qwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmrtyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmrtyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmrtyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmrtyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmrtyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmrtyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmrtyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnm

|  |
| --- |
| **Final Assignment Testing**  Writing Assertions (*Hamcrest matchers*),  Test Driven Development (*Mockito*) and  Behavior Driven Development (*Cucumber*)  5/24/2016  **Madalina Dragan, Boyko Surlev and Cristi Nita** |

Table of content

Contents

[Our choice of project and the concept behind it 2](#_Toc451780574)

[Test Driven Development 3](#_Toc451780575)

[Mockito mocking technique 3](#_Toc451780576)

[Positive aspects of TDD 4](#_Toc451780577)

[Refactoring after TDD 4](#_Toc451780578)

[Behavior Driven Development 4](#_Toc451780579)

[Cucumber Assertions 4](#_Toc451780580)

[Gherkin 4](#_Toc451780581)

[Writing assertions 5](#_Toc451780582)

[jUnit assertions 5](#_Toc451780583)

[Hamcrest assertions 5](#_Toc451780584)

[Hamcrest matchers 5](#_Toc451780585)

[Final thoughts 6](#_Toc451780586)

# Our choice of project and the concept behind it

The name of our project is Expenses Calculator. The concept is simple – we created a tool which could be used to save, evaluate and display the income and the expenses of the user. The tool can make calculations and help the user handle their transactions. We preferred to have a distinct and specified idea regarding what exactly we want to work on, simply because we thought that this would make the workflow easier and the work would become more interesting, since built and actual tool, which may be used by any user, who has some basic computer understandings. An expenses calculator was our best guess.

We had to use at least three of the test topics which were listed in the final assignment papers. We chose Test Driven Development, Behavior Driven Development and Assertions writing. We thought that the synergy between those topics will be perfect for our solution.

TDD gave us the possibility to investigate a mockery framework for unit tests in Java called “Mockito”. BDD faced us with a tool called “Cucumber”, which is useful for creating User Stories. The writing of assertions was a nice opportunity to test our Hamcrest skills. We learnt more about Matchers and assertions in general.

Next we will describe into details how we used TDD, BDD and Hamcrest and our reflection about it.

# Test Driven Development

Test Driven Development is based on three main steps. First we need to have a plan for what the method should do. Then we need to create the test for it and to check it fails. Then we need to add the actual functionality and check if it is working properly via the tests.

## Mockito mocking technique

Mockito is a mockery framework for unit tests. Essentially it mocks any object you pass to the mock method of the mockito library. Below it can be seen an example from our code, where we mock the mappers in the façade.

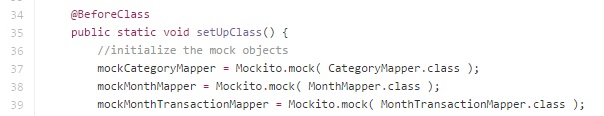


Figure 1. Example of mocking the mappers in the facade

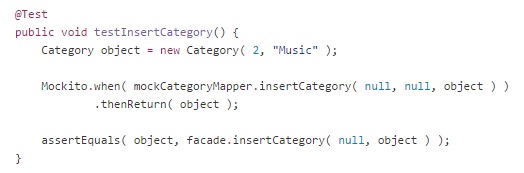
In this way the developer receives the possibility to manipulate the return objects/answers from the methods in the mock objects. An example is presented below.

Figure 2. Example of manipulation of the mapper responses

Via this technique any developer can manipulate and present what should be the functionality of a specific method. Then the tester can use a simple assertion test a class in isolation. A good example for this functionality can be a defaultMapper object and a façade object. The tester can mock the Mapper object, so that they may receive the correct response. Then they can test freely and securely the functionality of the Façade.

## Positive aspects of TDD

Test Driven is great in a project where the quality matters, not the time limits. It is really helpful, because one can easily find their mistakes and evaluate them. At the same time, the developer has a better understanding of what is the purpose of method, so the code will have a higher quality. Not only that, but it presents the user with a nice, smooth and straight-forward technique, which may change the development process from being an everyday job to a pleasant experience.

## Refactoring after TDD

Our team reached a point in which we had to refactor some of our code during our process of development. We assumed that this will be a painful process, because we need to check if the changed functionality actually works as expected. However, the checking process was as smooth as possible, because we already had tests for all important methods in our program. We simply had to run the tests, which we built using the TDD approach. Therefore, another advantage of TDD is that it makes refactoring faster.

# Behavior Driven Development

BDD was a really vague topic for most of us. We had to read and understand the concepts of it. Firstly, we came up with the idea that the bottom line of the Behavior Development is writing user stories. However, we were wrong. We discovered tools such as Cucumber, as we were advised in the assignment requirements. Our team was amazed and impressed by the fact that the Cucumber tool is supported in many languages such as Russian, Bulgarian, Arabic, Romanian etc.

## Cucumber Assertions

Cucumber is the perfect tool for running automated tests. It executes files, which have the .feature extension. The code inside those files should be written in a language called Gherkin. However, we believe that Gherkin and Cucumber are mostly referred as the same language/tool since GitHub evaluates the type of the code written in Gherkin as Cucumber.

## Gherkin

In few words, Gherkin is has a plain-text English structure. It is designed to be easy to use by non-developers. The structure of a feature description is presented in Figure 3.

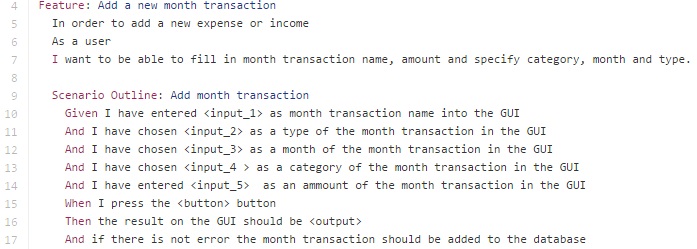


Figure 3. Feature description for inserting a transaction

This structure should be followed by a table where the writer should present the possible inputs and the correct output. This approach is very powerful, in case a team is struggling at understanding the basic concept of a project. The team/developers may create some Cucumber assertions with the customers/ designers/ managers and make some core decisions regarding the functionality of a specific product.

# Writing assertions

Writing assertions is an essential part of the testing process. They are a code feature of the Test Driven Development. They are also handy in case of refactoring. They may provide the developer with instant feedback after they had made changes to the code structure.

## jUnit assertions

jUnit assertions are the basic type of testing, known by our team in Java. They provide the most common fundamental functionality for testing, but sometimes this is not enough. This is why we are using Hamcrest assertion and matchers. However, a must to mention is that our solution includes both jUnit and Hamcrest tests of the Mappers. The reason for that is because we started in a slow way by building the basic structure of the tests and Mappers without any external libraries. Then we decided to rebuild our tests, but this time with Hamcrest.

## Hamcrest assertions

The concept of Hamcrest is similar to jUnit, but with slightly different syntax. For example, a popular jUnit method is asserTrue(… , …). The Hamcrest abbreviation of the very same test may be asserThat(… , is( … ) ) or . Here is a nice example:

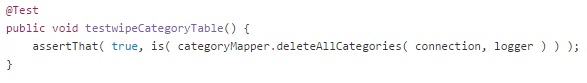


Figure 4. Hamcrest syntax for checking equality of objects

The first parameter is the actual object and the second parameter is called a matcher or an “expected object”.

## Hamcrest matchers

Hamcrest provides yet another special way of testing the functionality of our code. It provides the developer with the possibility of creating their own matches. Below it is presented a screenshot from one of our customized matchers.

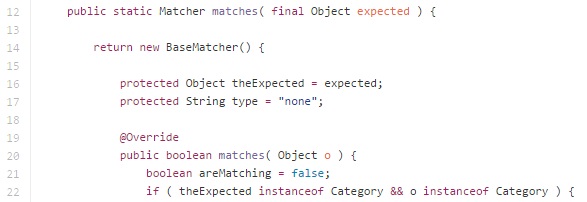


Figure 5. Example of creating customized matchers in Hamcrest

The only requirement is to create an object which returns a specific type of Matcher object. In this case, we are using BaseMatcher(), which includes a matches and describeTo methods, which provide the basic and most essential components of a test . The way to use those Hamcrest Matcher objects is even easier. The developer should simply import the Matcher class they created to the test class and call its functionality by using the matches( … ) method, as presented in the picture below:

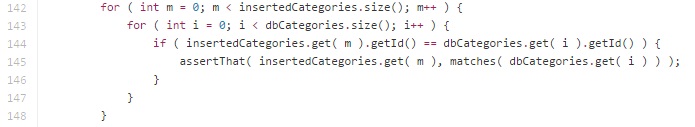


Figure 6. Example of using the hamcrest customized matcher

Hamcrest is really handy and we are sure that it includes much more functionality than what we have presented in this example. We want to point out that the Hamcrest testing plugin is giving the developer the possibility to build their own testing tools, which may serve a specific purpose depending on the type of the project, which is unique.

# Final thoughts

Generally, we are satisfied by the tools which we used during the development of our Expenses Calculator. We learnt something more about several new technologies and this experience can definitely become part of our CV. We had some questions during the development process and want to point out that we are glad that the teachers are providing answers to our questions as fast as possible. Thanks for that!