Lucas A. Nielsen
Christoffer Ludvigsen
Kaloyan Kolev

Gutenberg Project testing reflections:

Learning Goals:

1. We want to use Test Driven Development to see if the quality of our code is increased (as opposed to our regular assignments and other work were TDD was not present)

   Unit testing
   Integration testing
   Continuous Integration
2. We want to find good strategies and associated technologies to test a project using databases.
3. We want to use appropriate static techniques to reduce error margin.
4. We want to test the performance of the databases in order to improve product quality.

During the project each of us used test driven development to various decrees and doing this process we encountered two problems

1. Test driven development becomes more complex when you are trying to code something you haven't tried before(new logic, libraries, ect) it is harder to predict the solutions when you don't fully understand the methods you are gonna use yet

2. Test driven development is a different mindset so while it works great when things are going smoothly when complications and time pressure starts appearing it is hard to keep working in a way that feels unnatural and as a result we were prone to switch back to our ingrained work processes in those situations

Overall we think that test driven development has a lot of merit. It ensures the tests gets made, it forces the tests and the code to be made simultaneously so that people know when the code is as should be, since it can pass the test. And lastly it help guarantee that the code lives up to the project requirements, since the tests are build of those.

Like most thing test driven development is a tool, how proficient you are at it and how well it matches with the project, will have an impact on its practicality.

We felt that in this project, we used to much time trying to setup the test, which resulted in time delay in the production, and the code should just work. So we do not feel that by using Test driven development in this project has not improved our code quality.

For continuous integration we used Travis on our project, it ran the test every time the code was pushed to github and informed us via mail if our code had failed.
Generally the purpose of Travis is to run the test every time we merge to test that no complications arise and inform the people who are working on the project that they should not pull a broken project. In our project what Travis did most for us was to run the tests every

time we update the project. Keeping track of the project itself wasn't nearly as taxing as it would be for a company, since we had at maximum 2 people working on the same code at any point in time so for that purpose travis wasn't as necessary.
All in all travis didn't help us that much, but the time investment of setting it up was also very insignificant.
In this project we wanted to find a good strategy to test a project that makes use of one or more Database.
The strategy we ended up using was, to make use of a stub. A hard coded class that would return a predefined value.
By using this, we did not have to rely on a real database for testing the logic. Which was good thing since we only had the real database open and running in the last days of the project.
But even if we did had a real database from the start we still would have use the stub, since we would not have the tests to connect to a real database every time we want to test the logic or the GUI layer.
We feel that this strategy was the right one for our project, and it help us to quickly test our logic, which made us able to fix think quicker.

Before we started Making the Gutenberg project we decided that we would try to make use of appropriate static testing techniques. To see if it would help us catch problems early on in the development process and thereby save time.
In our development process we have used technical review, walkthroughs and informal review. Particularly in the early stats of the development.
If we look back and see how well these techniques has helped, doing this project. It has help clarify some misunderstanding between group members that, if not have been discovered would probably have resulted in us not having a working program for the deadline.
As an example we made a walkthrough of the first version of the interfaces and found some issues, with input and output values. It was a simple misunderstanding between the group members, and was easily fixed.
We held some techniques review regarding the database, the data structure and how to best import the data into our to databases.
We did also have numerous informal review where one group member would look at another's code or document, to help finding error, try help improve the code quality or make sure that it lives up to the set requirements.
An example of this was when one of our group's member´s had problems with the his SQL Queries, they did not return the data he expected and he could not tell why, so we sat down and reviewed it together, and found that he had joined the tables in a wrong order, which resulting that some of the table could not be join, and therefore it only return some of the data.
We can see that the static techniques help us catch some problems before they be a big issue and It also help us solve problems quicker.
However it cannot help us detect all problem before they can turn into a issue.
After we have coded the program, depending on the interfaces, we found that we had to change the interfaces again, because of some unforeseen issues. So we had to change the code right before our deadline which impacted the overall product quality.
However we felt that these techniques were helpful in our development process.

It is our first real project where we really are using these techniques, and only used the techniques in an informal manner. We believe with some more experience with these techniques, they would be even more useful for future projects.

We wanted to test the performance of the databases in order to improve product quality.
By having performance tests, that can give us an insight in how well our program is running.
We can figure out if the code needs improvement, and thereby improve the overall product quality.
We decided to do performance measurements to see how much time it takes between the two databases to complete their requests.

Average Times

Query	PostgreSQL	Neo4j
1	1784.2 ms	558.6 ms
2	134.8 ms	214 ms
3	137.2 ms	112.4 ms
4	1903.8 ms	3023.6 ms

Median Times

Query	PostgreSQL	Neo4j
1	1527 ms	310 ms
2	122 ms	206 ms
3	133 ms	179 ms
4	1698 ms	2927 ms

In our case the time tests we did, show that Neo4j and PostgreSQL are close in performance depending on what method we tested.
However we do believe that these results is not reliable to be used for a comparison.
Because Neo4j did not have the same amount of data as the PostgreSQL, due to lack of time we could not import all the data in Neo4j in time, so it only have around 500.000 relation out of 11 mio relation between books and cities.

We also would have wanted to test the performance of our system, Using Jmeter to make some Load and performance tests, on our Rest API.
But since the system first worked minutes before deadline, we simply did not have the time to do so.