Testing of Project Gutenberg

[Github](#) [Old Github](#)
Mohammed Salameh
Anders Geer Jakobsen

We decided to combine our databases semester project and our test semester project into one, allowing more focus into the project. Our focus during this project has been on the following topics specifically:

- Continuous integration & Maven
- Test Driven Development
- Unit Testing
- Mocking

Continuous Integration & Maven

During the project, we were facing a few challenges with the integration into Travis, our chosen continuous integration tool for this project. First of all, we are reasonably new to using Travis, not so much using CI in general due to a prior experience. Although Travis is a simple CI tool, we encountered some issues getting Travis to run our tests at all. It turned out to be an issue with our project, as changing to a new project solved this issue after having scoured the internet for solutions.

Maven in itself, is something we've become familiar with over the course of our studies, and as dependency tool. And manages dependencies put into the project or any submodules.

Test Driven Development

We wanted to use Test Driven Development, and started out doing test cases for some input after having thought up one of our methods.

Search variable allows only letters in the english alphabet - [A-Z]
Database name only allows letters of the english alphabet - [A-Z]
Collection name only allows letters of the english alphabet - [A-Z]
Allows any number of characters up to Integer.MAX_VALUE
Does not allow null values
Does not allow empty strings
Does not allow whitespace strings

And while the method has since changed its signature, these test cases and the corresponding tests were developed before the method was in place. Later on during the project, time constraints made it clear that we had to do tests after creation of our methods.

Did the development method make us write better code? Perhaps. It certainly forced us to think about what the methods were supposed to do when we started making tests for them, likely allowing us to use shorter time on getting sidetracked during the methods

With the help of TDD, we forced ourselves to make the necessary Interfaces, although this was hard to see at the start, since there was no body for the code. But it proved itself useful and made our unit tested code less tightly coupled.

Unit Testing

Our unit testing consists of testing small parts of our project, and only stuff that isn't either generated automatically or only does a single call like the methods parsing user input. In the above case, we wanted to use polymorphism from the beginning, to get used to writing more testable code. And similarly, we did have challenges, such as any variables in an interface are static and final by default, disallowing us to use them in the classes implementing the interface. And thus, decided to move some parts of the code, which would be the same, to every single class implementing it. After having been thinking about this afterwards, it might have been a better idea to move it to a single superclass, allowing every single class to use the same implementation of the method, without the need to copy the entire method. As we thought of this quite late in the process, we opted against implementing it that late.

On the evening of May 25th our database servers went down due to what looks like an outgoing dos attack from both of these, due to this we skipped an additional 8 unit tests in addition to our unit test supposed to pass. This was disabled due to issues with our mongo database blocking all incoming requests from any Ip other than loopback. - We thus have 9 skipped tests, which is something we are aware of.

We used both Normal Unit testing with J-Unit and JUnit with Hamcrest. Both of them are sufficient for us personally, but we do agree on the part that Hamcrest, with its existing matcher classes, makes tests more readable and closer to the normal speaking person. Also, the tests execution, fail/success, hamcrest can be implemented to give you a more accurate description on what went wrong.

Mocking & Dependency Injection

We were mocking away some parts of it, and realized we cannot allow checks for method calls in a constructor, as mocking never creates a new object, and this never goes into the constructor.

We have these Dependencies we can Mock away:

- IService: This interface is made for different save and load services, that would be required from our program. For now it is only PictureService that implements it, and has a method called savePictureFromUrl.
  If we want different loading/saving functionalities, we can simply add a new class with its specific implementations or even update our Interface to hold more methods.

- IUrlFetcher: The interface is made for different 3rd party URL’s, since our program is getting information, currently, from google we have a specific class for their request/response format. GoogleURL, simply just implements 2 methods, from its contract to IUrlFetcher:
    - createUrl
    - getUrl
- IWorker: Is made to be using polymorphism. Allowing the call to the methods connecting and getting data from the databases to be exactly the same.

In our learning goals, we had mentioned that our knowledge of Mocking, at the start of the project, was not sufficient. But with the help of the Book: Effective Unit Testing from the class, we have included the necessary SOLID principles that our project needed, in order for an easier testing phase. Even though we did include TDD in our process, we had to do a lot of refactoring of the tests. The resulted in some complications.

We also saw the light in terms of Dependency Injection(IoC) and Inversion of Control(DI) patterns, that removes dependencies from our code. With our interfaces we created abstractions by having our dependency classes, GoogleURL, PictureService, CitySearch, not initialize dependency within the class itself. This allows us to call the dependency then pass it to the implemented classes.

What we didn’t include, that else was written in our learning goals.

Test doubles: is considered a stand-in for the real code, and can thus include mocks, stubs and fakes and while we included mocks, we decided against using stubs, even though a stub is widely considered made without any logic whilst mocks has expectations about how it should be called. Thus our mocks can be considered combinations as we do return specific values, but also use mocks to verify number of usages.

Dummy objects: are used in our testing in the PlaceTest class, as they are objects passed through parameters without the actual parameter being used.

Fake Objects: While we could have used an implementation of fakes as our databases, we opted against it as Neo4J is quite hardware intensive and we wanted our tests to run on similar hardware to the implementation in the database project.