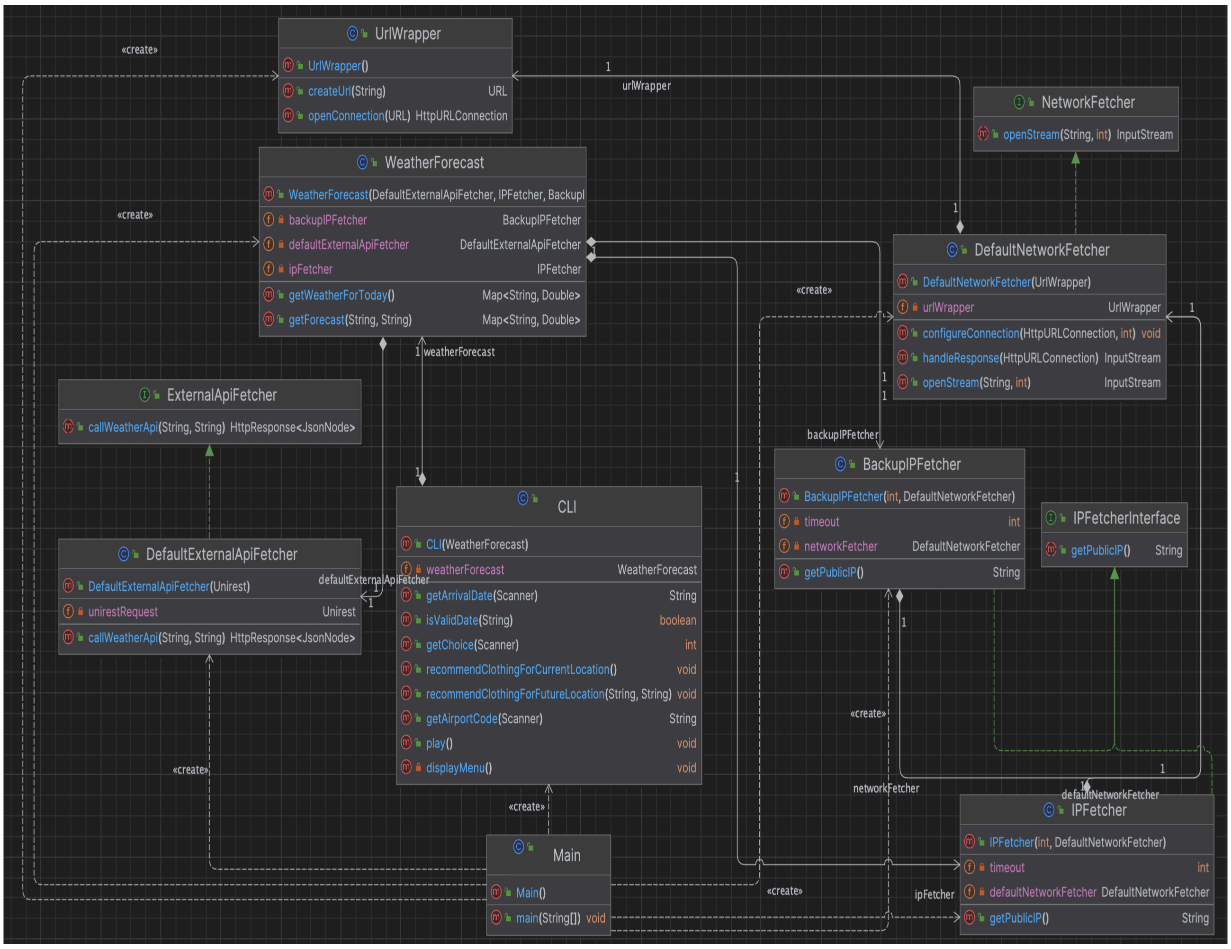
Assignment Part 1 (of 3) - Unit Testing

Program Diagram


UrlWrapper
UrlWrapper()
createUrl(String) URL
openConnection(URL) HttpURLConnection
WeatherForecast
WeatherForecast(DefaultExternalApiFetcher, IPFetcher, BackupI
backupIPFetcher BackupIPFetcher
defaultExternalApiFetcher DefaultExternalApiFetcher
ipFetcher IPFetcher
getWeatherForToday() Map<String, Double>
getForecast(String, String) Map<String, Double>
NetworkFetcher
openStream(String, int) InputStream
DefaultNetworkFetcher
DefaultNetworkFetcher(UrlWrapper)
urlWrapper UrlWrapper
configureConnection(HttpURLConnection, int) void
handleResponse(HttpURLConnection) InputStream
openStream(String, int) InputStream
ExternalApiFetcher
callWeatherApi(String, String) HttpResponse<JsonNode>
DefaultExternalApiFetcher
DefaultExternalApiFetcher(Unirest)
unirestRequest Unirest
callWeatherApi(String, String) HttpResponse<JsonNode>
CLI
CLI(WeatherForecast)
weatherForecast WeatherForecast
getArrivalDate(Scanner) String
isValidDate(String) boolean
getChoice(Scanner) int
recommendClothingForCurrentLocation() void
recommendClothingForFutureLocation(String, String) void
getAirportCode(Scanner) String
play() void
displayMenu() void
BackupIPFetcher
BackupIPFetcher(int, DefaultNetworkFetcher)
timeout int
networkFetcher DefaultNetworkFetcher
getPublicIP() String
IPFetcherInterface
getPublicIP() String
IPFetcher
IPFetcher(int, DefaultNetworkFetcher)
timeout int
defaultNetworkFetcher DefaultNetworkFetcher
getPublicIP() String
Main
Main()
main(String[]) void
«create»
urlWrapper
weatherForecast
backupIPFetcher
defaultExternalApiFetcher
networkFetcher
defaultNetworkFetcher
ipFetcher
1

Main – starts the program by calling CLI.play function.
CLI - provides a Command-line interface. It includes a method play() that contains an infinite loop using the displayMenu() method to showcase the menu and collect user input. It utilizes the getChoice() method for input retrieval. The user can choose from 3 options. If they enter a number outside the range 0-3, they are alerted to invalid input. Upon selecting option 1, the recommendClothingForCurrentLocation() method is invoked. This method, based on the IP address (location), determines the weather for the current location and suggests appropriate clothing. If option 2 is chosen, the user is prompted for the destination airport and arrival date. The program verifies the validity of the input (IATA airport code and date in the format YYYY-MM-DD not more than 10 days from today). Using this information, it retrieves the weather at that location and recommends suitable clothing. Option 3 exits the loop and closes the program.
DefaultNetworkFetcher - This class contains 3 methods. For opening a stream - openStream(), for configuring the connection (setting any timeouts and request methods) - configureConnection(), and for handling the response - handleResponse().
IPFetcher - This class contains a single method - getPublicIP(), which returns the public IP address.
BackupIPFetcher - This class also contains a single method - getPublicIP(), which returns the public IP address. However, it utilizes a call to another API.

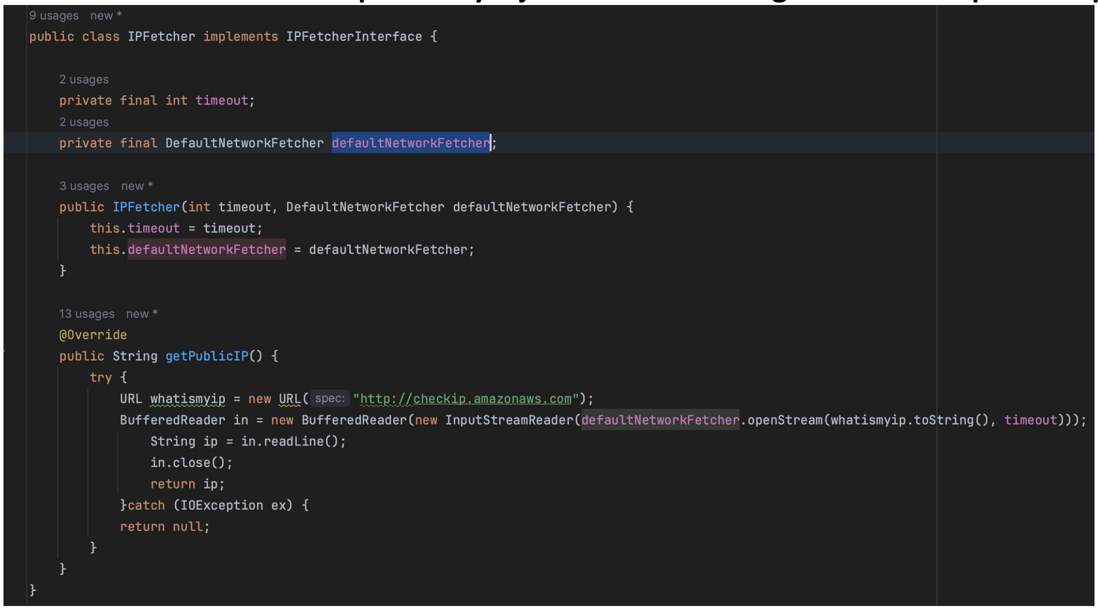
IPFetcherInterface - This interface is implemented by the classes IPFetcher and BackupIPFetcher. The interface includes the method getPublicIP().
NetworkFetcher - Interface for testing purposes.
UrlWrapper - A class for encapsulating the URL class. The URL class couldn't be easily mocked, so it was necessary to create a wrapper class.
DefaultExternalApiFetcher - contains a single method, callWeatherApi(), which returns an HttpResponse<JsonNode> response from the API based on either the IP address and date or the IATA airport code and arrival date.
ExternalApiFetcher - Interface for testing purposes.
WeatherForecast - contains two methods: getWeatherForToday() - returns today's weather based on the public IP (location) and the current date. If one API for retrieving the public IP does not respond within 3 seconds, the backup API is used.
getForecast() - based on the provided IATA airport code and arrival date, retrieves the weather forecast.

The program also includes 5 testing methods:
BackupIPFetcherTest
CLITest
DefaultNetworkFetcherTest
IPFetcherTest
WeatherForecastTest

Testing

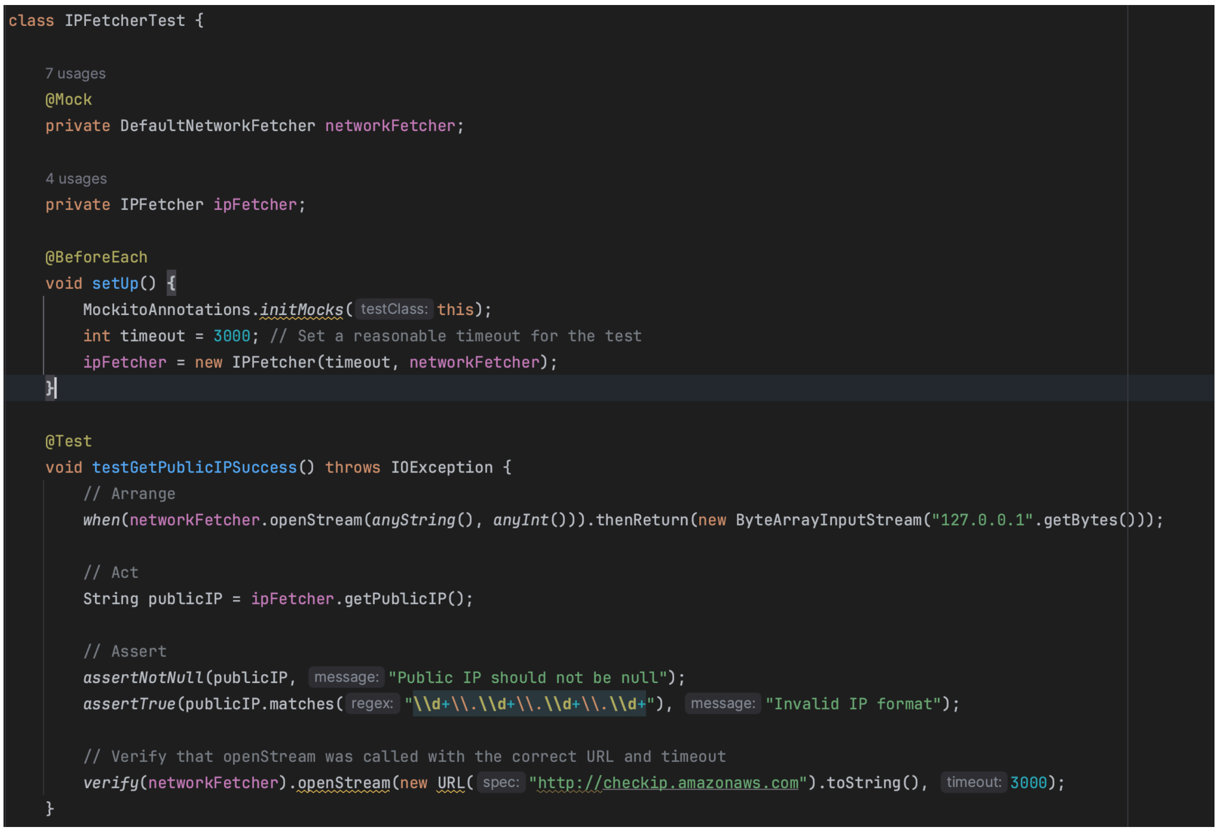
All testing has been attempted using dependency injection and leveraging mocking. This principle is utilized by all testing classes and the majority of the created methods within them.
I chose this approach because, in my opinion, it is the simplest and most clear.

-Let's demonstrate how dependency injection and mocking work in a simple example:



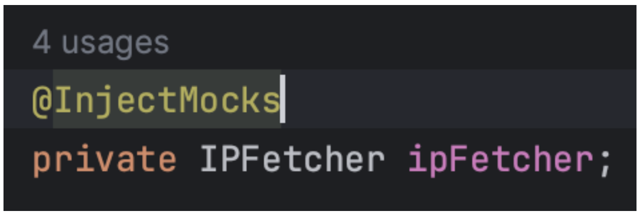
```
9 usages   new *
public class IPFetcher implements IPFetcherInterface {

    2 usages
    private final int timeout;
    2 usages
    private final DefaultNetworkFetcher defaultNetworkFetcher;

    3 usages   new *
    public IPFetcher(int timeout, DefaultNetworkFetcher defaultNetworkFetcher) {
        this.timeout = timeout;
        this.defaultNetworkFetcher = defaultNetworkFetcher;
    }

    13 usages   new *
    @Override
    public String getPublicIP() {
        try {
            URL whatismyip = new URL( spec: "http://checkip.amazonaws.com");
            BufferedReader in = new BufferedReader(new InputStreamReader(defaultNetworkFetcher.openStream(whatismyip.toString(), timeout)));
                String ip = in.readLine();
                in.close();
                return ip;
            }catch (IOException ex) {
            return null;
        }
    }
}
```

This is the class we will be testing. Note that in the constructor, I have two parameters defined: timeout and the DefaultNetworkFetcher class. The DefaultNetworkFetcher class is crucial for us, as we will be able to mock this class to simulate its behavior. Here, we are utilizing dependency injection.



```java
class IPFetcherTest {

    7 usages
    @Mock
    private DefaultNetworkFetcher networkFetcher;

    4 usages
    private IPFetcher ipFetcher;

    @BeforeEach
    void setUp() {
        MockitoAnnotations.initMocks( testClass: this);
        int timeout = 3000; // Set a reasonable timeout for the test
        ipFetcher = new IPFetcher(timeout, networkFetcher);
    }

    @Test
    void testGetPublicIPSuccess() throws IOException {
        // Arrange
        when(networkFetcher.openStream(anyString(), anyInt())).thenReturn(new ByteArrayInputStream("127.0.0.1".getBytes()));

        // Act
        String publicIP = ipFetcher.getPublicIP();

        // Assert
        assertNotNull(publicIP,  message: "Public IP should not be null");
        assertTrue(publicIP.matches( regex: "\\d+\\.\\d+\\.\\d+\\.\\d+"),  message: "Invalid IP format");

        // Verify that openStream was called with the correct URL and timeout
        verify(networkFetcher).openStream(new URL( spec: "http://checkip.amazonaws.com").toString(),  timeout: 3000);
    }
```

This is the testing class. First, we will mock the DefaultNetworkFetcher class so that we can later simulate its behavior. Then, we will create the IPFetcher class itself, into which we will inject the mocked DefaultNetworkFetcher class using the constructor.



```java
4 usages
@InjectMocks
private IPFetcher ipFetcher;
```

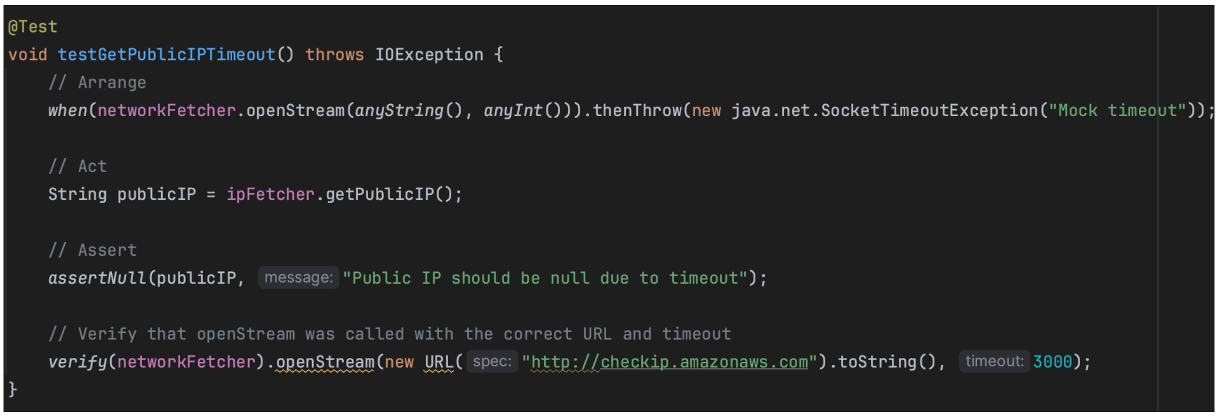
The same goal can be achieved using this approach as well.

In the body of the testing method, we can use when.thenReturn to simulate the behavior of the mocked class. In our case, we are returning the public IP.

Subsequently, by calling the tested method, we achieve the simulated behavior.

At the end, we verify whether we actually obtained the public IP.

Furthermore, using the verify method, we can verify whether certain methods were actually called.

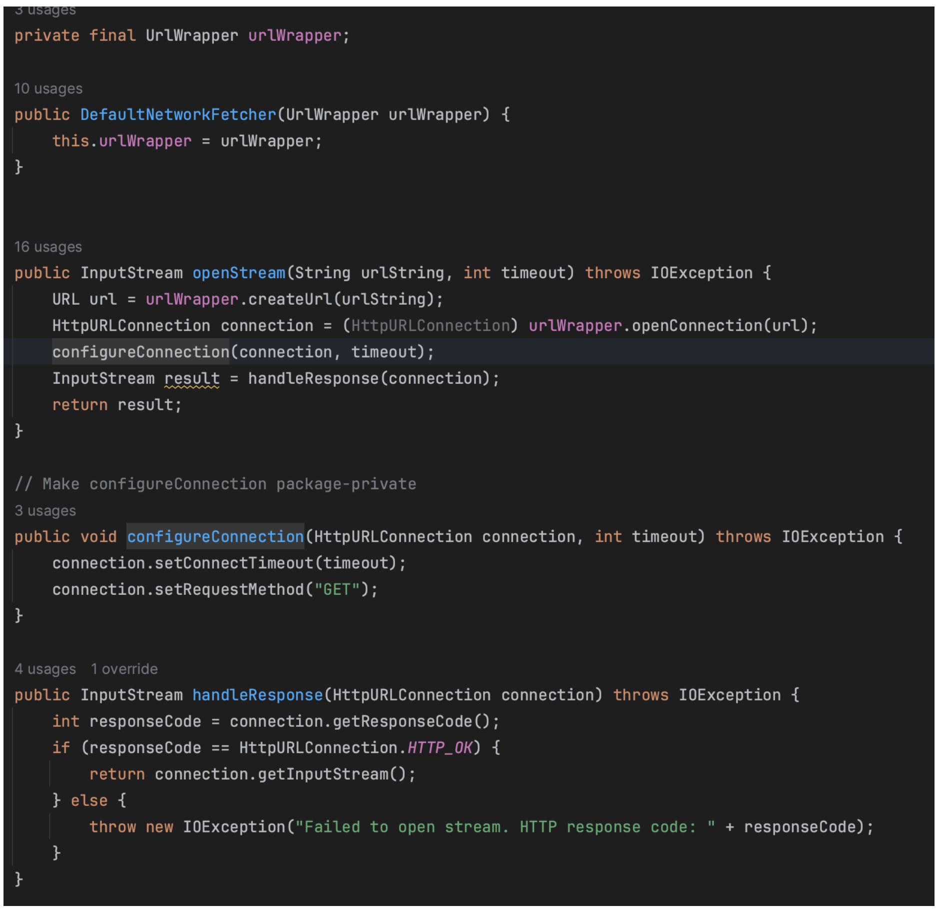


```java
@Test
void testGetPublicIPTimeout() throws IOException {
    // Arrange
    when(networkFetcher.openStream(anyString(), anyInt())).thenThrow(new java.net.SocketTimeoutException("Mock timeout"));

    // Act
    String publicIP = ipFetcher.getPublicIP();

    // Assert
    assertNull(publicIP,  message: "Public IP should be null due to timeout");

    // Verify that openStream was called with the correct URL and timeout
    verify(networkFetcher).openStream(new URL( spec: "http://checkip.amazonaws.com").toString(),  timeout: 3000);
}
```

In the same way, we can use the when.thenThrow method to throw exceptions, simulating erroneous behavior of the program.

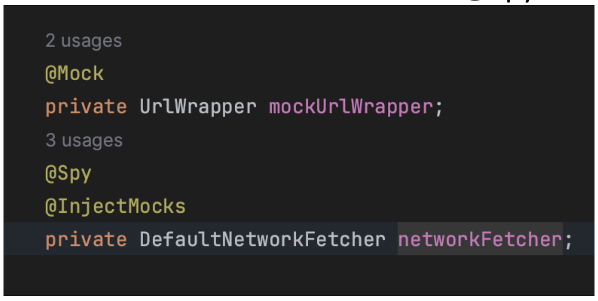
The classes BackupIPFetcher and WeatherForecast are tested in the same manner.

Furthermore, we have this class:



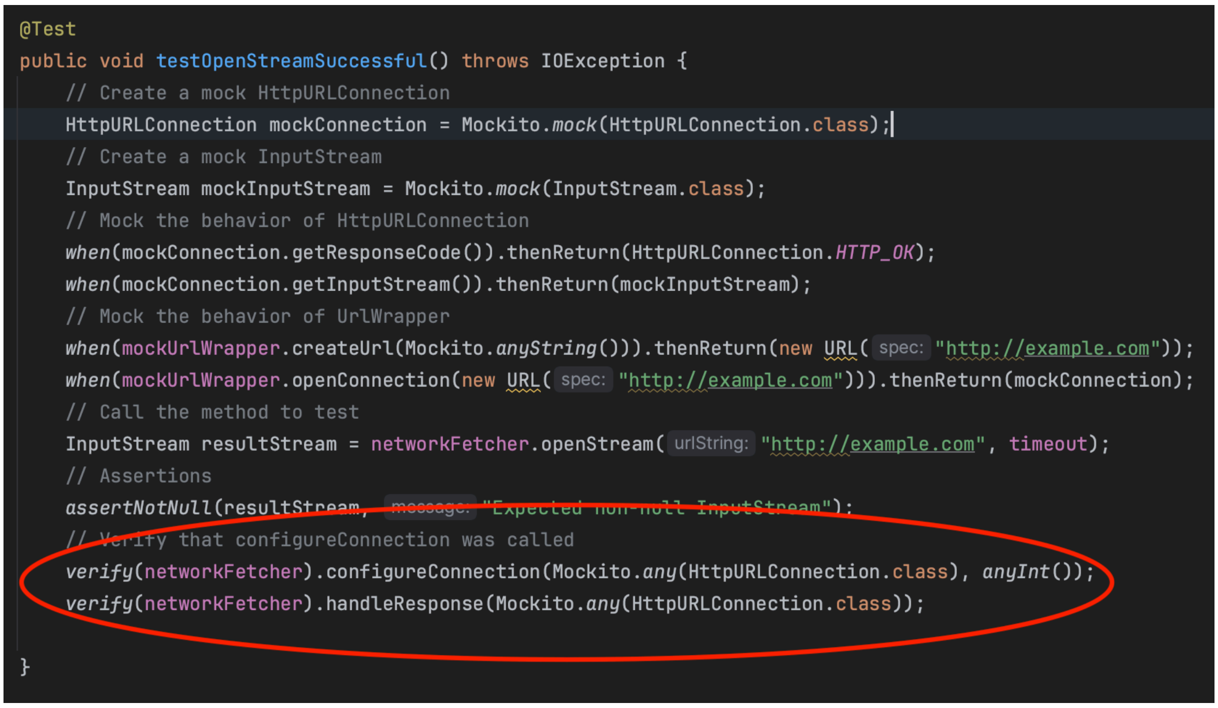
```java
3 usages
private final UrlWrapper urlWrapper;

10 usages
public DefaultNetworkFetcher(UrlWrapper urlWrapper) {
    this.urlWrapper = urlWrapper;
}


16 usages
public InputStream openStream(String urlString, int timeout) throws IOException {
    URL url = urlWrapper.createUrl(urlString);
    HttpURLConnection connection = (HttpURLConnection) urlWrapper.openConnection(url);
    configureConnection(connection, timeout);
    InputStream result = handleResponse(connection);
    return result;
}

// Make configureConnection package-private
3 usages
public void configureConnection(HttpURLConnection connection, int timeout) throws IOException {
    connection.setConnectTimeout(timeout);
    connection.setRequestMethod("GET");
}

4 usages   1 override
public InputStream handleResponse(HttpURLConnection connection) throws IOException {
    int responseCode = connection.getResponseCode();
    if (responseCode == HttpURLConnection.HTTP_OK) {
        return connection.getInputStream();
    } else {
        throw new IOException("Failed to open stream. HTTP response code: " + responseCode);
    }
}
```

There we can use Mockito feature: @Spy



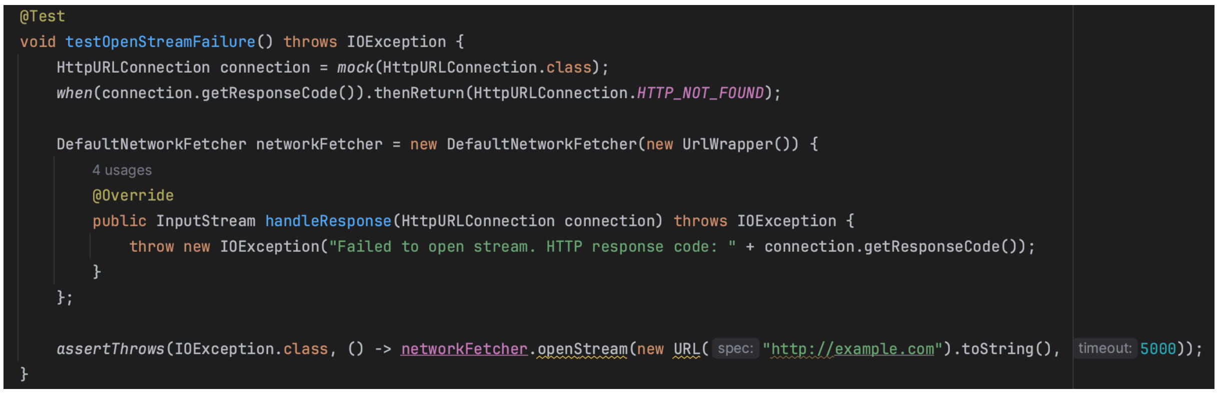
```
2 usages
@Mock
private UrlWrapper mockUrlWrapper;
3 usages
@Spy
@InjectMocks
private DefaultNetworkFetcher networkFetcher;
```

A spy is a partial mock. It allows you to mock specific methods of a real object while invoking the real methods for the rest.
In this case, @Spy is applied to the DefaultNetworkFetcher instance (networkFetcher). It means that you want to create a spy on the real DefaultNetworkFetcher object, and Mockito will track the interactions with both the real methods and the methods that you might mock.



```
@Test
public void testOpenStreamSuccessful() throws IOException {
    // Create a mock HttpURLConnection
    HttpURLConnection mockConnection = Mockito.mock(HttpURLConnection.class);
    // Create a mock InputStream
    InputStream mockInputStream = Mockito.mock(InputStream.class);
    // Mock the behavior of HttpURLConnection
    when(mockConnection.getResponseCode()).thenReturn(HttpURLConnection.HTTP_OK);
    when(mockConnection.getInputStream()).thenReturn(mockInputStream);
    // Mock the behavior of UrlWrapper
    when(mockUrlWrapper.createUrl(Mockito.anyString())).thenReturn(new URL( spec: "http://example.com"));
    when(mockUrlWrapper.openConnection(new URL( spec: "http://example.com"))).thenReturn(mockConnection);
    // Call the method to test
    InputStream resultStream = networkFetcher.openStream( urlString: "http://example.com", timeout);
    // Assertions
    assertNotNull(resultStream, message: "Expected non-null InputStream");
    // Verify that configureConnection was called
    verify(networkFetcher).configureConnection(Mockito.any(HttpURLConnection.class), anyInt());
    verify(networkFetcher).handleResponse(Mockito.any(HttpURLConnection.class));

}
```

This approach allows us to track, using the verify method, whether real methods were called. In case we didn't use @Spy, we would only be able to verify the invocation of mocked methods.



```java
@Test
void testOpenStreamFailure() throws IOException {
    HttpURLConnection connection = mock(HttpURLConnection.class);
    when(connection.getResponseCode()).thenReturn(HttpURLConnection.HTTP_NOT_FOUND);

    DefaultNetworkFetcher networkFetcher = new DefaultNetworkFetcher(new UrlWrapper()) {
        4 usages
        @Override
        public InputStream handleResponse(HttpURLConnection connection) throws IOException {
            throw new IOException("Failed to open stream. HTTP response code: " + connection.getResponseCode());
        }
    };

    assertThrows(IOException.class, () -> networkFetcher.openStream(new URL( spec: "http://example.com").toString(),  timeout: 5000));
}
```

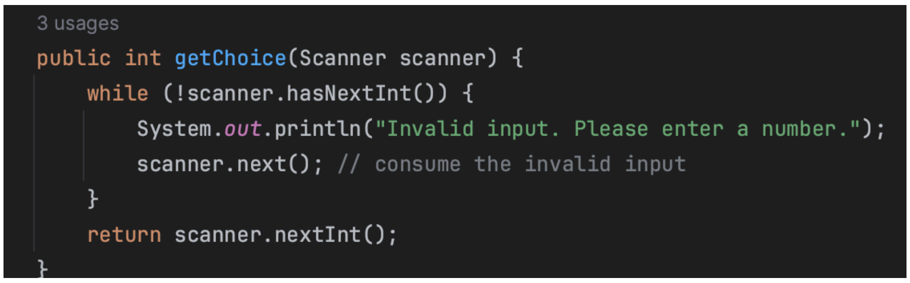
Furthermore, we can use @Override to override methods to achieve a specific requirement.

In this case, we are altering the behavior of the handleResponse method, which now returns an exception. This allows us to simulate a connection rejection.

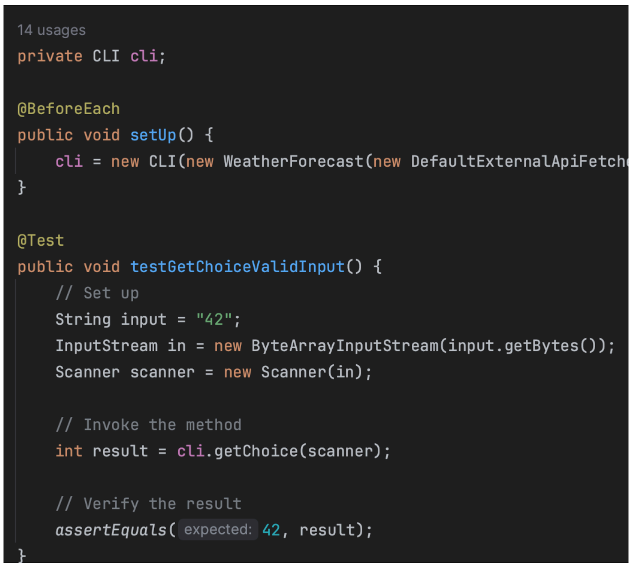
furthermore, in the testing class CLITest, I utilized testing with dummy objects.

Dummy objects are typically used when an object is required as a parameter for a method, but the actual behavior of that object is not important for the test scenario. The dummy object is used to fulfill the method's signature without introducing unnecessary complexity or dependencies.

Here's an example of how a dummy object might be used in a Java test:

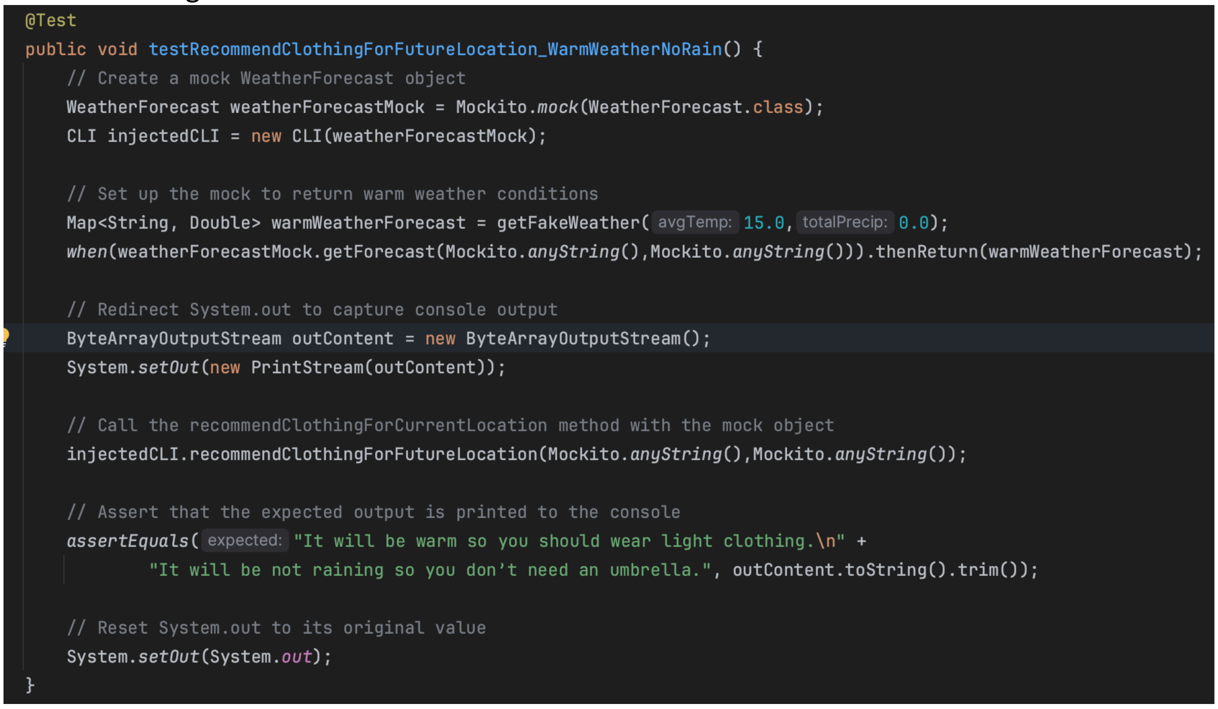


```java
3 usages
public int getChoice(Scanner scanner) {
    while (!scanner.hasNextInt()) {
        System.out.println("Invalid input. Please enter a number.");
        scanner.next(); // consume the invalid input
    }
    return scanner.nextInt();
}
```



```java
14 usages
private CLI cli;

@BeforeEach
public void setUp() {
    cli = new CLI(new WeatherForecast(new DefaultExternalApiFetch
}

@Test
public void testGetChoiceValidInput() {
    // Set up
    String input = "42";
    InputStream in = new ByteArrayInputStream(input.getBytes());
    Scanner scanner = new Scanner(in);

    // Invoke the method
    int result = cli.getChoice(scanner);

    // Verify the result
    assertEquals( expected: 42, result);
}
```

The last testing method I'd like to discuss is this one:

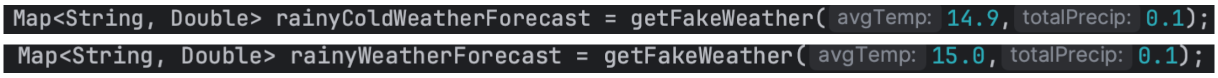


```java
@Test
public void testRecommendClothingForFutureLocation_WarmWeatherNoRain() {
    // Create a mock WeatherForecast object
    WeatherForecast weatherForecastMock = Mockito.mock(WeatherForecast.class);
    CLI injectedCLI = new CLI(weatherForecastMock);

    // Set up the mock to return warm weather conditions
    Map<String, Double> warmWeatherForecast = getFakeWeather( avgTemp: 15.0, totalPrecip: 0.0);
    when(weatherForecastMock.getForecast(Mockito.anyString(),Mockito.anyString())).thenReturn(warmWeatherForecast);

    // Redirect System.out to capture console output
    ByteArrayOutputStream outContent = new ByteArrayOutputStream();
    System.setOut(new PrintStream(outContent));

    // Call the recommendClothingForCurrentLocation method with the mock object
    injectedCLI.recommendClothingForFutureLocation(Mockito.anyString(),Mockito.anyString());

    // Assert that the expected output is printed to the console
    assertEquals( expected: "It will be warm so you should wear light clothing.\n" +
            "It will be not raining so you don't need an umbrella.", outContent.toString().trim());

    // Reset System.out to its original value
    System.setOut(System.out);
}
```

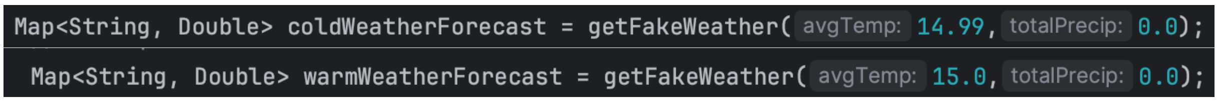
Here, we are again using mocking. However, it's important to mention that we are testing a method that, based on the temperature, determines what clothing to wear and, based on the precipitation, decides whether to take an umbrella. The threshold values are <15 degrees Celsius - cold, >=15 degrees Celsius - warm. For precipitation, 0.1mm indicates rain, while 0.0mm means no rain.

It's crucial to note that it's important to test values as close as possible to the extreme values. Often, programmers may inadvertently use, for example, > instead of >=, and similar mistakes, so testing as close as possible to the extreme values is necessary.

Our testing methods, therefore, make use of these values:



```java
Map<String, Double> rainyColdWeatherForecast = getFakeWeather( avgTemp: 14.9, totalPrecip: 0.1);
Map<String, Double> rainyWeatherForecast = getFakeWeather( avgTemp: 15.0, totalPrecip: 0.1);
```

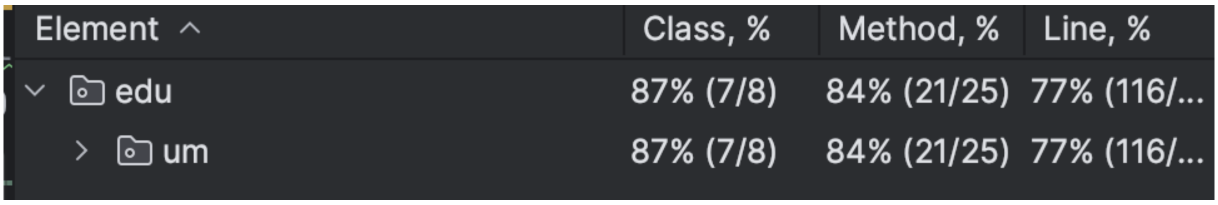


```java
Map<String, Double> coldWeatherForecast = getFakeWeather( avgTemp: 14.99, totalPrecip: 0.0);
Map<String, Double> warmWeatherForecast = getFakeWeather( avgTemp: 15.0, totalPrecip: 0.0);
```

It's also important to mention that in my testing method, I'm checking whether the console output equals a specific string. This approach can be debatable. If the programmer changes the message, the tests will immediately report an error. Instead, it would be more appropriate to return, for example, numerical values that represent certain states. However, for the purpose of our program, this approach may be sufficient.

Test Coverage

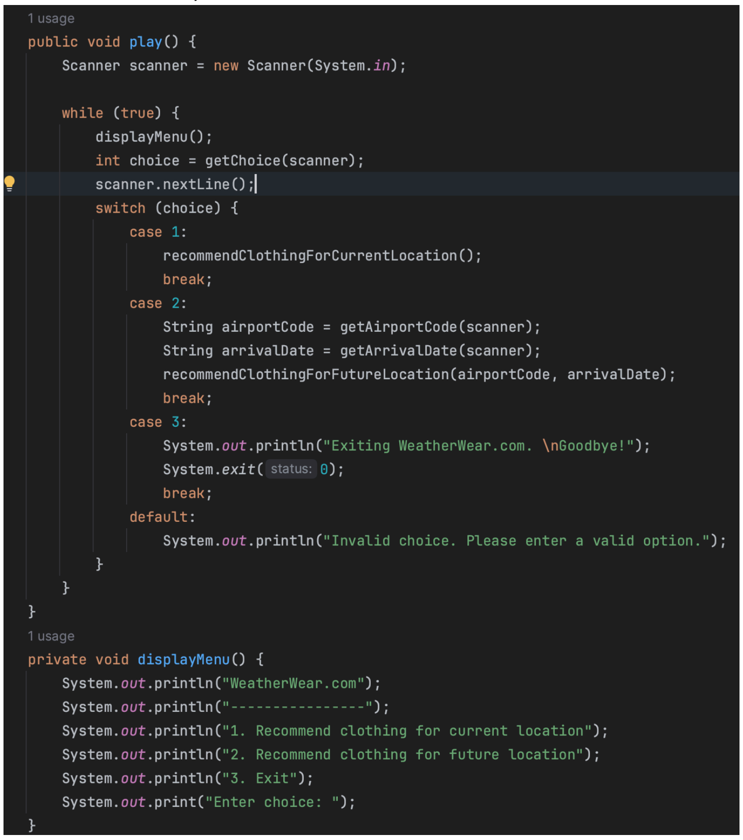
In my case I achieved 77% test coverage



Element	Class, %	Method, %	Line, %
edu	87% (7/8)	84% (21/25)	77% (116/...
um	87% (7/8)	84% (21/25)	77% (116/...

I haven't tested the main method, which only includes creating an instance of the CLI class and calling the cli.play() method.

In the CLI class, I haven't tested these two methods:

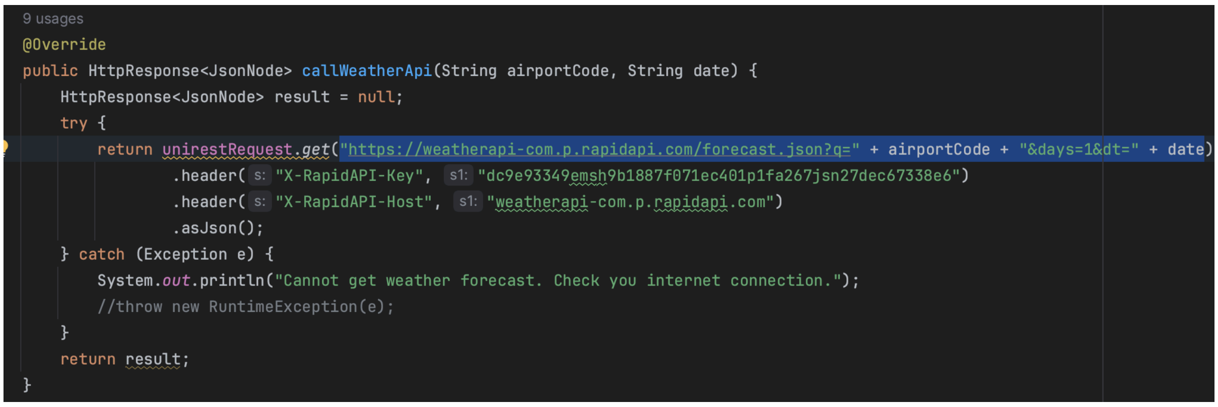


```
1 usage
public void play() {
    Scanner scanner = new Scanner(System.in);

    while (true) {
        displayMenu();
        int choice = getChoice(scanner);
        scanner.nextLine();
        switch (choice) {
            case 1:
                recommendClothingForCurrentLocation();
                break;
            case 2:
                String airportCode = getAirportCode(scanner);
                String arrivalDate = getArrivalDate(scanner);
                recommendClothingForFutureLocation(airportCode, arrivalDate);
                break;
            case 3:
                System.out.println("Exiting WeatherWear.com. \nGoodbye!");
                System.exit( status: 0);
                break;
            default:
                System.out.println("Invalid choice. Please enter a valid option.");
        }
    }
}
1 usage
private void displayMenu() {
    System.out.println("WeatherWear.com");
    System.out.println("---------------");
    System.out.println("1. Recommend clothing for current location");
    System.out.println("2. Recommend clothing for future location");
    System.out.println("3. Exit");
    System.out.print("Enter choice: ");
}
```

Testing the displayMenu() method doesn't make sense due to its simplicity. The method doesn't perform any complex operations. It simply prints text to the console, which is visible every time it's executed.

Here, testing the play() method is again debatable. It contains an "infinite" loop, making its testing more complex. However, the method could be divided, and different parts could be tested separately. Due to its simplicity and the fact that its functionality is mostly just calling other methods(that are already tested), I've decided not to test it as well.

I haven't tested this method either:



```
9 usages
@Override
public HttpResponse<JsonNode> callWeatherApi(String airportCode, String date) {
    HttpResponse<JsonNode> result = null;
    try {
        return unirestRequest.get("https://weatherapi-com.p.rapidapi.com/forecast.json?q=" + airportCode + "&days=1&dt=" + date)
                .header( s: "X-RapidAPI-Key",  s1: "dc9e93349emsh9b1887f071ec401p1fa267jsn27dec67338e6")
                .header( s: "X-RapidAPI-Host",  s1: "weatherapi-com.p.rapidapi.com")
                .asJson();
    } catch (Exception e) {
        System.out.println("Cannot get weather forecast. Check you internet connection.");
        //throw new RuntimeException(e);
    }
    return result;
}
```

Testing this method probably would be appropriate. Unfortunately, I encountered issues with mocking the Unirest data type, where the when().thenReturn() part consistently threw an error. Even after many hours of debugging, I couldn't resolve this issue.

Link to the GitHub repository with the program:

https://github.com/JanSkacel01/WeatherService