden or mocked during tests to control the behaviour of process_data() without relying on the actual system clock.

Activity: Refactoring Untestable Code

Objective

Improve testability of the DataClerk class by addressing hardcoded dependencies and tightly coupled logic.

Step-by-Step Instructions

1. **Initial Observation**

   If you run the current code before 8:00 PM, you should see the message "Ready to process the data".

   If you manually change the check to a time later than the current time, the message disappears.

   This demonstrates that the logic works, but this is **not a reliable test,** as it depends on the actual system time.

2. **Read "Activity 1 Observations"**

   These outline why the current implementation is untestable and suggest ideas for improvement.

3. **Refactor DataClerk to Inject the Dependency**
   - Refactor the class to accept a FileLog instance via the constructor.
   - This allows you to pass in a mock or stub in your tests.
   - Decide: should you inject a class instance or a protocol/interface (via Python's Protocol)?
4. **Test the Refactored Code**
   - Write a new test to confirm that the logic still behaves as expected.
   - Use unittest.mock.Mock to simulate the behavior of FileLog.
5. **Address the Hardcoded Time Logic**
   - Line now = datetime.now().time() is not test-friendly.
   - Refactor this into a protected or private method like _get_current_time() so it can be overridden or mocked.
   - Alternatively, pass a get_time lambda/function into process_data() for complete test control.
6. **Use Mocking to Verify Behaviour**
   - Use mock_file_log.clear_the_log.assert_called_once() to confirm that clear_the_log() was triggered correctly.

- This is a verification step—not checking state but checking interaction.

7. **Optional: Spy-Like Behaviour**
   - In Python, a Spy can be created by subclassing or wrapping the real object and overriding specific methods.
   - You could also use unittest.mock.patch.object() to override _get_current_time() in your test.

Solution can be found in why_spy_py git repo.