--- defectCodeTry.py ---

from test\_init import \*  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main()

--- test\_init.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, MagicMock, Mock  
  
from control.AvailabilityControl import AvailabilityControl  
from control.PriceControl import PriceControl  
from control.BrowserControl import BrowserControl  
from control.BotControl import BotControl  
  
from entity.BrowserEntity import BrowserEntity  
from entity.DataExportEntity import ExportUtils  
from entity.PriceEntity import PriceEntity  
from entity.AvailabilityEntity import AvailabilityEntity  
from entity.EmailEntity import send\_email\_with\_attachments

--- unitTest\_check\_availability.py ---

from test\_init import \*  
"""  
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 ---

from test\_init import \*  
"""  
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 ---

from test\_init import \*  
"""  
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 ---

from test\_init import \*  
"""  
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 ---

from test\_init import \*  
# 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 ---

from test\_init import \*  
"""  
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 ---

from test\_init import \*  
"""  
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 ---

from test\_init import \*  
"""  
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 ---

from test\_init import \*  
"""  
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 ---

from test\_init import \*  
"""  
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 ---

from test\_init import \*  
"""  
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\_\_])