Evaluation: ING Honours

Justin Praas | 17th of August 2017

# Introduction

The ING honours programme was a great way for me to study and get used to many different frameworks and utilities for programming in general. I’ve become more acquainted with Git, the Java API, JSON RPC (great library!) and Maven. Furthermore, it’s the first time I’ve been using an ORM like Hibernate.

My sincere apologies if this was meant to be a formally written evaluation, I completely failed at doing that.

# Implementation decisions

It’s fair to note that at the start of the project we worked in pairs. Some implementations that I’ve followed up upon were initially not designed by me, meaning that when I mention that *I’ve done something* might as well mean *we’ve done something*. Shout-out to Andrei Cojocaru!

## Which decisions made my life easier?

1. Classifying account types, card types and divide functionality of server and client was a big part of the architectural advantage I had during the implementation of all extensions.
2. I believe that the *TimeOperator* and *TimeEvent* that I designed, was a great help to all later extensions involving an event timed in the future. In short, what these classes do is for each event that should happen in the future, a TimeEvent is created. A TimeEvent contains a date, some information and commands (some kind of protocol I made for these objects) that will be handy when we simulate time. In fact, when we simulate time, the TimeOperator comes in to play: for each simulated day the TimeOperator fetches all unexecuted TimeEvents from the database for that specific day and executes the commands.
3. In the later stages of the project, a credit card and a credit account became involved into the program. It turned out that I could use a lot of inheritance in both instances, this would prevent me from writing a great deal of duplicate code in the rest of the program, hence I created two super classes:
   1. Account (superclass of: *BankAccount*, *SavingsAccount*, *CreditAccount*)
   2. Card (superclass of: *CreditCard, DebitCard*)
4. The *RequestValidator* was a handy tool to initially validate an incoming request to prevent exceptions from being thrown. It basically checks if all the information needed of the given method name is present and if the format/syntax of the parameters is correct.

## Which decisions made my life a living hell?

1. Writing ANYTHING to a text file/object file. Overall, the code that reads/writes everything is just a mess and the worst part is that it’s not indexed… After I was done finishing the extension that involved the persistent data I had immediate regrets. I could’ve easily stored *all* the data in the database. Indexed, easily fetched, easily written. In other words: let the RDBMS do all the work, it’s just so much faster and cleaner.
2. I decided that refactoring all RESTful methods into separate classes was just *too* much of a refactoring job to do, so that’s why I didn’t go through with it during later extensions. However, if I had a choice to do everything over again, I would definitely extract all methods into their separate class.
3. Inconsistent use of error messages/error codes and error-data usage. It was just a mess all over the place.
4. The *InterestHandler* was a big mess. First of all, a big mistake I made was that any event in the future, like computing and transferring interest, should not only be handled by simulating time, but also in *real time*. This made the whole code a little trickier. This is also why both the InterestHandler and the TimeOperator are threads. However, once it worked, it worked like a charm (although it costed me a whopping ton of time).
5. Mentioned in (1), the use of *ServerDataHandler* could become completely obsolete if I were to utilize the RDBMS better, instead of reading and writing my own data in text/object files. Just one big mess and honestly glad I never had to look at it ever again.
6. Although now and then the tests were a very valuable time saver (for example when testing the Interest computation and transferring), the tests were a mess *too*. At some point (around extension 9) I just gave up on writing tests, as the refactoring I frequently did just caused errors in the tests which I couldn’t be bothered to fix.

# Valuable lessons I learned

I’m certainly going to utilize the tools at hand in a more elaborate and better way. For example, I will use the RDBMS better when dealing with *persistent data which doesn’t really feel at home in a database*.

Furthermore, when I see a huge class coming up due to non-scalable design choices in the past, I’d take action earlier. A great example of this is the *ServerHandler*. Despite having reduced the code by refactoring and optimizing methods, it is still a class of 2137 lines of code. There are currently 28 RESTful methods programmed in the ServerHandler, which is obviously too much if you think about any new methods that should be implemented in the same class. Without refactoring I’d see this class rise to 4k lines of code easily.

Another important lesson I’ve learned is to *heavily design* before you start working on any new implementation. In this project I’ve not really done this as a result of all the code being a mess from the beginning, which in turn was too demotivating to make a good plan before coding.

# Future improvements on project setup

From the beginning we certainly needed someone like David to help us all out like he’s done in this project.

Perhaps it’s best to not work in pairs. Even though a group partner can be handy in certain situations, it also slow down the overall progress of the work.

# What have I learned

I’ve pointed out most of the aspects of this project from which I’ve learned and what exactly I’ve learned.