

Fundamentals of Software Testing
Unit Testing of proof of concept WeatheWear.com
Assignment submitted in partial fulfillment of CPS3230

Andre' Vella

November 17, 2023

Contents

1 Technologies used	2
1.1 Setting up api key	2
2 System Design	3
2.1 UML Diagram	3
3 Unit Testing Approach and Design	4
3.1 Naming convention	4
3.2 Test Design: Equivalence classes and boundary value analysis	4
3.3 Testing of direct I/O	5
3.3.1 Testing the menu	6
3.4 Testing of indirect I/O	7
3.4.1 Implementing an interface	7
3.4.2 Constructor Injection	7
3.4.3 Mocking	7
3.5 Error handling	9
3.5.1 Expected Exception Testing	9
3.6 Saboteur Pattern: Call backup location service if service fails	10
4 Code Coverage	12
4.1 Report	12

Section 1

Technologies used



Project Repo: <https://github.com/andimon/WeatherWear.com>

- Language: Java
- Testing framework: JUNIT 5
- Mocking framework: MOCKITO
- Rest Client: Jakarta RESTful Web Services
- Get Location API: <https://ip-api.com/>
- Location Backup API: <https://ipapi.co/>
- Get Location From IATA API: <https://iatageo.com/>
- Get Location From IATA BACKUP API: <https://rapidapi.com/Active-api/api/airport-info>
- Get Weather API: <https://open-meteo.com/>

1.1 Setting up api key

The backup service for obtaining a location from IATA is a RAPIDAI service hence requiring an api key. One can sign up here (<https://rapidapi.com/Active-api/api/airport-info>) to obtain an API key. The key is included as an environment variable in file `.env` that needs to be included in the backend folder.

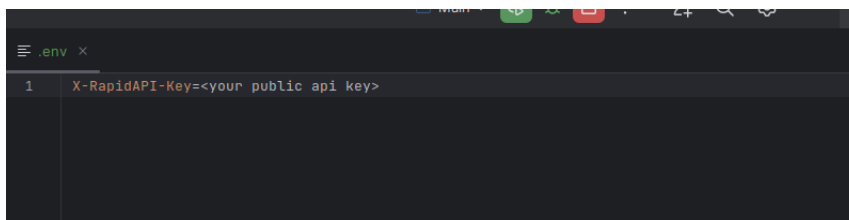


Figure 1.1: API key

Section 2

System Design

Keeping organisation of future tests in mind. The classes are designed to follow the **Single Responsibility Principle**. The effort is made for so while the separate test classes are testing for different classes they are also testing different

Furthermore, having loosely coupled subsystems should render the system more testable allowing us to obtain a high coverage by proper use of unit testing techniques.

2.1 UML Diagram

A UML for the entire project can be accessed from the following URL (*this may be outdated if the codebase will be updated in the future but it should capture the core structure of the concept nonetheless*) <https://github.com/andimon/WeatherWear.com/blob/main/uml.png>.

From the diagram one can observe how the Single Responsibility Principle is followed. Also one might notice the *use of interfaces*. This will allow us to test a specific area of concern independent of other areas of concern by injecting test doubles. This should render the resultant system more testable allowing for a higher code coverage and rest of mind that a unit test is indeed testing a particular subsystem under some state independent of other subsystems.

Section 3

Unit Testing Approach and Design

3.1 Naming convention

The following convention is used when naming any unit test:

$\langle UnitOfWorkDescription \rangle_ \langle StateUnderTest \rangle_ \langle ExpectedBehaviour \rangle$

. Where

- $\langle UnitOfWorkDescription \rangle$ describes the method or class under test.
- $\langle StateUnderTest \rangle$ describes a fixed state of system.
- $\langle ExpectedBehaviour \rangle$ describes the expected behaviour from $\langle UnitOfWorkDescription \rangle$ under $\langle StateUnderTest \rangle$.

Such convention allows us to describe what a test does without delving into the inner workings of the test itself.



The ability to reuse the unit of work description and state under test with different expected behaviours allows us to separate different assertions amongst different tests. Ensuring that a test tests for one thing at a time.

3.2 Test Design: Equivalence classes and boundary value analysis

When testing whether the resultant weather is deemed as cold or warm we need to identify all the values of temperatures for which the temperature is deemed as cold and the other where the temperature is deemed as hot.

It is not ideal to write test for each value in big range of values for which a temperature is deemed as cold or is deemed as hot. With equivalence class and boundary value test analysis we test for which values are considered of importance when testing.

In this case we identify 2 equivalence classes i.e. sets where each value in the set represents a particular outcome i.e.

- $\{x : x \leq 15\}$. Values for which the temperature is deemed as cold.
- $\{x : x > 15\}$. Values for which the temperature is deemed as warm.

We test for these values since errors are prone to happen at the bounds of each equivalence class. The former class is exactly bounded below by 15 and the latter class is tightly bounded above by 15.

Hence as bounds we take 15 and 15.00001 that is a value slightly greater than 15.

```

1 private static final double IS_COLD_THRESHOLD = 15;
2
3 @Test
4 public void
5 decideWhatIsFutureWeather_slightlyBiggerThanFifteenCelsius
6 _temperatureIsDeemedAsWarm()
7 throws Exception {
8     Mockito.when(weatherClientMock.getWeather(dummyLocation, day)).thenReturn(
9         new Weather(IS_COLD_THRESHOLD + 0.00001, 0));
10    WeatherDecider weatherDecider = new WeatherDecider(locationClientMock,
11        weatherClientMock);
12    Assertions.assertFalse(weatherDecider.decideWeather(IATA, day).isCold());
13 }

```

Listing 3.1: Test for second identified bound



Unit tests are written for some other values in each equivalence class. Equivalence classes and boundary value analysis are applied to other scenarios where a unit under test is tested against equivalent ordered range of states.

3.3 Testing of direct I/O

Testing the direct I/O of a component under test follows the pattern below:

- **Setup:** set up fixtures of test
- **Exercise:** call some method to test with the direct input
- **Verify:** Assert that the direct output i.e. the return value is as expected

The service expects a capitalized IATA. For example let us verify that we except that a lowercase IATA is invalid in our system.

```

1 @Test
2 public void
3 isIATAValid_ThreeLowercaseCharacters_Invalid() {
4     // exercise
5     boolean isValid = validation.isIATAValid("mla");
6     // verify
7     Assertions.assertFalse(isValid);
8 }

```

Listing 3.2: Test for first identified bound

3.3.1 Testing the menu

The menu is designed as independent component in our system

The menu was designed to be testable by injecting an `InputStream`, `PrintStream` and a `IRecommendClothing` (the interface for subsystem which returns the clothing recommendation based on weather status).

In the test classes the input stream and print stream where set to

`ByteArrayInputStream` and `ByteArrayOutputStream` respectively. This allows us to fix a user interaction with the system and assert the menu content on screen.

For example, we may want to simulate a valid user interaction system as per spec.

Suppose the following interaction with the system which is valid as per spec:

Whenever the user chooses option 2 he is requested by the menu to write a IATA and a date then the menu will make a call to clothes recommender DOC to check what clothes to recommend based on the information inputted.

```

1 @Test
2 public void whenRunningMenu_option2WithWrongIATAoption3
3     _expectedWrongIATAMessageAndQuitMessage()
4     throws Exception {
5     //setup
6     Mockito.when(recommendClothingMock.recommendClothing("mla", "2023-01-01")).
7         thenThrow(new IllegalArgumentException("IATA is invalid"));
8     String option = "2\nmla\n2023-01-01\n3\n";
9     InputStream dummyInputStream = new ByteArrayInputStream(option.getBytes());
10    Menu menu = new Menu(dummyInputStream, PRINT_STREAM, recommendClothingMock)
11    ;
12    //exercise
13    menu.start(); // start menu
14    //verify
15    String expectedMenuOutputContent = printMenu() + "Enter 3 digit airport
16    IATA (in uppercase format) : " + "Enter day of arrival (in format YYYY-MM-
17    DD): "
18    + "Error - IATA is invalid\n" + printMenu() + "Exiting WeatherWear.com";
19    Assertions.assertEquals(expectedMenuOutputContent, DUMMY_OUTPUT_STREAM.
20    toString().trim());
21 }

```

Listing 3.3: Testing menu content against a valid menu interaction

The input stream captures the interaction and will act as the direct input for menu component and the total content on screen represents the direct output of the menu component and is asserted against a string representing the expected content.



The assertion will increase in complexity as the input stream gets bigger as the expected output will grow in size with the user interaction. One might consider asserting the existence of some part of output stream rather than asserting its contents in its entirety. In this implementation the whole print stream content is verified. Note we also make *IWeatherDecider* dependent on component act as a Saboteur as explained in 3.5. This tests the graceful handling of the interface with expected expectations.

3.4 Testing of indirect I/O

3.4.1 Implementing an interface

All the sub components making up the WeatherWear system that are required to be used by other sub components implements an interface.

3.4.2 Constructor Injection

Whenever a sub system needs to make use of another sub component to properly function then the interface is injected using constructor based dependency injection.

Constructor dependency injection is used instead of other dependency injection alternatives since

- The dependencies are required by a sub system to properly function.
- Constructor injections clearly states the dependencies of the class (subsystem).

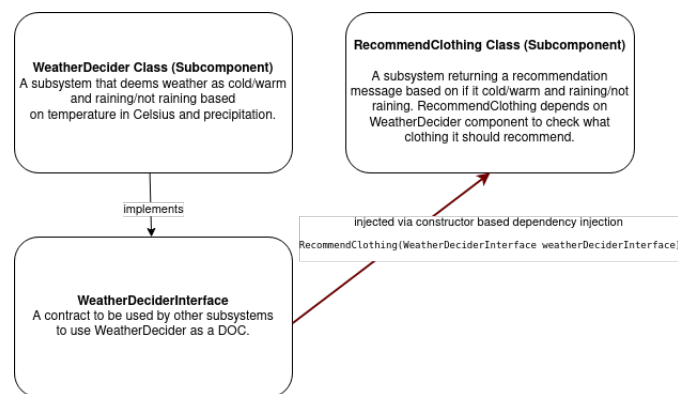


Figure 3.1: WeatherDecider subsystem as a DOC of RecommendClothing subsystem

3.4.3 Mocking

Mocks are as used test doubles.

For example let us consider a test for RecommendClothing subsystem.

We want to check that given a particular Weather instance generated by WeatherDecider gives the correct clothing recommendation.

In this test case the RecommendClothing class is the sub-component under test and the WeatherDecider class is its dependent on component and hence its indirect output should be mocked.

```

1 private String RECOMMEND_LIGHT_CLOTHING_AND_NO_UMBRELLA = "It is warm so you
  should wear light clothing.\nIt is not raining so you don't need an
  umbrella.";
2
3 @BeforeEach
4 public void setUpBeforeEach() {
5     weatherDeciderMock = Mockito.mock((WeatherDeciderInterface.class));
6 }

```



```

7
8 @Test
9 public void
10     RecommendClothing_WarmAndNotRaining_RecommendLightClothingAndNoUmbrella()
11     throws Exception {
12         //setup
13         boolean isRaining = false;
14         boolean isCold = false;
15         WeatherPossibility weatherPossibility = new WeatherPossibility(isRaining,
16             isCold);
17         Mockito.when(weatherDeciderMock.decideWeather()).thenReturn(
18             weatherPossibility);
19         RecommendClothing recommendClothing = new RecommendClothing(
20             weatherDeciderMock);
21         //exercise
22         String message = recommendClothing.recommendClothing();
23         //verify
24         Assertions.assertEquals(RECOMMEND_LIGHT_CLOTHING_AND_NO_UMBRELLA, message);
25     }

```

Listing 3.4: Mocking WeatherDeciderInterface using Mockito



(Why not stubs instead?) In this case the sub-component under test has no indirect output so a simple stub would suffice. Mocks are used everywhere for uniformity.

In simple cases stubs may prove to be a better option since the behaviour of the response of the dependent on component is defined elsewhere making the tests more clear and less brittle.

In setup shown in Listing 3.8 a new mock environment will be setup for each test allowing ensuring that tests do not interfere with each other.

3.4.3.1 Handling indirect outputs

The real power of mocking shines when a component under test sends an indirect output.

A real system scenario and its corresponding test are given below.

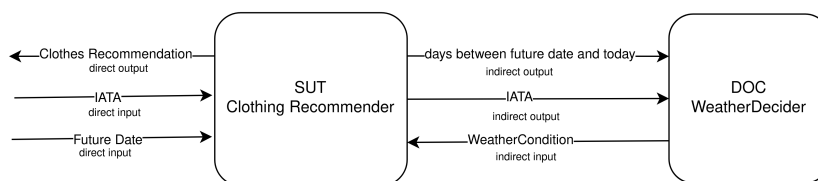


Figure 3.2: Direct vs Indirect I/O

Fixing the clock current date and passing this date as a future date as a direct input in our system we expect that as indirect output we will have 0 days. All the indirect outputs are captured when mocking the return of the DOC with the direct outputs as arguments.

```

1 private Clock clock;
2
3 private static final String validIATA = "MLA";

```

```

4 private static final String expectedWarmAndNotRainingClothingRecommendation = "
   It is warm so you should wear light clothing.\nIt is not raining so you don
   't need an umbrella.";
5
6 private WeatherDecider weatherDecider;
7
8 @BeforeAll
9 public void setUpBeforeAll(){
10     clock = Clock.fixed(Instant.parse(("2023-01-01T00:00:00.00Z")), ZoneId.of("
       UTC"));
11 }
12 @BeforeEach
13 public void setUpBeforeEach(){
14     weatherDecider = Mockito.mock(WeatherDecider.class);
15 }
16
17 @Test
18 public void
   RecommendClothing_WarmAndNotRaining_RecommendLightClothingAndNoUmbrella
19 Message() throws Exception {
20     //setup
21     boolean isRaining = false;
22     boolean isCold = false;
23     WeatherPossibility weatherPossibility = new WeatherPossibility(isRaining ,
       isCold);
24     Mockito.when(weatherDecider.decideWeather(validIATA,0)).thenReturn(
       weatherPossibility);
25     RecommendClothing recommendClothing = new RecommendClothing(weatherDecider ,
       clock);
26     //exercise
27     String message = recommendClothing.recommendClothing(validIATA , "2023-01-01"
       );
28     //verify
29     Assertions.assertEquals(expectedWarmAndNotRainingClothingRecommendation ,
       message);
30 }

```

Listing 3.5: Indirect outputs are "ML" and 0

3.5 Error handling

It is preferred for some components to return exceptions rather than error codes. Returning an error code promotes the use of expressions in predicates i.e.

if(something(parameter)==E`OK) that may lead to deeply nested structures, apart from this a caller must deal with error code immediately.

On the other hand exceptions can be dealt with try/catch separating the error processing code from the expected to work code.

With some known inputs an exception should be thrown. This scenario is tested using a test pattern called the Expected Exception Testing.

3.5.1 Expected Exception Testing

JUnit's `assertThrows` and `assertEquals` methods are used to verify the type of exception and the exception message an invalid direct input should throw in a component under test.

For example when trying to obtain a clothing recommendation for a future date the inputted date should be between 0 to 10 days from today's date if not then a `DateTimeException` exception with an appropriate message is thrown.

For example if the date is in the past then we should expect an exception. This is tested as follows.

```

1 @BeforeAll
2 public void setupBeforeAll(){
3     clock = Clock.fixed(Instant.parse(("2023-01-01T00:00:00.00Z")), ZoneId.of("
4         UTC"));
5 }
6 @BeforeEach
7 public void setupBeforeEach(){
8     weatherDecider = Mockito.mock(WeatherDecider.class);
9 }
10 @Test
11 public void RecommendClothing_DateIsInThePast_ExpectedDateTimeException()
12     throws Exception {
13     //setup
14     String invalidDate = "2022-12-31";
15     RecommendClothing recommendClothing = new RecommendClothing(weatherDecider,
16         clock);
17     //verify
18     DateTimeException exception = Assertions.assertThrows(DateTimeException.
19         class, () -> {recommendClothing.recommendClothing(validIATA, invalidDate);},
20         "DateTimeException is expected");
21     Assertions.assertEquals("Expected date to be between 0 and 10 days",
22         exception.getMessage());
23 }

```

Listing 3.6: Expected Exception Test

3.6 Saboteur Pattern: Call backup location service if service fails

It is required by the system that if a location cannot be received by a service than it will try to obtain it from a backup service.

This strategy is implemented in class LocationClient where if the get current location service fails i.e. throws a known exception then it will call a get current location backup service and if the get location by IATA service fails i.e. throws an exception the it will call a get location by IATA service.

The rest client interface is mocked and acts a saboteur (injects invalid indirect input such as an exception to a component under test) when the IRI is of the first service. With this setup we call the location client service and we assert that the backup service has been called.

Respective test displaying this are given below.

```

1 private final HttpResponseMessage GOODLOCAITONBACKUPCLIENTSERVIERESPONSE = new
2     HttpResponseMessage(200, "{...}");
3
4 @BeforeEach
5 public void setupBeforeEachTest() {
6     IRestClient = Mockito.mock(IRestClient.class);
7 }
8 @Test
9 public void
10     WhenGettingCurrentLocation_WithSocketTimeOutWithFirstLocationService
11     _CallBackUpServiceOnce() throws SocketTimeoutException, JsonProcessingException
12     {
13     //setup
14     Mockito.when(IRestClient.request(HttpRequestMethods.GET, "http://ip-api.com
15         ", "/json")).thenThrow(SocketTimeoutException.class);
16 }

```

```

12 Mockito.when(IRestClient.request(HttpRequestMethods.GET, "https://ipapi.co"
13 , "/json")).thenReturn(GOODLOCAITONBACKUPCLIENTSERVIERESPONSE);
14 LocationClient locationClient = new LocationClient(IRestClient);
15 //exercise
16 locationClient.getLocation();
17 //assert
18 Mockito.verify(IRestClient, Mockito.times(1)).request(HttpRequestMethods.
19 GET, "https://ipapi.co", "/json");
20 }

```

Listing 3.7: Get current location service fails due to SocketTimeout Exception then check if back up service is called

```

1 @Test
2 public void
3 WhenGettingLocationFromIATA_WithSocketTimeOutWithFirstLocationService_CallBackUpServiceOnce
4 () throws SocketTimeoutException, JsonProcessingException {
5 //setup
6 Mockito.when(IRestClient.request(HttpRequestMethods.GET, "https://www.
7 iatageo.com", "/getLatLng/" + IATA)).thenThrow(SocketTimeoutException.class
8 );
9 Mockito.when(IRestClient.request(ArgumentMatchers.eq(HttpRequestMethods.GET
10 ), ArgumentMatchers.eq("https://airport-info.p.rapidapi.com"),
11 ArgumentMatchers.eq("/airport"), Mockito.any(), Mockito.any())).thenReturn(
12 GOODLOCATONFROMIATACLIENTBACKUPSERVICERESPONSE);
13 LocationClient locationClient = new LocationClient(IRestClient);
14 //exercise
15 locationClient.getLocation(validIATA);
16 //assert
17 Mockito.verify(IRestClient, Mockito.times(1)).request(ArgumentMatchers.eq(
18 HttpRequestMethods.GET), ArgumentMatchers.eq("https://airport-info.p.
19 rapidapi.com"), ArgumentMatchers.eq("/airport"), Mockito.any(), Mockito.any
20 ());
21 }

```

Listing 3.8: Get future location service fails due to SocketTimeout Exception then check if back up service is called

Section 4

Code Coverage

The initial goal was to at least get 80% line coverage as a way of checking that the system is tested in its entirety. The initial goal is satisfied with 93% line coverage.

The high coverage is obtained due to the design of a testable system. The system is deemed as testable since it is composed of multiple subsystems where each system is tested independently of the other by use of dependency injection and test doubles. .

All classes gets 100% line coverage except for the main class which gets 0% coverage.

The main class ties all the components making up the system together. Integration testing is well suited for checking that all the components function as expected as a unified group (these are not units tests as they test multiple behaviours at once!).

4.1 Report

Current scope: all classes

Overall Coverage Summary

Package	Class, %
all classes	92.3% (12/13)

Coverage Breakdown


Package 	Class, %
org.weatherwear	0% (0/1)
org.weatherwear.clients.GenericRestClient	100% (3/3)
org.weatherwear.clients.LocationClient	100% (1/1)
org.weatherwear.clients.Models	100% (2/2)
org.weatherwear.clients.WeatherClient	100% (1/1)
org.weatherwear.clothesrecommender	100% (1/1)
org.weatherwear.menu	100% (1/1)
org.weatherwear.utilities	100% (1/1)
org.weatherwear.weatherdecider	100% (2/2)

Figure 4.1: Overall report



Note that having good high coverage is good for checking which components of the system are not tested and hence we identify why they are not tested and if no good reason arise we write tests for them. It does not say anything about the quality of the tests. Apart from obtaining a good coverage various inputs and outputs where tested according to the expected behaviour of the system. Inputs leading to a happy path of code and others leading to errors are tested. Deciding the inputs that should be test using boundary analysis. Also it was made sure that each and every single one of the test has a descriptive name, tests one thing, is deterministic, has no conditional logic, independent of other tests and is understandable.