TEST SEMESTER PROJECT EVALUATION & REFLECTIONS

By Group 4 – Murched Kayed, Hallur vid Neyst & Zaeem Shafiq

MAY 24, 2019

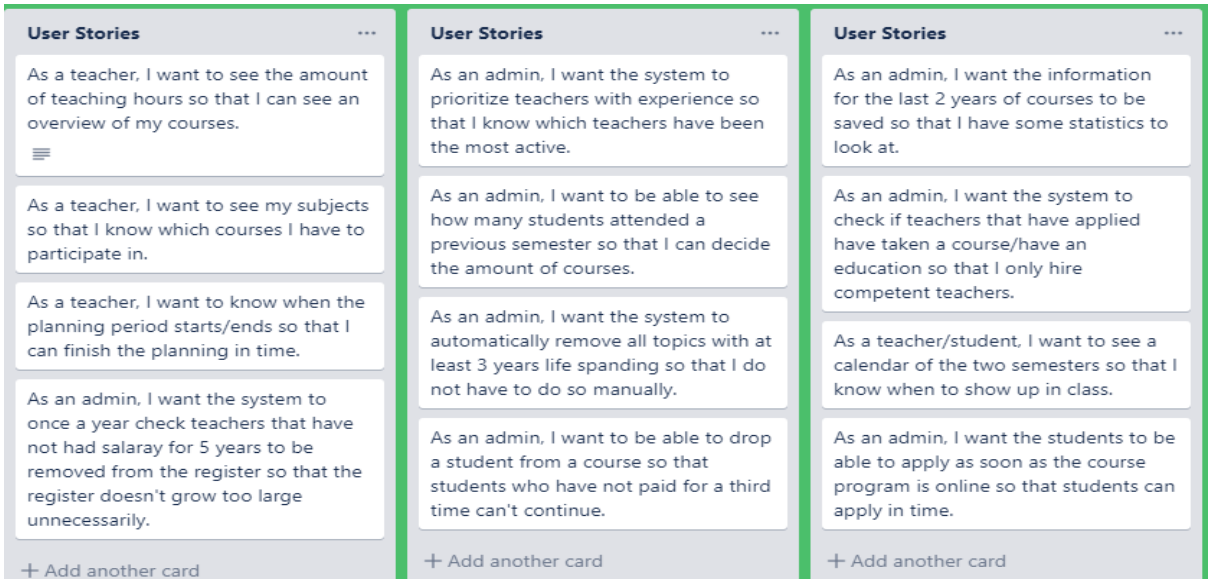
COPENHAGEN BUSINESS ACADEMY

Table of Contents

Team collaboration

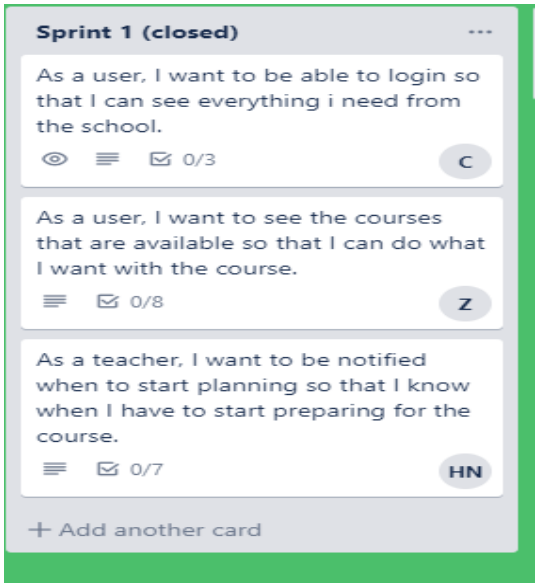
We started our development process by reading the assignment description thoroughly, brainstorming and discussing if we all three understood the needed requirements in the same way. From the outset, we agreed that we would develop our system according to the agile project management framework Scrum's principles. We therefore decided to start writing user stories, which we later could split into sprints. In order to have an easy overview of our Product Backlog, we decided to use Trello as our collaboration tool. Trello simply organizes projects into boards and shows what's being worked on, who's working on what, and where something is in a process.

Below is shown a picture from Trello containing our user stories:



Before we gave out user stories, each user story was split into tasks and we also wrote acceptance criteria for each user story. We tried to make sure that all the user stories were independent, which would make the user stories have a good test-manner face. This would be an advantage since it will make it easier to test each one of them independently.

Below is shown an example of one of our sprints in Trello.



From the start, we also agreed that we would use Test-driven development (TDD) in our development process, i.e. writing the tests before implementing the code. In addition, we also agreed that we would use Behavior-driven development (BDD) using Cucumber and Gherkin.

We agreed to use Java Server Pages (JSP) and servlets as we could save a lot of time and thus be able to focus a lot more on the testing part. Alternatively, we would have spent time creating a separate frontend (and an API) for our system and that's exactly why it is an advantage to use JSP as it contains both the backend and the frontend in one single project. However, we are aware that JSP is an older technology that is no longer used, so we would obviously not use it for a real project either.

A large part of our communication has been done using Discord, where we regularly held a lot of online meetings. We also used AnyDesk for screen sharing if there was a need to show code or something else. In addition, we also often met physically and had some meetings at the school.

We used GitHub as our code hosting platform and agreed that we should create a branch for every user story. We also agreed that one must create a pull request to merge with the master branch. At last but not least, we also agreed to use Travis CI to run our regressions tests.

Test automation strategy

The following criteria were considered during our decision on what to apply so our automation (the technology by which a process or procedure is performed with minimal human assistance) is effective and efficient:

- We want to make our test automation strategy result oriented. We felt it was important that we could understand and define the objective of the automation strategy. This was done by using a tool called JaCoCo (Java Code Coverage) and by taking a look at our acceptance tests in Trello. JaCoCo would tell us which methods we were missing, while the acceptance tests could determine if the missing test areas were necessary to implement.

- We want to work with a suitable test automation tool (Selenium and Travis CI). Selenium is made precisely for the purpose of improving the test automation strategy. We really liked the idea of it being able to perform without human assistance, whereas without it we would have to manually perform in order to test the web pages etc. We also liked the idea of Travis CI, because for one it was a requirement of the assignment, but also it gives up the setup needed for CI/CD. This assured that we have a working project on our GitHub repository where every time changes are being made, the Travis CI will run the build and tell if the build/needed tests are successful.

- We wanted to make sure that our tests are all reusable, and that they are in no way non-repeatable, and do not defect the system and/or the production database. This was being done by using a test database each time we needed it, and then also clean it either before or after the test.

Non-functional requirements

We had a lot of thoughts about non-functional requirements, but due to lack of time, we did not get to implement it. Among other things, we talked about we could test the performance using JMeter and BlazeMeter, which would have been a non-functional test. In addition, we also talked about that we could have used non-functional

security testing in connection with our login, but unfortunately, we did not have time for this either.
However, we did use some non-functional testing in the form of usability testing. This was done to see how easy to use our system was and was simply done by letting real users make use of the system.

Testability

- **Observability**
  - While writing our methods, we had observability in mind to make it possible to see where it went wrong if the system crashed. In our case we did this by writing some “print”-statements in the code that tell us how far we have come in the code. E.g. if we want to create a student object and insert it in the database, we print “creating student” just before we initialize the student object. Then we print “creating SQL query” just before our SQL query and so on and so forth. This allows us to easily observe how far the runner has come in the code.
- **Controllability**
  - We reduced the amount of dependencies, e.g. by ensuring that every class had a constructor. This separated the application logic from the instantiation logic and made it possible for us to make us of the dependency injection when testing. Now if we instantiate in our test classes, it should have its own act and therefore never interfere with the system.
- **Smallness**
  - All our tests meet the criteria of being in the “smallness”. In terms of the size of the project, we have reduced the amount of lines of code as much as possible, so that there is less to test. When it comes to the actual tests, we also want them to have a “singularity”, meaning that the instantiations used should be as little as possible, which also means that the test should focus on one feature at a time.

Test design techniques

Among other things, we have chosen to make use of Specification-Based/Black-Box techniques in the form of Equivalence Partitioning (EP) and Boundary Value Analysis

(BVA). This has proved to be extremely beneficial. These techniques helped us, among other things, to reduce redundant test cases, achieve a very high-test coverage and structuring the test cases.

Equivalence Partitioning (EP) was used to divide set of test condition into a partition which should be considered the same. We partitioned the test input data into a number of classes having an equivalent number of data. The test cases were then designed for each class/partition. It allowed us to identify valid as well as invalid equivalence classes. EP helped us to reduce the number of test cases.

Boundary Value Analysis (BVA) was used to explore errors at the boundary of the input domain. It is observed that in most applications, errors occur at the boundary values. We used Boundary Value Analysis as the next part of Equivalence Partitioning where we selected test cases at the borders of the equivalence classes. This helped us a lot to catch any input errors that might interrupt with the proper functioning of our program.

In addition, to test the software solution's internal structure and code, we also used Structure-Based/White-Box techniques in the form of unit testing and test coverage measurement. We gained a lot of benefits from this, e.g. our test cases could be easily automated.

Unit testing turned out to be extremely beneficial to us as it really goes hand-in-hand with agile programming. Unit testing build in tests that allowed us to make changes more easily. In other words, unit tests facilitated safe refactoring. In addition, unit tests improved the quality of our code. Unit testing also helped us identifying every defect that may have come up before code was sent further for integration testing. Furthermore, unit testing also helped us finding bugs early in the process.

We used JaCoCo as test coverage measurement tool. JaCoCo measured how many lines of our code were actually tested. JaCoCo was very useful for us as it generated test coverage reports that gave us a clear picture of how much of the code we had tested and how much we needed to test.