--- defectCodeTry.py ---

```
import pytest

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

--- unitTest_check_availability.py ---

import sys, os, pytest, logging

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

from unittest.mock import patch, AsyncMock

from control.AvailabilityControl import AvailabilityControl

from entity.DataExportEntity import ExportUtils

"""

Executable steps for the 'Check_Availability' use case:

1. Control Layer Command Reception

This test will ensure that AvailabilityControl.receive_command() handles the "check_availability" command properly, including parsing and validating parameters such as URL and optional date string.

2. Availability Checking

This test focuses on the AvailabilityEntity.check_availability() function to verify that it correctly processes the availability check against a provided URL and optional date string. It will ensure that the availability status is accurately determined and returned.

3. Data Logging to Excel

This test checks that the event data is correctly logged to an Excel file using DataExportEntity.log_to_excel(). It will verify that the export includes the correct data formatting, timestamping, and file handling, ensuring data integrity.

4. Data Logging to HTML

Ensures that the event data is appropriately exported to an HTML file using DataExportEntity.export_to_html(). This test will confirm the data integrity and formatting in the

```
HTML output, ensuring it matches expected outcomes.
"""



# Testing the control layer's ability to receive and process the "check_availability" command
@pytest.mark.asyncio
async def test_control_layer_command_reception():
    logging.info("Starting test: Control Layer Command Reception for check_availability command")

    command_data = "check_availability"
    url = "https://example.com/reservation"
    date_str = "2023-10-10"

                            with        patch('control.AvailabilityControl.AvailabilityControl.receive_command',
new_callable=AsyncMock) as mock_receive:
        control = AvailabilityControl()
        await control.receive_command(command_data, url, date_str)

        logging.info("Verifying that the receive_command was called with correct parameters")
        mock_receive.assert_called_with(command_data, url, date_str)
        logging.info("Test passed: Control layer correctly processes 'check_availability'")

# Testing the availability checking functionality from the AvailabilityEntity
@pytest.mark.asyncio
async def test_availability_checking():
    with patch('entity.AvailabilityEntity.AvailabilityEntity.check_availability', new_callable=AsyncMock)
```

```
as mock_check:
        # Mock returns a tuple mimicking the real function's output
        mock_check.return_value = ("Checked availability: Availability confirmed",
                                   "Data saved to Excel file at ExportedFiles\\excelFiles\\check_availability.xlsx.",
                                                                   "HTML file saved and updated at ExportedFiles\\htmlFiles\\check_availability.html.")
                result = await AvailabilityControl().check_availability("https://example.com/reservation", "2023-10-10")

        # Properly access the tuple and check the relevant part
         assert "Availability confirmed" in result[0]  # Accessing the first element of the tuple where the status message is


# Testing the Excel logging functionality
@pytest.mark.asyncio
async def test_data_logging_excel():
    logging.info("Starting test: Data Logging to Excel for check_availability command")

     with patch('entity.DataExportEntity.ExportUtils.log_to_excel', return_value="Data saved to Excel file at path.xlsx") as mock_excel:
        excel_result = ExportUtils.log_to_excel("check_availability", "https://example.com", "Available")

        logging.info("Verifying Excel file creation and data logging")
        assert "path.xlsx" in excel_result, "Excel data logging did not return expected file path"
        logging.info("Test passed: Data correctly logged to Excel")
```

```
# Testing the HTML export functionality
@pytest.mark.asyncio
async def test_data_logging_html():
    logging.info("Starting test: Data Export to HTML for check_availability command")

    with patch('entity.DataExportEntity.ExportUtils.export_to_html', return_value="Data exported to HTML file at path.html") as mock_html:
        html_result = ExportUtils.export_to_html("check_availability", "https://example.com", "Available")

        logging.info("Verifying HTML file creation and data export")
        assert "path.html" in html_result, "HTML data export did not return expected file path"
        logging.info("Test passed: Data correctly exported to HTML")

if __name__ == "__main__":
    pytest.main([__file__])
```

```
--- unitTest_close_browser.py ---

import sys, os, pytest, logging

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

#################################################################################
################################

from unittest.mock import patch, AsyncMock, MagicMock

from control.BrowserControl import BrowserControl

from entity.BrowserEntity import BrowserEntity

"""

Executable steps for the !close_browser use case:

1. Control Layer Processing

This test ensures that BrowserControl.receive_command() handles the "!close_browser" command
correctly.


2. Browser Closing

This test focuses on the BrowserEntity.close_browser() method to ensure it executes the browser
closing process.


3. Response Generation

This test validates that the control layer correctly interprets the response from the browser closing
step and returns the appropriate result to the boundary layer.

"""


# Test for Control Layer Processing

@pytest.mark.asyncio

async def test_control_layer_processing():
```

```
    logging.info("Starting test: Control Layer Processing for close_browser")

    with patch('entity.BrowserEntity.BrowserEntity.close_browser') as mock_close:
        # Configure the mock to return different responses based on the browser state
        mock_close.side_effect = ["Browser closed successfully.", "No browser is currently open."]
        browser_control = BrowserControl()

        # First call simulates the browser being open and then closed
        result = await browser_control.receive_command("close_browser")
        assert result == "Control Object Result: Browser closed successfully."
        logging.info(f"Test when browser is initially open and then closed: Passed with '{result}'")

        # Second call simulates the browser already being closed
        result = await browser_control.receive_command("close_browser")
        assert result == "Control Object Result: No browser is currently open."
        logging.info(f"Test when no browser is initially open: Passed with '{result}'")


# Test for Browser Closing

def test_browser_closing():
    logging.info("Starting test: Browser Closing")

    # Patching the webdriver.Chrome directly at the point of instantiation
    with patch('selenium.webdriver.Chrome', new_callable=MagicMock) as mock_chrome:
        mock_driver = mock_chrome.return_value  # Mock the return value which acts as the driver
```

```
    mock_driver.quit = MagicMock()  # Mock the quit method of the driver

    browser_entity = BrowserEntity()
    browser_entity.browser_open = True  # Ensure the browser is considered open
    browser_entity.driver = mock_driver  # Set the mock driver as the browser entity's driver

    result = browser_entity.close_browser()

    mock_driver.quit.assert_called_once()  # Check if quit was called on the driver instance
    logging.info("Expected outcome: Browser quit method called.")
    logging.info(f"Actual outcome: {result}")

    assert result == "Browser closed."
    logging.info("Test passed: Browser closing was successful")


# Test for Response Generation
@pytest.mark.asyncio
async def test_response_generation():
    logging.info("Starting test: Response Generation for close_browser")

                            with patch('control.BrowserControl.BrowserControl.receive_command',
new_callable=AsyncMock) as mock_receive:
    mock_receive.return_value = "Browser closed successfully."

    browser_control = BrowserControl()
```

```
        result = await browser_control.receive_command("close_browser")

        logging.info("Expected outcome: 'Browser closed successfully.'")
        logging.info(f"Actual outcome: {result}")

        assert result == "Browser closed successfully."
        logging.info("Step 3 executed and Test passed: Response generation was successful")

# This condition ensures that the pytest runner handles the test run.
if __name__ == "__main__":
    pytest.main([__file__])
```

--- unitTest_get_price.py ---

```
import sys, os, pytest, logging
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from unittest.mock import patch, AsyncMock
from control.PriceControl import PriceControl


"""
Executable steps for the 'get_price' use case:
1. Control Layer Processing
    This test will ensure that PriceControl.receive_command() correctly processes the "get_price" command,
  including proper URL parameter handling and delegation to the get_price method.

2. Price Retrieval
    This test will verify that PriceEntity.get_price_from_page() retrieves the correct price from the webpage,
  simulating the fetching process accurately.

3. Data Logging to Excel
      This test checks that the price data is correctly logged to an Excel file using DataExportEntity.log_to_excel(),
  ensuring that data is recorded properly.

4. Data Logging to HTML
      This test ensures that the price data is correctly exported to an HTML file using DataExportEntity.export_to_html(),
```

validating the data export process.

5. Response Assembly and Output

This test will confirm that the control layer assembles and outputs the correct response, including price information,

Excel and HTML paths, ensuring the completeness of the response.

"""

```
# Testing the control layer's ability to process the "get_price" command
@pytest.mark.asyncio
async def test_control_layer_processing():
    logging.info("Starting test: Control Layer Processing for get_price command")

    # Mock the actual command handling to simulate command receipt and processing
    with patch('control.PriceControl.PriceControl.receive_command', new_callable=AsyncMock) as mock_receive:
        mock_receive.return_value = await PriceControl().get_price("https://example.com/product")
        result = await PriceControl().receive_command("get_price", "https://example.com/product")

            logging.info("Verifying that the receive_command correctly processed the 'get_price' command")
        assert "get_price" in str(mock_receive.call_args)
        logging.info("Test passed: Control layer processing correctly handles 'get_price'")

# Testing the price retrieval functionality from the PriceEntity
@pytest.mark.asyncio
```

```
async def test_price_retrieval():
    logging.info("Starting test: Price Retrieval from webpage")

        with patch('entity.PriceEntity.PriceEntity.get_price_from_page', return_value="100.00") as mock_price:
        price_control = PriceControl()
        result = await price_control.get_price("https://example.com/product")

        logging.info("Expected fetched price: '100.00'")
        assert "100.00" in result
        logging.info("Test passed: Price retrieval successful and correct")

# Testing the Excel logging functionality
@pytest.mark.asyncio
async def test_data_logging_excel():
    logging.info("Starting test: Data Logging to Excel")

    with patch('entity.DataExportEntity.ExportUtils.log_to_excel', return_value="Data saved to Excel file at path.xlsx") as mock_excel:
        price_control = PriceControl()
        _, excel_result, _ = await price_control.get_price("https://example.com/product")

        logging.info("Verifying Excel file creation and data logging")
        assert "path.xlsx" in excel_result
        logging.info("Test passed: Data correctly logged to Excel")
```

```
# Testing the HTML export functionality
@pytest.mark.asyncio
async def test_data_logging_html():
    logging.info("Starting test: Data Export to HTML")

    with patch('entity.DataExportEntity.ExportUtils.export_to_html', return_value="Data exported to HTML file at path.html") as mock_html:
        price_control = PriceControl()
        _, _, html_result = await price_control.get_price("https://example.com/product")

        logging.info("Verifying HTML file creation and data export")
        assert "path.html" in html_result
        logging.info("Test passed: Data correctly exported to HTML")

# Testing response assembly and output correctness
@pytest.mark.asyncio
async def test_response_assembly_and_output():
    logging.info("Starting test: Response Assembly and Output")

    # Mocking get_price to return a tuple of price, excel file path, and html file path
            with patch('control.PriceControl.PriceControl.get_price', new_callable=AsyncMock) as mock_get_price:
            mock_get_price.return_value = ("100.00", "Data saved to Excel file at path.xlsx", "Data exported to HTML at path.html")
        price_control = PriceControl()
        result = await price_control.receive_command("get_price", "https://example.com/product")
```

```
        # Unpack the result tuple for clarity
        price, excel_path, html_path = result

        logging.info("Checking response contains price, Excel and HTML paths")
        assert price == "100.00", "Price did not match expected value"
        assert "path.xlsx" in excel_path, "Excel path did not contain expected file name"
        assert "path.html" in html_path, "HTML path did not contain expected file name"

        logging.info("Test passed: Correct response assembled and output")

if __name__ == "__main__":
    pytest.main([__file__])
```

--- unitTest_login.py ---

```
import sys, os, pytest, logging
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from unittest.mock import patch, AsyncMock, Mock
from control.BrowserControl import BrowserControl
from entity.BrowserEntity import BrowserEntity

"""
Executable steps for the !login command use case:
1. Control Layer Processing
This test will ensure that BotControl.receive_command() handles the "!login" command correctly, including proper parameter passing and validation.

2. Website Interaction
This test will focus on the BrowserEntity.login() function to ensure it processes the request to log into the website using the provided credentials.

3. Response Generation
This test will validate that the control layer correctly interprets the response from the website interaction step and returns the appropriate result to the boundary layer.
"""

# test_bot_control_login.py
@pytest.mark.asyncio
async def test_control_layer_login():
    logging.info("Starting test: Control Layer Processing for Login")
```

```
    with patch('entity.BrowserEntity.BrowserEntity.login', new_callable=AsyncMock) as mock_login:
        mock_login.return_value = "Login successful!"
        browser_control = BrowserControl()

        result = await browser_control.receive_command("login", "example.com", "user", "pass")

        logging.info(f"Expected outcome: Control Object Result: Login successful!")
        logging.info(f"Actual outcome: {result}")

        assert result == "Control Object Result: Login successful!"
            logging.info("Step 1 executed and Test passed: Control Layer Processing for Login was successful")

@pytest.fixture
def browser_entity_setup():     # Fixture to setup the BrowserEntity for testing
    with patch('selenium.webdriver.Chrome') as mock_browser:    # Mocking the Chrome browser
        entity = BrowserEntity()    # Creating an instance of BrowserEntity
        entity.driver = Mock()  # Mocking the driver
        entity.driver.get = Mock()  # Mocking the get method
        entity.driver.find_element = Mock() # Mocking the find_element method
        return entity

def test_website_interaction(browser_entity_setup):
    logging.info("Starting test: Website Interaction for Login")
```

```
    browser_entity = browser_entity_setup   # Setting up the BrowserEntity
    browser_entity.login = Mock(return_value="Login successful!")   # Mocking the login method

    result = browser_entity.login("http://example.com", "user", "pass")  # Calling the login method

    logging.info("Expected to attempt login on 'http://example.com'")
    logging.info(f"Actual outcome: {result}")

    assert "Login successful!" in result    # Assertion to check if the login was successful
    logging.info("Step 2 executed and Test passed: Website Interaction for Login was successful")

# test_response_generation.py
@pytest.mark.asyncio
async def test_response_generation():
    logging.info("Starting test: Response Generation for Login")

                                with          patch('control.BrowserControl.BrowserControl.receive_command',
new_callable=AsyncMock) as mock_receive:
        mock_receive.return_value = "Login successful!"
        browser_control = BrowserControl()

        result = await browser_control.receive_command("login", "example.com", "user", "pass")

        logging.info("Expected outcome: 'Login successful!'")
        logging.info(f"Actual outcome: {result}")
```

```
        assert "Login successful!" in result
            logging.info("Step 3 executed and Test passed: Response Generation for Login was successful")


# This condition ensures that the pytest runner handles the test run.
if __name__ == "__main__":
    pytest.main([__file__])
```

--- unitTest_navigate_to_website.py ---

```
import sys, os, pytest
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from unittest.mock import patch, AsyncMock
from control.BrowserControl import BrowserControl
from entity.BrowserEntity import BrowserEntity
from test_init import logging

# Define executable steps from the provided use case
"""
Executable steps for the navigate_to_website command:
1. Command Processing and URL Extraction
   - Ensure that the command is correctly processed and the URL is extracted and passed accurately to the control layer.

2. Browser Navigation
   - Verify that the browser control object receives the command and correctly triggers navigation to the URL.

3. Response Generation
   - Check that the correct response about navigation success or failure is generated and would be passed back to the boundary.
"""

# Test for Command Processing and URL Extraction
@pytest.mark.asyncio
```

```
async def test_command_processing_and_url_extraction():
    logging.info("Starting test: test_command_processing_and_url_extraction")
                                with          patch('control.BrowserControl.BrowserControl.receive_command',
new_callable=AsyncMock) as mock_receive:
        mock_receive.return_value = "Navigating to URL"
        browser_control = BrowserControl()

        # Simulate receiving the navigate command with a URL
        result = await browser_control.receive_command("navigate_to_website", "http://example.com")

        logging.info(f"Expected outcome: 'Navigating to URL'")
        logging.info(f"Actual outcome: {result}")

        assert result == "Navigating to URL"
        logging.info("Step 1 executed and Test passed: Command Processing and URL Extraction was
successful")

# Test for Browser Navigation
@pytest.mark.asyncio
async def test_browser_navigation():
    logging.info("Starting test: test_browser_navigation")
      with  patch('entity.BrowserEntity.BrowserEntity.navigate_to_website',  new_callable=AsyncMock)
as mock_navigate:
        mock_navigate.return_value = "Navigation successful"
        browser_entity = BrowserEntity()
        result = await browser_entity.navigate_to_website("http://example.com")
```

```
        logging.info("Expected outcome: 'Navigation successful'")
        logging.info(f"Actual outcome: {result}")

        assert result == "Navigation successful"
        logging.info("Step 2 executed and Test passed: Browser Navigation was successful")

# Test for Response Generation
@pytest.mark.asyncio
async def test_response_generation():
    logging.info("Starting test: test_response_generation")
                                with patch('control.BrowserControl.BrowserControl.receive_command',
new_callable=AsyncMock) as mock_receive:
        mock_receive.return_value = "Navigation confirmed"
        browser_control = BrowserControl()

        result = await browser_control.receive_command("confirm_navigation", "http://example.com")

        logging.info("Expected outcome: 'Navigation confirmed'")
        logging.info(f"Actual outcome: {result}")

        assert result == "Navigation confirmed"
        logging.info("Step 3 executed and Test passed: Response Generation was successful")

# This condition ensures that the pytest runner handles the test run.
if __name__ == "__main__":
```

```
pytest.main([__file__])
```

--- unitTest_project_help.py ---

```
import sys, os, pytest, logging
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
################################################################################
################################
from unittest.mock import patch, AsyncMock
from control.BotControl import BotControl


"""
Executable steps for the project_help use case:
1. Control Layer Processing
This test will ensure that BotControl.receive_command() handles the "project_help" command correctly, including proper parameter passing.
"""


# test_project_help_control.py
@pytest.mark.asyncio
async def test_project_help_control():
    # Start logging the test case
    logging.info("Starting test: test_project_help_control")

    # Mocking the BotControl to simulate control layer behavior
        with patch('control.BotControl.BotControl.receive_command', new_callable=AsyncMock) as mock_command:
        # Setup the mock to return the expected help message
        expected_help_message = "Here are the available commands:..."
```

```
        mock_command.return_value = expected_help_message

        # Creating an instance of BotControl
        control = BotControl()

        # Simulating the command processing
        result = await control.receive_command("project_help")

        # Logging expected and actual outcomes
        logging.info(f"Expected outcome: '{expected_help_message}'")
        logging.info(f"Actual outcome: '{result}'")

        # Assertion to check if the result is as expected
        assert result == expected_help_message
        logging.info("Step 1 executed and Test passed: Control Layer Processing was successful")

# This condition ensures that the pytest runner handles the test run.
if __name__ == "__main__":
    pytest.main([__file__])
```

--- unitTest_receive_email.py ---

```
import sys, os, pytest, logging
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
################################################################################
################################
from unittest.mock import patch, AsyncMock
from control.BotControl import BotControl
from entity.EmailEntity import send_email_with_attachments
"""
Executable steps for the receive_email use case:
1. Control Layer Processing
This test will ensure that BotControl.receive_command() handles the "receive_email" command correctly, including proper parameter passing.

2. Email Handling
This test will focus on the EmailEntity.send_email_with_attachments() function to ensure it processes the request and handles file operations and email sending as expected.

3. Response Generation
This test will validate that the control layer correctly interprets the response from the email handling step and returns the appropriate result to the boundary layer.
"""

# test_bot_control.py
@pytest.mark.asyncio
async def test_control_layer_processing():
```

```
    # Start logging the test case
    logging.info("Starting test: test_control_layer_processing")

    # Mocking the email sending function to simulate email sending without actual I/O operations
        with patch('entity.EmailEntity.send_email_with_attachments', new_callable=AsyncMock) as
mock_email:
        mock_email.return_value = "Email with file 'testfile.txt' sent successfully!"
        # Creating an instance of BotControl
        bot_control = BotControl()

        # Calling the receive_command method and passing the command and filename
        result = await bot_control.receive_command("receive_email", "testfile.txt")

        # Logging expected and actual outcomes
        logging.info(f"Expected outcome: 'Email with file 'testfile.txt' sent successfully!'")
        logging.info(f"Actual outcome: {result}")

        # Assertion to check if the result is as expected
        assert result == "Email with file 'testfile.txt' sent successfully!"
        logging.info("Step 1 executed and Test passed: Control Layer Processing was successful")


# test_email_handling.py
def test_email_handling():
    # Start logging the test case
    logging.info("Starting test: test_email_handling")
```

```
    # Mocking the SMTP class to simulate sending an email
    with patch('smtplib.SMTP') as mock_smtp:
        # Simulating the sending of an email
        result = send_email_with_attachments("testfile.txt")

        # Logging expected and actual outcomes
        logging.info("Expected outcome: Contains 'Email with file 'testfile.txt' sent successfully!'")
        logging.info(f"Actual outcome: {result}")

        # Assertion to check if the result contains the success message
        assert "Email with file 'testfile.txt' sent successfully!" in result
        logging.info("Step 2 executed and Test passed: Email handling was successful")


# test_response_generation.py
@pytest.mark.asyncio
async def test_response_generation():
    # Start logging the test case
    logging.info("Starting test: test_response_generation")

    # Mocking the BotControl.receive_command to simulate control layer behavior
        with patch('control.BotControl.BotControl.receive_command', new_callable=AsyncMock) as
mock_receive:
        mock_receive.return_value = "Email with file 'testfile.txt' sent successfully!"
```

```
        # Creating an instance of BotControl
        bot_control = BotControl()

        # Calling the receive_command method and passing the command and filename
        result = await bot_control.receive_command("receive_email", "testfile.txt")

        # Logging expected and actual outcomes
        logging.info("Expected outcome: 'Email with file 'testfile.txt' sent successfully!'")
        logging.info(f"Actual outcome: {result}")

        # Assertion to check if the result is as expected
        assert "Email with file 'testfile.txt' sent successfully!" in result
        logging.info("Step 3 executed and Test passed: Response generation was successful")


# This condition ensures that the pytest runner handles the test run.
if __name__ == "__main__":
    pytest.main([__file__])



"""
@pytest.mark.asyncio
async def test_handle_receive_email():
    # Explanation: Patching the 'receive_command' to simulate control layer behavior without actual execution.
```

```
    with patch('control.BotControl.BotControl.receive_command', new_callable=AsyncMock) as mock_receive_command:
        # Expected return value from the mocked method
        mock_receive_command.return_value = "Email with file 'monitor_price.html' sent successfully!"

        # Instantiate BotControl to test the interaction within the control layer
        control = BotControl()

        # Explanation: This line simulates the control layer receiving the 'receive_email' command with a filename.
        result = await control.receive_command("receive_email", "monitor_price.html")

        # Logging the result to understand what happens when the command is processed
        logging.info(f'Result of receive_command: {result}')

        # Explanation: Assert that the mocked method returns the expected result
        assert result == "Email with file 'monitor_price.html' sent successfully!"
        # Explanation: Ensure that the method was called exactly once with expected parameters
        mock_receive_command.assert_called_once_with("receive_email", "monitor_price.html")
        """
```

--- unitTest_start_monitoring_availability.py ---

```
import sys, os, pytest, asyncio, logging
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
################################################################################
################################
from unittest.mock import patch, AsyncMock
from control.AvailabilityControl import AvailabilityControl


"""
Executable steps for the `start_monitoring_availability` use case:

1. Control Layer Processing:
        This test ensures that `AvailabilityControl.receive_command()` handles the "start_monitoring_availability" command correctly,
   including proper parameter passing for the URL, date, and frequency.

2. Availability Monitoring Initiation:
   This test verifies that the control layer starts the monitoring process by calling `check_availability()` at regular intervals.

3. Stop Monitoring Logic:
        This test confirms that the monitoring can be stopped correctly using the "stop_monitoring_availability" command and that the final results are collected.
"""

# Test 1: Control Layer Processing
```

```
@pytest.mark.asyncio
async def test_control_layer_processing():
    logging.info("Starting test: test_control_layer_processing")

    url = "https://example.com/availability"
    frequency = 1
    logging.info(f"Testing command processing for URL: {url} with frequency: {frequency}")

    # Mock the actual command handling to simulate command receipt and processing
                            with        patch('control.AvailabilityControl.AvailabilityControl.receive_command',
new_callable=AsyncMock) as mock_receive:
        logging.info("Patching receive_command method...")

        # Simulate receiving the 'start_monitoring_availability' command
         result = await AvailabilityControl().receive_command("start_monitoring_availability", url, None,
frequency)

        logging.info("Verifying if 'start_monitoring_availability' was processed correctly...")
        assert "start_monitoring_availability" in str(mock_receive.call_args)
        assert mock_receive.call_args[0][1] == url
        assert mock_receive.call_args[0][3] == frequency
        logging.info("Test passed: Control layer processed 'start_monitoring_availability' correctly.")

# Test 2: Availability Monitoring Initiation
@pytest.mark.asyncio
async def test_availability_monitoring_initiation():
```

```
    logging.info("Starting test: test_availability_monitoring_initiation")

    availability_control = AvailabilityControl()
    url = "https://example.com/availability"
    frequency = 3
    logging.info(f"Initiating availability monitoring for URL: {url} with frequency: {frequency}")

    # Mock the check_availability method to return a constant value
        with patch.object(availability_control, 'check_availability', new_callable=AsyncMock) as mock_check_availability:
        logging.info("Patching check_availability method...")
        mock_check_availability.return_value = "Available"

        # Start the monitoring process (monitoring in a separate task)
            monitoring_task = asyncio.create_task(availability_control.start_monitoring_availability(url, None, frequency))
        logging.info("Monitoring task started.")

        # Simulate a brief period of monitoring (e.g., for two intervals)
        await asyncio.sleep(8)
                logging.info(f"Simulated monitoring for 5 seconds, checking number of calls to check_availability.")

        # Check if check_availability was called twice due to the frequency
            assert mock_check_availability.call_count == 2, f"Expected 2 availability checks, but got {mock_check_availability.call_count}"
```

```
        logging.info("Test passed: Availability monitoring initiated and 'check_availability' called twice.")

        # Stop the monitoring
        logging.info("Stopping availability monitoring...")
        availability_control.stop_monitoring_availability()
        await monitoring_task  # Wait for the task to stop

    # Ensure monitoring stopped and results were collected
    assert len(availability_control.results) == 2
    logging.info(f"Test passed: Monitoring stopped with {len(availability_control.results)} results.")

# Test 3: Stop Monitoring Logic
@pytest.mark.asyncio
async def test_stop_monitoring_logic():
    logging.info("Starting test: test_stop_monitoring_logic")

    availability_control = AvailabilityControl()
    url = "https://example.com/availability"
    frequency = 1
    logging.info(f"Initiating monitoring to test stopping logic for URL: {url} with frequency: {frequency}")

    # Mock check_availability method
        with patch.object(availability_control, 'check_availability', new_callable=AsyncMock) as mock_check_availability:
        logging.info("Patching check_availability method...")
        mock_check_availability.return_value = "Available"
```

```
    # Start monitoring
        monitoring_task = asyncio.create_task(availability_control.start_monitoring_availability(url,
None, frequency))
    logging.info("Monitoring task started.")

    # Simulate monitoring for one interval
    await asyncio.sleep(2)
    logging.info("Simulated monitoring for 6 seconds, stopping monitoring now.")

    # Stop the monitoring
    availability_control.stop_monitoring_availability()
    await monitoring_task  # Wait for the task to stop

    # Ensure the monitoring has stopped
    assert availability_control.is_monitoring == False
    assert len(availability_control.results) >= 1
    logging.info(f"Test passed: Monitoring stopped with {len(availability_control.results)} result(s).")

if __name__ == "__main__":
    pytest.main([__file__])
```

--- unitTest_start_monitoring_price.py ---

```
import sys, os, pytest, asyncio, logging
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from unittest.mock import patch, AsyncMock
from control.PriceControl import PriceControl


"""
Executable steps for the `start_monitoring_price` use case:

1. Control Layer Processing:
        This test will ensure that `PriceControl.receive_command()` correctly handles the "start_monitoring_price" command,
    including proper URL and frequency parameter passing.

2. Price Monitoring Initiation:
     This test will verify that the control layer starts the monitoring process by repeatedly calling `get_price()` at regular intervals.

3. Stop Monitoring Logic:
    This test confirms that the monitoring can be stopped correctly using the "stop_monitoring_price" command and that final results are collected.
"""

# Test 1: Control Layer Processing for start_monitoring_price command
@pytest.mark.asyncio
async def test_control_layer_processing():
```

```
    logging.info("Starting test: test_control_layer_processing")

    url = "https://example.com/product"
    frequency = 2
    logging.info(f"Testing command processing for URL: {url} with frequency: {frequency}")

    # Mock the actual command handling to simulate command receipt and processing
        with patch('control.PriceControl.PriceControl.receive_command', new_callable=AsyncMock) as
mock_receive:
        logging.info("Patching receive_command method...")

        # Simulate receiving the 'start_monitoring_price' command
        result = await PriceControl().receive_command("start_monitoring_price", url, frequency)

        logging.info("Verifying if 'start_monitoring_price' was processed correctly...")
        assert "start_monitoring_price" in str(mock_receive.call_args)
        assert mock_receive.call_args[0][1] == url
        assert mock_receive.call_args[0][2] == frequency
        logging.info("Test passed: Control layer processed 'start_monitoring_price' correctly.")

# Test 2: Price Monitoring Initiation
@pytest.mark.asyncio
async def test_price_monitoring_initiation():
    logging.info("Starting test: test_price_monitoring_initiation")

    price_control = PriceControl()
```

```
    url = "https://example.com/product"
    frequency = 3
    logging.info(f"Initiating price monitoring for URL: {url} with frequency: {frequency}")

    # Mock the get_price method to return a constant value
    with patch.object(price_control, 'get_price', new_callable=AsyncMock) as mock_get_price:
        logging.info("Patching get_price method...")
        mock_get_price.return_value = "100.00"

        # Start the monitoring process (monitoring in a separate task)
        monitoring_task = asyncio.create_task(price_control.start_monitoring_price(url, frequency))
        logging.info("Monitoring task started.")

        # Simulate a brief period of monitoring (e.g., two intervals)
        await asyncio.sleep(8)
        logging.info(f"Simulated monitoring for 5 seconds, checking number of calls to get_price.")

        # Check if get_price was called twice due to the frequency
                assert mock_get_price.call_count == 2, f"Expected 2 price checks, but got {mock_get_price.call_count}"
        logging.info("Test passed: Price monitoring initiated and 'get_price' called twice.")

        # Stop the monitoring
        logging.info("Stopping price monitoring...")
        price_control.stop_monitoring_price()
        await monitoring_task  # Wait for the task to stop
```

```
    # Ensure monitoring stopped and results were collected
    assert len(price_control.results) == 2
    logging.info(f"Test passed: Monitoring stopped with {len(price_control.results)} results.")

# Test 3: Stop Monitoring Logic
@pytest.mark.asyncio
async def test_stop_monitoring_logic():
    logging.info("Starting test: test_stop_monitoring_logic")

    price_control = PriceControl()
    url = "https://example.com/product"
    frequency = 2
    logging.info(f"Initiating monitoring to test stopping logic for URL: {url} with frequency: {frequency}")

    # Mock get_price method
    with patch.object(price_control, 'get_price', new_callable=AsyncMock) as mock_get_price:
        logging.info("Patching get_price method...")
        mock_get_price.return_value = "100.00"

        # Start monitoring
        monitoring_task = asyncio.create_task(price_control.start_monitoring_price(url, frequency))
        logging.info("Monitoring task started.")

        # Simulate monitoring for one interval
        await asyncio.sleep(3)
```

```
        logging.info("Simulated monitoring for 3 seconds, stopping monitoring now.")

        # Stop the monitoring
        price_control.stop_monitoring_price()
        await monitoring_task  # Wait for the task to stop

        # Ensure the monitoring has stopped
        assert price_control.is_monitoring == False
        assert len(price_control.results) >= 1
        logging.info(f"Test passed: Monitoring stopped with {len(price_control.results)} result(s).")

if __name__ == "__main__":
    pytest.main([__file__])
```

--- unitTest_stop_monitoring_availability.py ---

```
import sys, os, pytest, logging, asyncio
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
################################################################################
################################
from unittest.mock import patch, AsyncMock
from control.AvailabilityControl import AvailabilityControl


"""
Executable steps for the 'Stop_monitoring_availability' use case:

1. Control Layer Processing:
        This test ensures that `AvailabilityControl.receive_command()` correctly handles the "stop_monitoring_availability" command.

2. Monitoring Termination:
   This test verifies that the control layer terminates an ongoing availability monitoring session.

3. Final Results Summary:
   This test confirms that the control layer returns the correct summary of monitoring results once the process is terminated.
"""

# Test 1: Control Layer Processing for stop_monitoring_availability command
@pytest.mark.asyncio
async def test_control_layer_processing():
```

```
    logging.info("Starting test: Control Layer Processing for stop_monitoring_availability command")

                    with patch('control.AvailabilityControl.AvailabilityControl.receive_command',
new_callable=AsyncMock) as mock_receive:
        # Simulate receiving the 'stop_monitoring_availability' command
        result = await AvailabilityControl().receive_command("stop_monitoring_availability")

        # Verify that the command was processed correctly
        assert "stop_monitoring_availability" in str(mock_receive.call_args)
            logging.info("Test passed: Control layer processed stop_monitoring_availability command
successfully.")

# Test 2: Monitoring Termination
@pytest.mark.asyncio
async def test_monitoring_termination():
    logging.info("Starting test: Monitoring Termination for stop_monitoring_availability")

    availability_control = AvailabilityControl()
    availability_control.is_monitoring = True  # Simulate that monitoring is active
        availability_control.results = ["Availability at URL was available.", "Availability was checked
again."]

    # Simulate monitoring stop
    logging.info("Stopping availability monitoring...")
    result = availability_control.stop_monitoring_availability()
```

```
    # Verify that monitoring was stopped and flag was set correctly
    assert availability_control.is_monitoring == False
    logging.info("Test passed: Monitoring was terminated successfully.")

# Test 3: Final Results Summary
@pytest.mark.asyncio
async def test_final_summary_generation():
    logging.info("Starting test: Final Results Summary for stop_monitoring_availability")

    availability_control = AvailabilityControl()
    availability_control.is_monitoring = True  # Simulate an ongoing monitoring session
        availability_control.results = ["Availability at URL was available.", "Availability was checked again."]

    # Simulate the monitoring stop and ensure results are collected
    logging.info("Stopping availability monitoring and generating final summary...")
    result = availability_control.stop_monitoring_availability()

    # Verify that the summary contains the expected results
    assert "Availability at URL was available." in result
    assert "Availability was checked again." in result
    assert "Monitoring stopped successfully!" in result
    logging.info("Test passed: Final summary generated correctly.")

if __name__ == "__main__":
    pytest.main([__file__])
```

--- unitTest_stop_monitoring_price.py ---

```
import sys, os, pytest, logging
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from unittest.mock import patch, AsyncMock
from control.PriceControl import PriceControl


"""
Executable steps for the `stop_monitoring_price` use case:

1. Control Layer Processing:
        This test will ensure that `PriceControl.receive_command()` correctly handles the "stop_monitoring_price" command,
    including the proper termination of the price monitoring process.

2. Stop Monitoring Logic:
     This test verifies that the control layer stops the price monitoring process and collects the final results correctly.

3. Final Summary Generation:
    This test confirms that the control layer generates and returns a final summary of the monitoring session, containing the collected price results.
"""

# Test 1: Control Layer Processing for stop_monitoring_price command
@pytest.mark.asyncio
async def test_control_layer_processing():
```

```
    logging.info("Starting test: test_control_layer_processing")

    # Mock the actual command handling to simulate command receipt and processing
        with patch('control.PriceControl.PriceControl.receive_command', new_callable=AsyncMock) as mock_receive:
        logging.info("Patching receive_command method...")

        # Simulate receiving the 'stop_monitoring_price' command
        result = await PriceControl().receive_command("stop_monitoring_price")

        logging.info("Verifying if 'stop_monitoring_price' was processed correctly...")
        assert "stop_monitoring_price" in str(mock_receive.call_args)
                logging.info("Test passed: Control layer processed 'stop_monitoring_price' command correctly.")

# Test 2: Stop Monitoring Logic
@pytest.mark.asyncio
async def test_stop_monitoring_logic():
    logging.info("Starting test: test_stop_monitoring_logic")

    price_control = PriceControl()
    price_control.is_monitoring = True  # Simulate an ongoing monitoring session

    # Mock the stop_monitoring_price method
                                    with patch.object(price_control, 'stop_monitoring_price', wraps=price_control.stop_monitoring_price) as mock_stop_monitoring:
```

```
        logging.info("Patching stop_monitoring_price method...")

        # Simulate the stop command
        result = price_control.stop_monitoring_price()

        logging.info("Checking if monitoring stopped and results were collected...")
        assert price_control.is_monitoring == False
        logging.info("Monitoring was successfully stopped.")
        assert len(price_control.results) >= 0  # Ensuring that results were collected
        logging.info("Results were collected successfully.")
        logging.info("Test passed: Stop monitoring logic executed correctly.")


# Test 3: Final Summary Generation
@pytest.mark.asyncio
async def test_final_summary_generation():
    logging.info("Starting test: test_final_summary_generation")

    price_control = PriceControl()
    price_control.is_monitoring = True  # Simulate an ongoing monitoring session
    price_control.results = ["Price at URL was $100", "Price dropped to $90"]  # Mock some results

    # Simulate the monitoring stop and ensure results are collected
    logging.info("Stopping price monitoring and generating final summary...")
    result = price_control.stop_monitoring_price()
```

```
    # Ensure that the summary contains the expected results
    logging.info("Verifying the final summary contains the collected results...")
    assert "Price at URL was $100" in result
    assert "Price dropped to $90" in result
    assert "Price monitoring stopped successfully!" in result  # Updated to match the actual result
    logging.info("Test passed: Final summary generated correctly.")


if __name__ == "__main__":
    pytest.main([__file__])
```