SWE 573 Project Report

Gokhan Ozgozen

December 22, 2016

Abstract

Starting with a project description document, written with natural language; the aim of this project is to apply all fundamental software development incentives to produce valid, well-designed and well-documented software. Challenges were to elicit requirement specifications from a project description, designing software architecture, using 3rd party API support, making a good plan to follow, constantly gardening code base of the project and finally delivering all software 'goods' in time.

I will explain my personal challenges during the project and give final instructions of how to run the project, how it starts up with sequence diagrams and how unit-testing is done and it's code-coverage.

1 Project Challenges

In this section, I will mention all challenges I faced during the project, following a time pattern.

1.1 Requirement Elicitation

Creating a consistent, complete and unambiguous project description was the first problem. Constantly speaking with the stakeholders, asking correct questions asserting mutual understanding was the key to solve the problem. Unless there is a tangible product on the table, it is nearly impossible to validate that what stakeholder wants is the same as what software engineer understands. Therefore, certain techniques needs to be applied in order to avoid costly misunderstandings. Those techniques are to create visual mock-up documents, eliminating ambiguity in the requirements. Also creating a frequent milestones to give stakeholders to something to show for allows them to see project path and allow them to get the feeling of what the final product would look like.

1.2 Project Plan

Creating a perfect project plan is the ultimate impossible challenge. However, rather than rushing into how to produce software, planning resources, aligning important dates and using time in a productive manner creates wonders. In this project, there was a strict deadline and requirements did not change rapidly. This is not always the case. Project plan is not a landmark that built once and watch forever, it adopts and changes as the project itself changes. I had to update my project plan time to time to reflect reality of my status, also I needed a realistic project plan to follow otherwise I would have so much tasks in my hand that It would be so hard to know to start from which one.

My project plan was not detailed enough, pieces of it was not atomic enough that there are still divisions can be made. In the project deliverable you can see that some tasks are way too abstract and needs further work to be better.

1.3 Project Architecture

This project is written with Java JDK 1.7. Also I used Maven, Spring and Hibernate with MYSQL technologies. MVC Design pattern is the main pattern wrapping all sub patters, the system consisted of Singleton Controller and Manager objects, static Utility classes and numerous model objects to represent data.

All business logic is implemented in Management and Utility classes. This is why I wrote almost all my test cases against Management and Utility classes. I will show the test results and coverage in the end of this paper.

I also used Travis CI as my continuous integration tool. You can see that project is valid and all tests are passing currently. Having a good test coverage allows me to follow my project status and be notified if I somehow broke the code base.

1.4 Code Management

Known as code gardening. Our maintenance job as software engineers not only develop a good-quality working project, but also to make sure it stays this way. APIs change, underlying OS changes, user behavior changes, project requirements change, new bugs emerge... All those changes creates a static code base to collapse eventually therefore any project requires constant gardening in a dynamic environment. After participating each lecture, I took notes and created Github issues to remind me of the things the project is missing and I needed to adjust my code to meet these new requirements. However, changing a working code always brings the risk of breaking it. This is where the Unit-Tests prove their importance. I had to maintain high code coverage, ensure all tests are passing and new functionality is added. As the project becomes highly sophisticated, refactorings needed to be done and reduce complexity. Dividing and conquering code blocks, reducing complexity allowed me to understand and read the code better.

1.5 Bugfixes

Bugs started to emerge as soon as I started to write code. The rule of thumb here is to ensure every bug has an issue on Github. We forget things, especially things we like to forget. When there is a bug in the system and developer notices it, next thing the developer does is to solve the puzzle, asking how to fix it. When the answer is found, the bug is forgotten. However, bugs needs to be reported in issues along with the way to show how to reproduce it. I have to admit I didn't do that all the time. There are some bugs even I was too lazy to create an issue to. But those bugs are the ones that survived my bug-fixing milestones the most.

There is a plan to implement new features into the system. Anything new in the system, creates new bugs into the system as well. This is why, for 2 weeks I dedicated my entire time just to fix bugs without introducing anything new to the system. This is how I handled bugs, however with more unit-tests and more coverage, with better front-end UI testing, bugs can be eliminated before getting into system.

2 Reports

At this section I will give metric reports regarding to my project.

2.1 Tests

I created test packages and distributed all my tests into those packages, namely: API, Calculation, Exercise, ModelMapping, Performance and Security. Security package for instance contains classes that are testing security features of the codebase.

2.2 Test Coverage

Test coverage mainly covers Manager and Utility classes because those classes are making all the calculation and business logic implementations.

2.3 GitHub Issues

I opened 63 issues, 32 of them are closed and 31 of them are still at various states. The reason I failed to close all issues are:

- They are contradicting each other

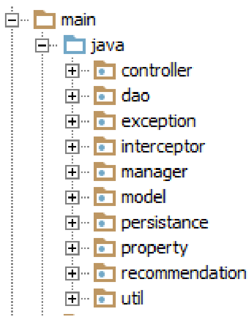


Figure 1: Showing package representation of the main section of my project.

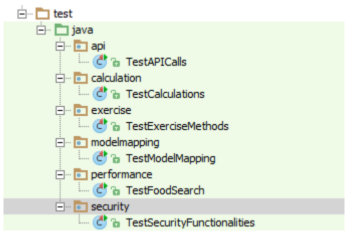
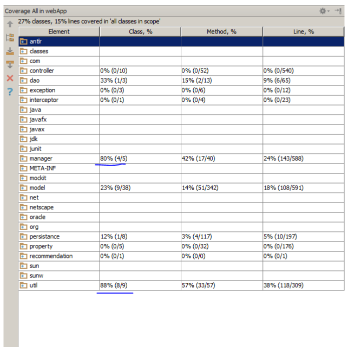


Figure 2: Showing package representation of the test section of my project.



27% classes, 15% lines covered in 'all classes in scope'

Element	Class, %	Method, %	Line, %
antlr			
classes			
com			
controller	0% (0/10)	0% (0/52)	0% (0/540)
dao	33% (1/3)	15% (2/13)	9% (6/65)
exception	0% (0/3)	0% (0/6)	0% (0/12)
interceptor	0% (0/1)	0% (0/4)	0% (0/23)
java			
javafx			
javax			
jdk			
junit			
manager	80% (4/5)	42% (17/40)	24% (143/588)
META-INF			
mockit			
model	23% (9/38)	14% (51/342)	18% (108/591)
net			
netscape			
oracle			
org			
persistance	12% (1/8)	3% (4/117)	5% (10/197)
property	0% (0/5)	0% (0/32)	0% (0/176)
recommendation	0% (0/1)	0% (0/0)	0% (0/1)
sun			
sunw			
util	88% (8/9)	57% (33/57)	38% (118/309)

Figure 3: Test coverage by package, notice utility and manager coverage is high since tests aim those.

- They are opened with high expectations and when things get tight and deadlines came closer, I gave up some of them
- Some of the issues were optional, they are in IceBox status currently.

3 Running Application on Your Machine

3.1 Prerequisites

- Java JDK 1.7
- Maven 3.x.x or after.
- MYSQL database and a user with all CRUD privileges.
- Working an internet connection.
- At least 3GB RAM and 3GB HDD space.
- (Optional) IntelliJ IDEA IDE.

3.2 Setup

- Download the code from github page.
- (Optional) Create a maven project from pom.xml.
- mvn clean install command to install project and dependencies.
- Find the hibernate.cfg.xml file and replace Database user credentials with yours.
- mvn tomcat7:run command to start the server.

3.3 Startup Procedure

Startup takes a while if there is no data in the MYSQL. The server connects to NDB Database via API calls and fetches raw JSON data. Then processes the data and creates internal indexes, also saves the data to MYSQL database. When all data is fetched from NDB Database, the server no longer requests from NDB and uses its own cache for faster service providing. However, initial startup takes a bit long (took 45 minutes on my Intel i7 3820 machine with SSD). Memory usages peaks around 2.8 GB under heavy caching, 3GB should be safe.

As I described initial startup in the Figure-4 above, you can see the server interacts with NDB, takes data and stores it while processing the data. Assigning search keywords, generates guid for food and nutrition datas.

4 Final Words

The aim of this project is not essentially producing a HealthTracker project, but to grasp controlling Software Management tools, facing high level problems like resource and time management, agreement with stakeholders or facing low level technical problems like Hibernate 2nd Level Caching problems. The idea is to learn tools and use them effectively to produce good solutions. I thank my instructor Suzan Uskudarli for her wisdom and patience, looking forward to learn more from her.

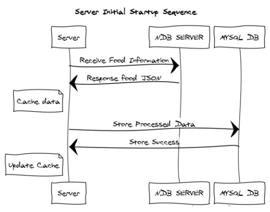


Figure 4: When server is running up for the first time, it takes a while to gather data.

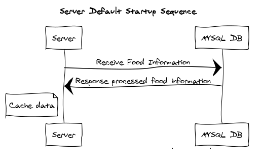

Server Default Startup Sequence
Server
MYSQL DB
Receive Food Information
Response processed food information
Cache data
Server
MYSQL DB

Figure 5: When server is running up in a consequent time, it only interacts with MYSQL DB.