Fundamentals of Software Testing

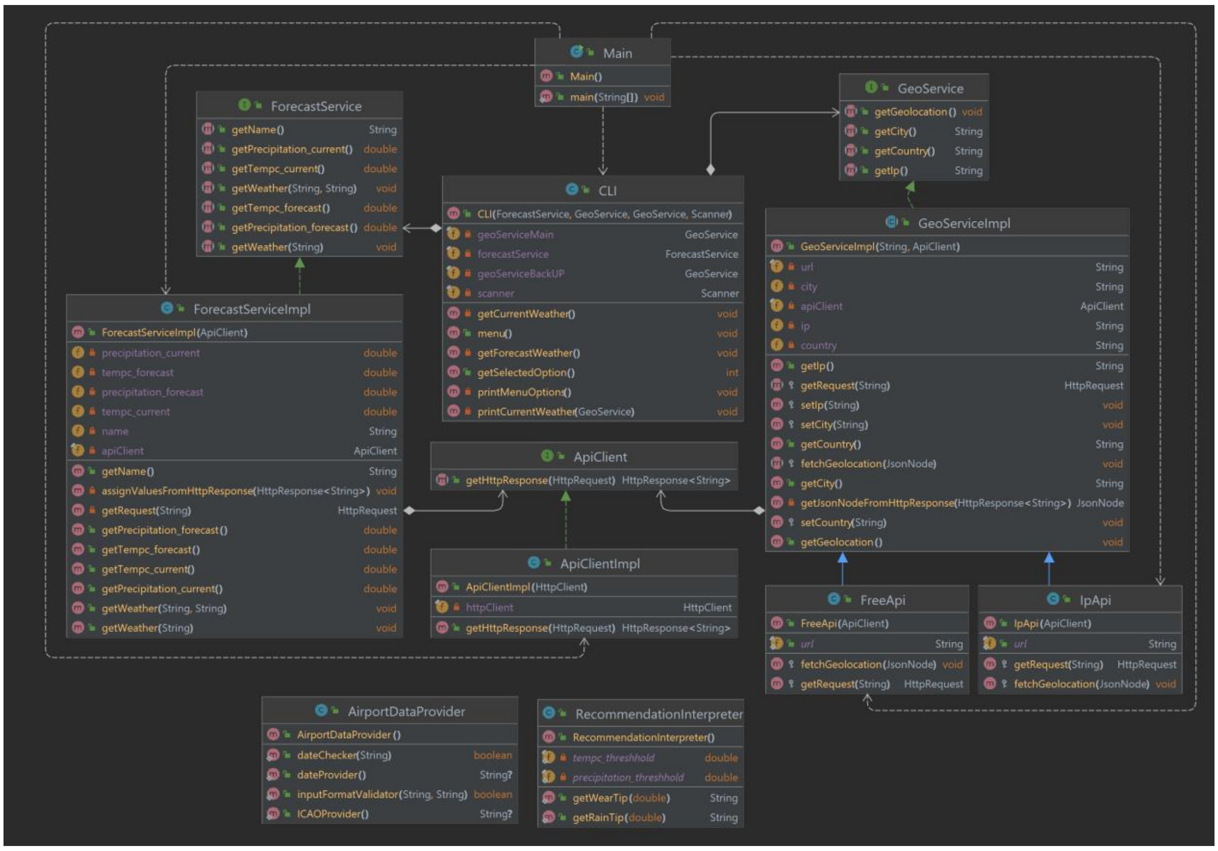
Assignment Part 1 (of 3) – Unit Testing

Otmar Nezdařil – 2312649

CODE: HTTPS://GITHUB.COM/OTHEEC/WEATHERWEAR.COM

Construction of functional system

Used classes are depicted of on class diagram attached bellow.


Main
Main()
main(String[]) void
ForecastService
getName() String
getPrecipitation_current() double
getTempc_current() double
getWeather(String, String) void
getTempc_forecast() double
getPrecipitation_forecast() double
getWeather(String) void
GeoService
getGeolocation() void
getCity() String
getCountry() String
getIp() String
CLI
CLI(ForecastService, GeoService, GeoService, Scanner)
geoServiceMain GeoService
forecastService ForecastService
geoServiceBackUP GeoService
scanner Scanner
getCurrentWeather() void
menu() void
getForecastWeather() void
getSelectedOption() int
printMenuOptions() void
printCurrentWeather(GeoService) void
GeoServiceImpl
GeoServiceImpl(String, ApiClient)
url String
city String
apiClient ApiClient
ip String
country String
getIp() String
getRequest(String) HttpRequest
setIp(String) void
setCity(String) void
getCountry() String
fetchGeolocation(JsonNode) void
getCity() String
getJsonNodeFromHttpResponse(HttpResponse<String>) JsonNode
setCountry(String) void
getGeolocation() void
ForecastServiceImpl
ForecastServiceImpl(ApiClient)
precipitation_current double
tempc_forecast double
precipitation_forecast double
tempc_current double
name String
apiClient ApiClient
getName() String
assignValuesFromHttpResponse(HttpResponse<String>) void
getRequest(String) HttpRequest
getPrecipitation_forecast() double
getTempc_forecast() double
getTempc_current() double
getPrecipitation_current() double
getWeather(String, String) void
getWeather(String) void
ApiClient
getHttpResponse(HttpRequest) HttpResponse<String>
ApiClientImpl
ApiClientImpl(HttpClient)
httpClient HttpClient
getHttpResponse(HttpRequest) HttpResponse<String>
FreeApi
FreeApi(ApiClient)
url String
fetchGeolocation(JsonNode) void
getRequest(String) HttpRequest
IpApi
IpApi(ApiClient)
url String
getRequest(String) HttpRequest
fetchGeolocation(JsonNode) void
AirportDataProvider
AirportDataProvider()
dateChecker(String) boolean
dateProvider() String?
inputFormatValidator(String, String) boolean
ICAOProvider() String?
RecommendationInterpreter
RecommendationInterpreter()
tempc_threshhold double
precipitation_threshhold double
getWearTip(double) String
getRainTip(double) String

There is a brief description of my classes:

CLI - Basic command line interface, used for communication with user. Carries functionality a of calling appropriate methods to demonstrate general functionality of the system.

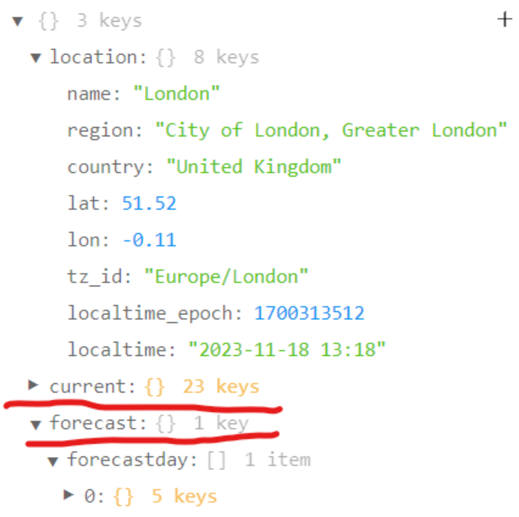
ApiClient – This class contains just one method to call API based on passed request. Method is located v separate class, to be easily accessed and called for calling weather API and geolocation APIs and avoiding redundancy of the code.

Implemented using java.net.http package, because for me it was easiest testable way of implementation.

GeoServiceImpl – This abstract class provides functionality for 2 its geolocation subclasses, where each of them is using another API, so they differ in sent requests and responded data fetching, but overall, they share most of the functionality from their base abstract class.

FreeApi and **IpApi** – each class just return Http request and fetch received response.

ForecastServiceImpl – Implementation of weather service. I have used same API to get current weather and weather forecast data, because APIs response contains both (response always contains forecast for at least 1 day regardless on parameters).



```
▼ {} 3 keys                              +
  ▼ location: {} 8 keys
      name: "London"
      region: "City of London, Greater London"
      country: "United Kingdom"
      lat: 51.52
      lon: -0.11
      tz_id: "Europe/London"
      localtime_epoch: 1700313512
      localtime: "2023-11-18 13:18"
  ▶ current: {} 23 keys
  ▼ forecast: {} 1 key
    ▼ forecastday: [] 1 item
      ▶ 0: {} 5 keys
```

AirportDateProvider – handles functionality of providing validated ICAO code and date.

ReccomendationInterpreter – contains static methods to return strings with responses to provide wear and rain recommendation tips.

Testing

All test classes are located based on default IntelliJ “generate>test” feature. Every class uses full dependency injection to be easily testable (excluding CLI, will be mentioned below). Several interfaces are also implemented, although they are not necessary, but they would find their place in case of expanding the system or using stubs/spies instead of mocking. With proper encapsulation, just only the public methods are tested, considering the (package)private methods logic is tested via the public ones.

RecommendationInterpreter

Rain participation: Testing methods are quite simple, since input variable is never null and it is always passed, I have rain participation tips based on just 2 different incomes: 0, or any value, which is not zero. Number lower than zero means there is something wrong with the weather API data (not the fetching, but actual response) so I verify find out whether are APIs data about the weather correct.

Temperature: Testing of temperature method requires one extra step – testing the limit value of the condition – 15.

AirportDataProvider

For checking date and ICAO format I have used one format checker method to which I have passed different parameters of expected format of checked string. Format checker checks also for null or an empty string.

I cannot check if provided ICAO is ICAO of real airport, but since the API for every incorrect ICAO responds with “name:"Metar"” etc. response, it does not really matter (from of point of view of stability of the system and avoiding any unhandled exception or crash).

Testing date involves one extra step, checking if the date falls within the valid range. To avoid using same code for calculating days, in the program I have used Callendar and in tests I have used LocalDate. I have also test covered exception while parsing limits of the day date. Tested values are today (start of the valid range), last possible value of the valid range (10 days into the future), before and after valid range and from the middle of the range.

ApiClientImpl

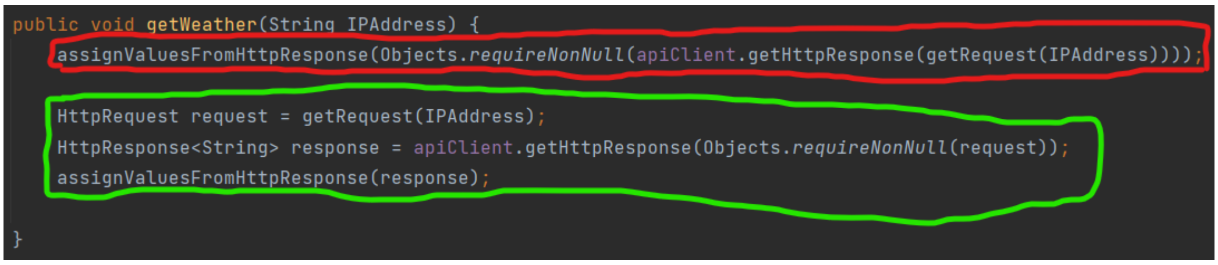
Method getHttpResponse has 4 possible outcomes:

- Returning response when status code is 200.
- Returning null while status code is not 200.
- 2 exceptions

To avoid using internet connection, I have mocked HttpClient (and injected to ApiClient) and HttpResponse. When HttpClient send request, mocked response is returned, or exception thrown. Also, when response is returned, status code is mocked.

ForecastServiceImpl

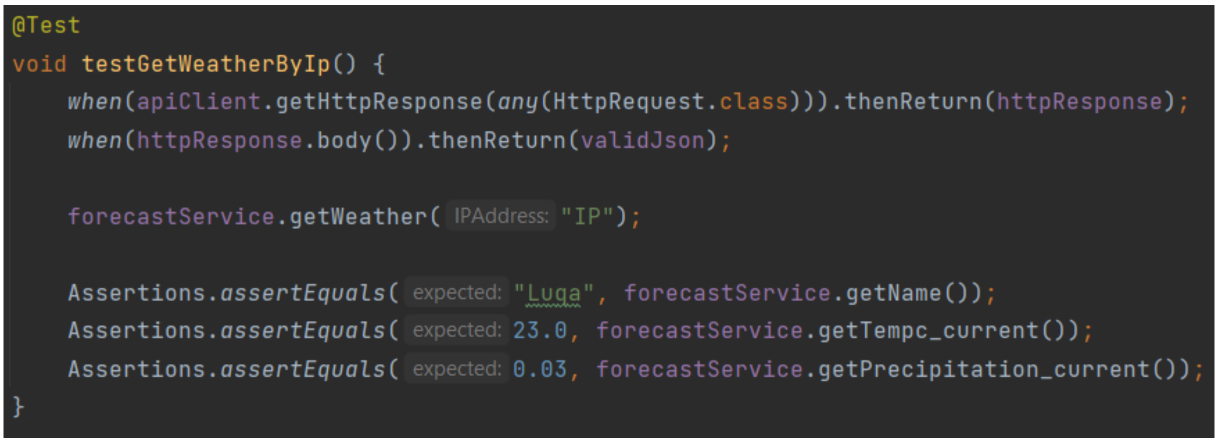
The only public testable methods are getters and overloaded method getWeather(). To better understand its logic, I will expand it to more lines to make the logic and the mocking order behind it clear.



```java
public void getWeather(String IPAddress) {
    assignValuesFromHttpResponse(Objects.requireNonNull(apiClient.getHttpResponse(getRequest(IPAddress))));

    HttpRequest request = getRequest(IPAddress);
    HttpResponse<String> response = apiClient.getHttpResponse(Objects.requireNonNull(request));
    assignValuesFromHttpResponse(response);

}
```

The red and green code fragments fulfill same functionality – red one is used in my code and the second one is expanded for its better understanding.

- The first line in green code fragments tests creating request (getRequest method).
- The second line indicates that we need to mock ApiClients behavior (and from that we know we need to mock HttpResponse)
- Third line calls private method, which calls rest of the classes' methods – getters – therefore (almost – if I don't mention the overload method with the same implemented logic) whole class is tested by one test.



```java
@Test
void testGetWeatherByIp() {
    when(apiClient.getHttpResponse(any(HttpRequest.class))).thenReturn(httpResponse);
    when(httpResponse.body()).thenReturn(validJson);

    forecastService.getWeather( IPAddress: "IP");

    Assertions.assertEquals( expected: "Luga", forecastService.getName());
    Assertions.assertEquals( expected: 23.0, forecastService.getTempc_current());
    Assertions.assertEquals( expected: 0.03, forecastService.getPrecipitation_current());
}
```

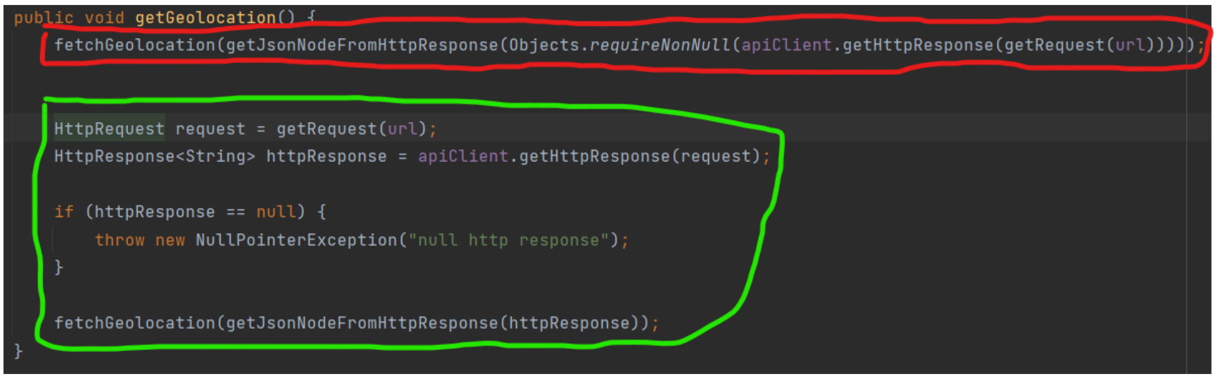
I need to mention that I know that good unit test should not use file system, but to test private methods (assignValuesFromHttpResponse and getRequest), I need to pass to the test real JSON data. It is possible to hard code it to the test (in the same way I did it in FreeApi and IpApi tests), but for future maintenance and better orientation in larger JSONs I did it this way.

By accepting any(HttpRequest.class) in apiClient.getHttpResponse, getRequest method is tested successfully.

In very similar way I test getWeather overload with metar and date. Last needed test if for testing JsonProcessingException again in getWeather method, thrown by assignValuesFromHttpResponse()).

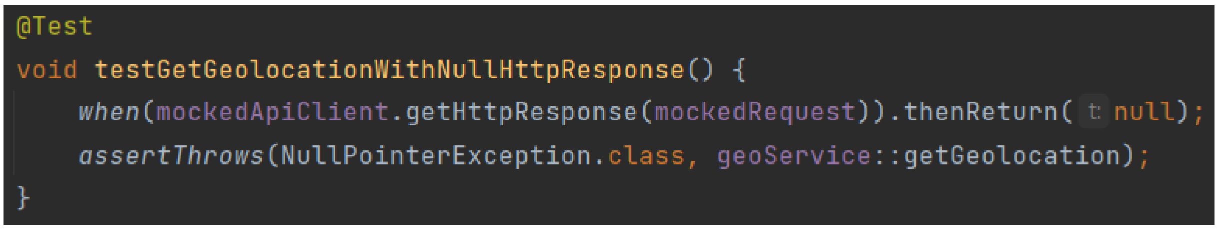
GeoServiceImpl

Again, as in last class above, here are 2 code snippets with same functionality:



```java
public void getGeolocation() {
    fetchGeolocation(getJsonNodeFromHttpResponse(Objects.requireNonNull(apiClient.getHttpResponse(getRequest(url)))));

    HttpRequest request = getRequest(url);
    HttpResponse<String> httpResponse = apiClient.getHttpResponse(request);

    if (httpResponse == null) {
        throw new NullPointerException("null http response");
    }

    fetchGeolocation(getJsonNodeFromHttpResponse(httpResponse));
}
```

Since it is an abstract class, I need to implement all methods before creating and instance to test its whole functionality. I would be able to test it from the subclasses, but in this way, I can start testing earlier during development. But as I said, I could achieve needed code coverage even from subclasses.



```java
@Test
void testGetGeolocationWithNullHttpResponse() {
    when(mockedApiClient.getHttpResponse(mockedRequest)).thenReturn( t: null);
    assertThrows(NullPointerException.class, geoService::getGeolocation);
}
```

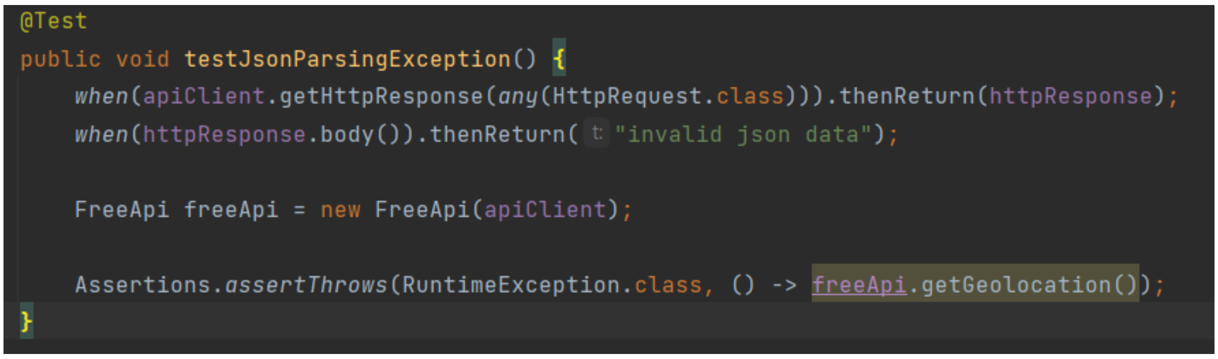
This test covers the exception thrown in the green highlighted code above.

FreeApi and IpApi

These two classes are different just in URL passed to the abstract base class and implementation of fetcher a request builder (one has the 3 seconds timeout). Tests differs in passed mocked response.

The test logic behind them is very simple: created request is passed to mocked ApiClient, which responds with mocked response. Request is really created and the response if fetchable etc., so the code of whole class is testable.

In one the classes I have also tested the JSON parsing error with invalid JSON.

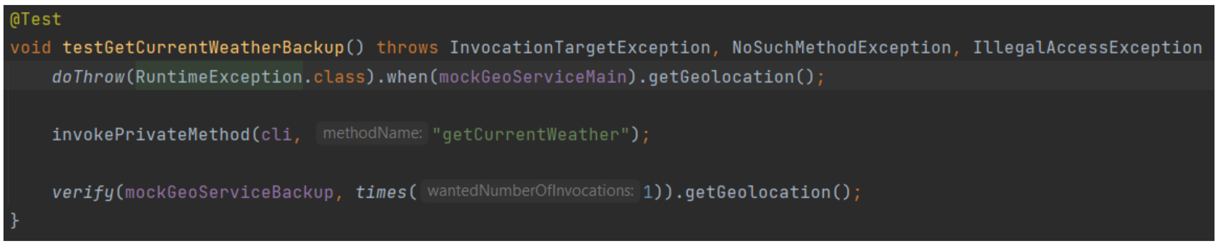


```java
@Test
public void testJsonParsingException() {
    when(apiClient.getHttpResponse(any(HttpRequest.class))).thenReturn(httpResponse);
    when(httpResponse.body()).thenReturn( t: "invalid json data");

    FreeApi freeApi = new FreeApi(apiClient);

    Assertions.assertThrows(RuntimeException.class, () -> freeApi.getGeolocation());
}
```

CLI

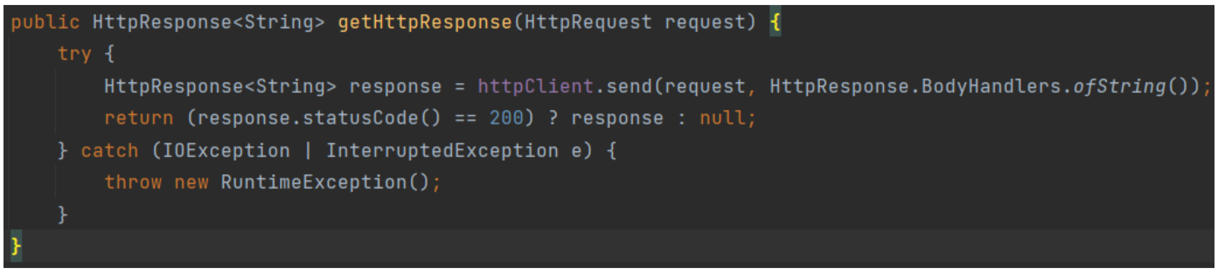
Because of the encapsulation, I copied *invokePrivateMethod* from ChatGPT (marked by comment in my code at git) to be able to call private methods. Another possible way would be implementing an interface of making methods public.

My geolocation back up is checked by this test:



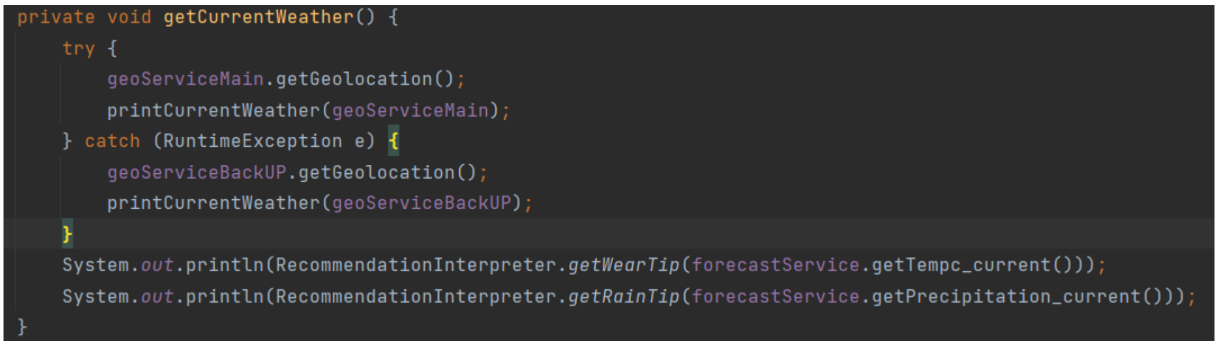
```java
@Test
void testGetCurrentWeatherBackup() throws InvocationTargetException, NoSuchMethodException, IllegalAccessException
    doThrow(RuntimeException.class).when(mockGeoServiceMain).getGeolocation();

    invokePrivateMethod(cli,  methodName: "getCurrentWeather");

    verify(mockGeoServiceBackup, times( wantedNumberOfInvocations: 1)).getGeolocation();
}
```

If request times out, one of these exceptions is caught and RunTimeException is thrown in ApiClient.



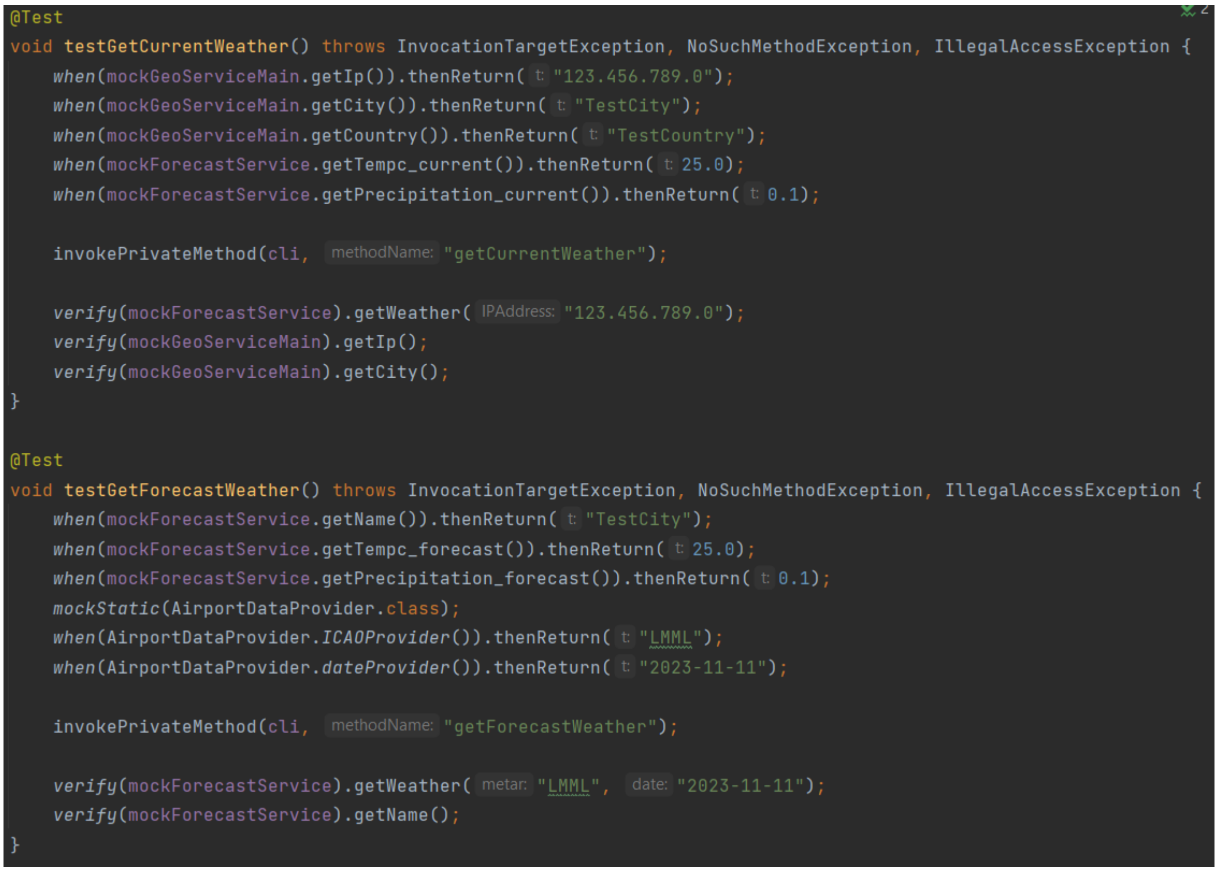
```java
public HttpResponse<String> getHttpResponse(HttpRequest request) {
    try {
        HttpResponse<String> response = httpClient.send(request, HttpResponse.BodyHandlers.ofString());
        return (response.statusCode() == 200) ? response : null;
    } catch (IOException | InterruptedException e) {
        throw new RuntimeException();
    }
}
```

In this try and catch block, second geolocation API is called, while exception is caught.



```java
private void getCurrentWeather() {
    try {
        geoServiceMain.getGeolocation();
        printCurrentWeather(geoServiceMain);
    } catch (RuntimeException e) {
        geoServiceBackUP.getGeolocation();
        printCurrentWeather(geoServiceBackUP);
    }
    System.out.println(RecommendationInterpreter.getWearTip(forecastService.getTempc_current()));
    System.out.println(RecommendationInterpreter.getRainTip(forecastService.getPrecipitation_current()));
}
```

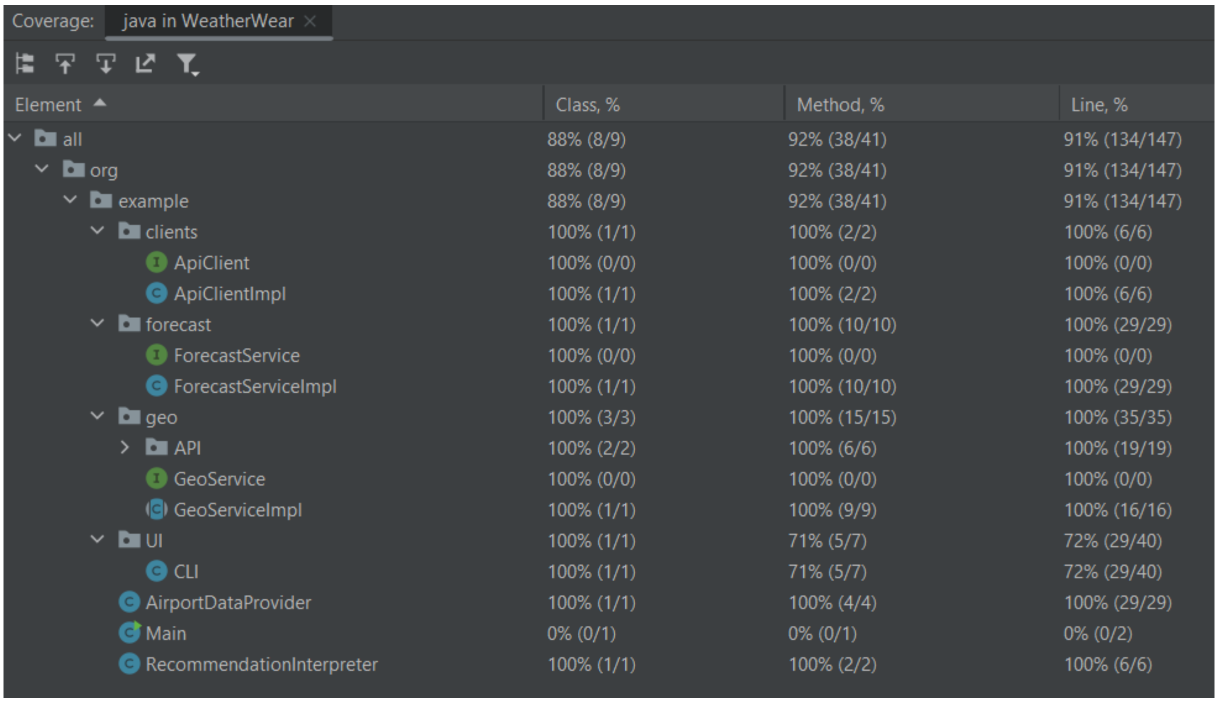
My remaining test in CLI just verify, if appropriate methods were called and if the CLI whole connection of geolocation, weather (/tips) and data providers work properly.



```
@Test
void testGetCurrentWeather() throws InvocationTargetException, NoSuchMethodException, IllegalAccessException {
    when(mockGeoServiceMain.getIp()).thenReturn( t: "123.456.789.0");
    when(mockGeoServiceMain.getCity()).thenReturn( t: "TestCity");
    when(mockGeoServiceMain.getCountry()).thenReturn( t: "TestCountry");
    when(mockForecastService.getTempc_current()).thenReturn( t: 25.0);
    when(mockForecastService.getPrecipitation_current()).thenReturn( t: 0.1);

    invokePrivateMethod(cli,  methodName: "getCurrentWeather");

    verify(mockForecastService).getWeather( IPAddress: "123.456.789.0");
    verify(mockGeoServiceMain).getIp();
    verify(mockGeoServiceMain).getCity();
}

@Test
void testGetForecastWeather() throws InvocationTargetException, NoSuchMethodException, IllegalAccessException {
    when(mockForecastService.getName()).thenReturn( t: "TestCity");
    when(mockForecastService.getTempc_forecast()).thenReturn( t: 25.0);
    when(mockForecastService.getPrecipitation_forecast()).thenReturn( t: 0.1);
    mockStatic(AirportDataProvider.class);
    when(AirportDataProvider.ICAOProvider()).thenReturn( t: "LMML");
    when(AirportDataProvider.dateProvider()).thenReturn( t: "2023-11-11");

    invokePrivateMethod(cli,  methodName: "getForecastWeather");

    verify(mockForecastService).getWeather( metar: "LMML",  date: "2023-11-11");
    verify(mockForecastService).getName();
}
```

Code coverage

I have achieved code line coverage 91%. This code is 100% coverage testable, but the tests would not make any further sense (testing methods with just System.out.prinln() or testing main method). So only not 100% coverage classes are located in CLI and main.



Element	Class, %	Method, %	Line, %
all	88% (8/9)	92% (38/41)	91% (134/147)
org	88% (8/9)	92% (38/41)	91% (134/147)
example	88% (8/9)	92% (38/41)	91% (134/147)
clients	100% (1/1)	100% (2/2)	100% (6/6)
ApiClient	100% (0/0)	100% (0/0)	100% (0/0)
ApiClientImpl	100% (1/1)	100% (2/2)	100% (6/6)
forecast	100% (1/1)	100% (10/10)	100% (29/29)
ForecastService	100% (0/0)	100% (0/0)	100% (0/0)
ForecastServiceImpl	100% (1/1)	100% (10/10)	100% (29/29)
geo	100% (3/3)	100% (15/15)	100% (35/35)
API	100% (2/2)	100% (6/6)	100% (19/19)
GeoService	100% (0/0)	100% (0/0)	100% (0/0)
GeoServiceImpl	100% (1/1)	100% (9/9)	100% (16/16)
UI	100% (1/1)	71% (5/7)	72% (29/40)
CLI	100% (1/1)	71% (5/7)	72% (29/40)
AirportDataProvider	100% (1/1)	100% (4/4)	100% (29/29)
Main	0% (0/1)	0% (0/1)	0% (0/2)
RecommendationInterpreter	100% (1/1)	100% (2/2)	100% (6/6)