**Rules and Guidelines for Your Test Setup**

Based on your requirements and the professor's expectations, here's a summary of the rules and practices you need to follow for your testing environment and structure. I'll also provide detailed documentation for how this setup works.

**1. Clear and Detailed Logging:**

* **Logging in Each Test:** Every test should start with clear logging that indicates the beginning and end of the test, along with detailed logs of the expected and received outcomes in the entity and control layers.
* **Log Outputs for Each Layer:** Ensure that the entity and control layers log their expected and actual results separately to help identify any mismatches or issues.

**2. Testing Related Control and Entity Object Methods:**

* **Use Case Testing:** For every use case, you need to test the behavior of both the control and entity objects. These tests will ensure that the business logic and operations of both layers are functioning as expected.
* **Mocking Dependencies:** Use unittest.mock.patch to mock dependencies or external calls in both the control and entity objects, focusing on their internal behavior.

**3. 4 Test Cases for Each Use Case:**

For each use case, you need four distinct test cases to fully cover the behavior of the system:

* **Successful Outcome:** The happy path where the system behaves as expected.
* **Already Running/Error Handled by Control:** The case where the control layer detects that the operation has already been performed or gracefully handles an error.
* **Error in Control Layer:** When an error is thrown in the control layer, ensure the error is correctly handled and logged.
* **Error in Entity Layer:** When an error occurs in the entity layer, check that the control layer correctly processes and reports the error.

**Example: Launch Browser Use Case Documentation**

I'll explain the use case using the example of launching a browser, focusing on the control and entity layers.

**Overview**

**Use Case:** Launching a browser.

**Objective:** The control layer sends a request to the entity layer to launch a browser, handles the response, and returns the result.

**Tests**

1. **Test 1: Successful Launch (Entity and Control Layers)**
   * **Objective:** Test that the browser is launched successfully in both the entity and control layers.
   * **Mocking:** Mock the entity layer's method launch\_browser() to return a success message.
   * **Validation:**
     + **Entity Layer:** Verify that the entity layer returns the expected message ("Browser launched.").
     + **Control Layer:** Ensure that the control layer processes this message and returns "Control Object Result: Browser launched.".
   * **Logging:** Log the start of the test, expected and received messages for both layers, and the completion of the test.
2. **Test 2: Browser Already Running (Entity and Control Layers)**
   * **Objective:** Test the scenario where the browser is already running.
   * **Mocking:** Mock the entity layer to return "Browser is already running.".
   * **Validation:**
     + **Entity Layer:** Verify that the entity layer returns "Browser is already running.".
     + **Control Layer:** Ensure the control layer processes this message and returns "Control Object Result: Browser is already running.".
   * **Logging:** Similar to Test 1, ensure clear logging.
3. **Test 3: Control Layer Failure**
   * **Objective:** Test how the control layer handles an error in the entity layer.
   * **Mocking:** Simulate an exception in the entity layer using mock\_launch.side\_effect = Exception("Internal error").
   * **Validation:**
     + **Control Layer:** Ensure that the control layer correctly processes the entity error and returns an appropriate message, such as "Control Layer Exception: Internal error.".
   * **Logging:** Clear logging to identify when the exception was caught and how it was handled.
4. **Test 4: Entity Layer Failure**
   * **Objective:** Test the scenario where the entity layer throws an exception during the browser launch.
   * **Mocking:** Simulate an exception in the entity layer using mock\_launch.side\_effect = Exception("Failed to launch browser: Internal error").
   * **Validation:**
     + **Control Layer:** Check that the control layer captures and logs this exception properly.
   * **Logging:** Ensure detailed logging throughout the process.

**Logging Example for Each Test:**

For every test case, you want to ensure that the following logs are present:

* **Starting Test:** Log the start of the test.
* **Entity Layer Expected/Received:** Log the expected and received outcomes for the entity layer.
* **Control Layer Expected/Received:** Log the expected and received outcomes for the control layer.
* **Test Completion:** Log the successful completion of the test.

**Explanation of How This Works:**

* **pytestmark = pytest.mark.asyncio:** Marks the entire test file as using asynchronous code, allowing you to run await in tests.
* **Logging:** logging.info() is used throughout the tests to record both the expected and received outcomes at each layer. This ensures that you have a clear record of what is happening during each step of the test.
* **Mocking:** patch() is used to replace the actual method calls in the entity and control layers with mock objects. This lets you simulate different responses and behaviors without executing real logic.
* **Assertions:** assert statements check that the received values match the expected outcomes. If an assertion fails, the test will report which layer failed and why.
* **Test Structure:** Each test is independent, focusing on a different aspect of the use case (e.g., success, already running, failure in control or entity).

By adhering to these principles and structures, your testing environment will be well-documented, easily reproducible, and meet the expectations of your professor.

import pytest, logging

from unittest.mock import patch

from test\_init import base\_test\_case, log\_test\_start\_end, setup\_logging

# Enable asyncio for all tests in this file

pytestmark = pytest.mark.asyncio

setup\_logging()

async def test\_launch\_browser\_success(base\_test\_case):

    with patch('entity.BrowserEntity.BrowserEntity.launch\_browser') as mock\_launch:

        # Setup mock return and expected outcomes

        mock\_launch.return\_value = "Browser launched."

        expected\_entity\_result = "Browser launched."

        expected\_control\_result = "Control Object Result: Browser launched."

        # Execute the command

        result = await base\_test\_case.browser\_control.receive\_command("launch\_browser")

        # Log and assert the outcomes

        logging.info(f"Entity Layer Expected: {expected\_entity\_result}")

        logging.info(f"Entity Layer Received: {mock\_launch.return\_value}")

        assert mock\_launch.return\_value == expected\_entity\_result, "Entity layer assertion failed."

        logging.info("Unit Test Passed for entity layer.\n")

        logging.info(f"Control Layer Expected: {expected\_control\_result}")

        logging.info(f"Control Layer Received: {result}")

        assert result == expected\_control\_result, "Control layer assertion failed."

        logging.info("Unit Test Passed for control layer.")

async def test\_launch\_browser\_already\_running(base\_test\_case):

    with patch('entity.BrowserEntity.BrowserEntity.launch\_browser', return\_value="Browser is already running.") as mock\_launch:

        expected\_entity\_result = "Browser is already running."

        expected\_control\_result = "Control Object Result: Browser is already running."

        result = await base\_test\_case.browser\_control.receive\_command("launch\_browser")

        logging.info(f"Entity Layer Expected: {expected\_entity\_result}")

        logging.info(f"Entity Layer Received: {mock\_launch.return\_value}")

        assert mock\_launch.return\_value == expected\_entity\_result, "Entity layer assertion failed."

        logging.info("Unit Test Passed for entity layer.\n")

        logging.info(f"Control Layer Expected: {expected\_control\_result}")

        logging.info(f"Control Layer Received: {result}")

        assert result == expected\_control\_result, "Control layer assertion failed."

        logging.info("Unit Test Passed for control layer.")

async def test\_launch\_browser\_failure\_control(base\_test\_case):

    with patch('entity.BrowserEntity.BrowserEntity.launch\_browser', side\_effect=Exception("Internal error")) as mock\_launch:

        expected\_result = "Control Layer Exception: Internal error"

        result = await base\_test\_case.browser\_control.receive\_command("launch\_browser")

        logging.info(f"Control Layer Expected to Report: {expected\_result}")

        logging.info(f"Control Layer Received: {result}")

        assert result == expected\_result, "Control layer failed to handle or report the entity error correctly."

        logging.info("Unit Test Passed for control layer error handling.")

async def test\_launch\_browser\_failure\_entity(base\_test\_case):

    with patch('entity.BrowserEntity.BrowserEntity.launch\_browser', side\_effect=Exception("Failed to launch browser: Internal error")) as mock\_launch:

        expected\_control\_result = "Control Layer Exception: Failed to launch browser: Internal error"

        result = await base\_test\_case.browser\_control.receive\_command("launch\_browser")

        logging.info(f"Entity Layer Expected Failure: Failed to launch browser: Internal error")

        logging.info(f"Control Layer Received: {result}")

        assert result == expected\_control\_result, "Control layer failed to report entity error correctly."

        logging.info("Unit Test Passed for entity layer error handling.")

if \_\_name\_\_ == "\_\_main\_\_":

    pytest.main()

import sys, os, logging, pytest

from unittest.mock import patch, MagicMock

# Ensure all necessary paths are included for modules that tests need to access

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_))))

def setup\_logging():

    """Set up logging without timestamp and other unnecessary information."""

    logger = logging.getLogger()

    if not logger.hasHandlers():

        logging.basicConfig(level=logging.INFO, format='%(message)s')

# Custom fixture for logging test start and end

@pytest.fixture(autouse=True)

def log\_test\_start\_end(request):

    test\_name = request.node.name

    logging.info(f"------------------------------------------------------\nStarting test: {test\_name}\n")

    # Yield control to the test function

    yield

    # Log after the test finishes

    logging.info(f"\nFinished test: {test\_name}\n------------------------------------------------------")

# Import your control classes

from control.BrowserControl import BrowserControl

from control.AccountControl import AccountControl

from control.AvailabilityControl import AvailabilityControl

from control.PriceControl import PriceControl

from control.BotControl import BotControl

@pytest.fixture

def base\_test\_case():

    """Base test setup that can be used by all test functions."""

    test\_case = MagicMock()

    test\_case.browser\_control = BrowserControl()

    test\_case.account\_control = AccountControl()

    test\_case.availability\_control = AvailabilityControl()

    test\_case.price\_control = PriceControl()

    test\_case.bot\_control = BotControl()

    return test\_case

@pytest.fixture

def username():

    return "sample\_username"

@pytest.fixture

def account\_id():

    return "sample\_account\_id"

@pytest.fixture

def website():

    return "http://example.com"