Design by Contract Programming Techniques

Faculty of automation and computer science | TEHNICAL UNIVERSITY – CLUJ NAPOCA

GROUP 30423

Adrian - Radu Macocian

2018

# Assignment objective

Consider an application Design by Contract for processing clients and bank accounts for a bank. A hash map is used to store the bank accounts and the clients are the keys.

Secondary Objectives:

* Define a BankProc interface which will later be implemented by the bank class, so that we can add the pre and post conditions in the interface
* Define a class for Person and an abstract class for Account, which is extended by Savings Account and Spending Account
* The account may have a list of share holders, but only one main holder
* Implement an Observer Design Pattern which will inform the main holder of the account whenever something happens to said account
* Implement the Bank class using a Hash table. The key will be generated based on the main holder of the account, and it will store the accounts that said person holds. A person may own more accounts
* Use JTables to reveal any bank related information, such as accounts and people
* Define a well formed type method for the bank class
* Implement the bank class using the design by contract method (involving pre, post conditions, invariants and assertions)
* Implement a test driver for the system
* The data populating the bank class should be save and load from a file, so it has to be serializable

# Assignment analysis, assumptions, use-cases, errors

The Bank management software saves all the data in real time on the external file, so that if the program crashes everything is safe.

The use-cases are a little more complicated: once we run the application, we have to choose whether we want to run person operations, or account operations. If we chose to run person operations we then have a new window showing all the existing people using the bank, and the option to add a new person, delete an existing person or edit the information on one person. Choosing to run account options shows a list of all the existing savings accounts and spending accounts, and gives the possibility to add new accounts or to remove existing ones. The exceptional use cases are whenever the user tries to ignore an input field, or tries to pass invalid. It also give the option of editing existing accounts by adding share holders, changing the balance or by removing share holders. The other 2 options are to deposit or withdraw money from any account from the JTable. The exceptional user case is the very first time the software is run, since it tries to get the bank data from the external file which doesn’t exist yet. So the program must be made aware that the files is yet to be created. All the others exceptional user cases are handled by exceptions, which mostly involve filling the required text fields wrong.

# Projection

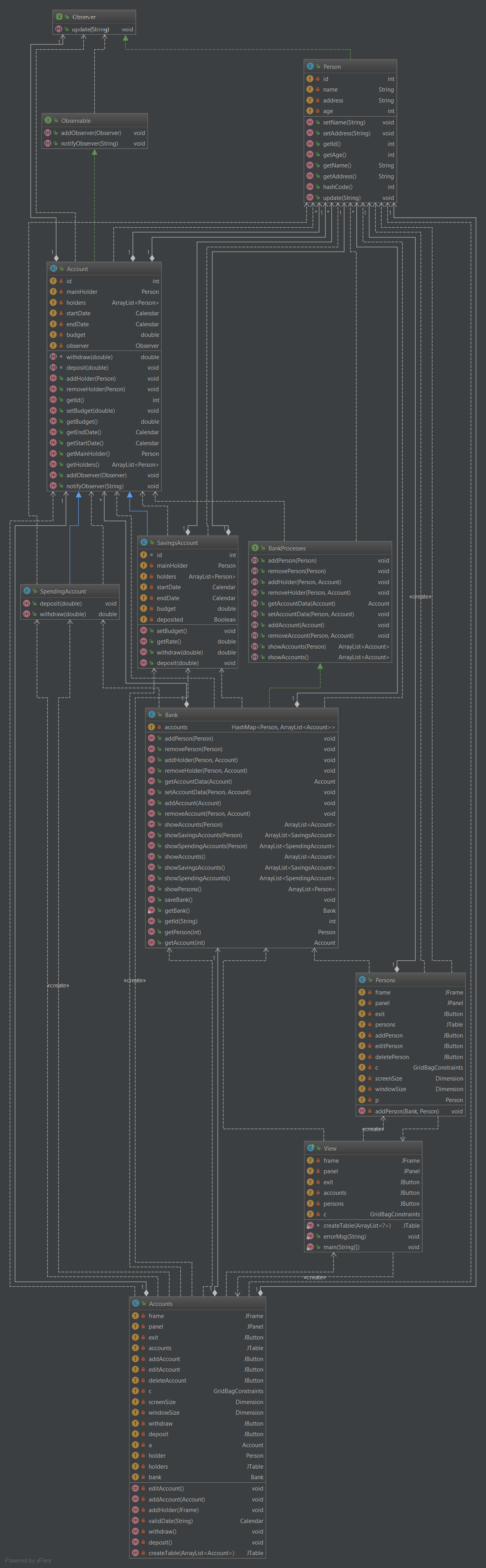
The whole project is split into 2 large areas: the model package and the GUI and 2 other smaller packages: the processes package and the observer package.

The model package stores the classes which would be equivalent to the data in the database so that they can be easily manipulated inside the application. It stores all the account classes, the person class and the bank class. The bank is the glue which keeps everything together, since the person and the account class don’t interact directly with each other.

The processes package store only the bankProc interface, which holds all the processes inside the bank class so that we can add the pre, post conditions and invariants

The observer package has the observable and observer interfaces defined by me. They are then implemented by the account or the user respectively.

The GUI layer is the what its name says. Except it actually is the heavy load of the whole the application. While the model and the processes packages have the complicated logic behind them, the presentation layer is the wall of text – brute force that runs the whole application. Anything the can be done inside the application is done in the GUI therefore it is handled by the presentation layer.



# Implementation

The whole graphical user interface uses grid bag layout for all of the only panels it uses. The resolution for the main frame is set with an absolute value on 640 x 480 pixels and the position is set for the center of the screen. All the other frames are set on different sizes based on their need. The main GUI class (the View) has 2 methods, one for creating reflexively the person tables, and one for creating the error windows in case the fields are incorrectly filled. It has the possibility to create through action listeners the 2 additional classes, which are also windows, the persons management window and the accounts management windows.

There are some buttons which reappear throughout the application which are stored as attributes of the class such as the exit button and the back button. The tables are updated by closing and recreating the window in which they are stored whenever a change occurs to their data

The other classes of the presentation layer are persons and accounts. Each of them does exactly what it says: create a new window and manage the persons or the accounts by adding, editing or removing one of the objects. Both of them create a new frame which is the same size as the main one, on which the attributes of the chosen class can be entered and then passed on to the business layer.

The main bank is the only needed object to be passed throughout the frames.

# Testing

The application is used to create a connection to a database and manipulate its data. The Junit tests are not efficient in this case since there is no complicated logic happening in the backend and everything can be tested and verified through the graphical user interface. The only possible tests would be checking whether the connection to the database and the data manipulation is accurate, which can be easily checked through the user interface or in other cases, it can be checked through a database management application such as mysql workbench.

# Conclusions

The system works as a bank simulator that is capable of storing a number of clients and accounts, and then communicate with it constantly in a fast and efficient manner (O(n)). The application is also capable of exporting a log for each of the actions that are taking place in an account as they are being created. It’s structured in an object oriented programming style having the 4 packages, each communicating and doing its own thing in order to create a fully bank application. Each class has its own functionality and is essentially vital to the program. The data is passed through them continuously so each of them has to do its job correctly. The whole graphical interface is done manually, and although I considered (and tried) creating the user interface without using a layout I came to the conclusion that the software would look much cleaner and would be more responsive if I did use a layout. Since I already learned some of the basics of the GridBagLayout I used it for this application as well. No helping software was used, so now I’m even more capable of finding any bugs and fixing them a lot easier since I know exactly why I did something or why something is in a specific place. I’ve learned a lot about how to update a panel and frame constantly during runtime so that the data shown on screen is always accurate.

The only problem I found with the program is that some of the time the tables would not update when changed. Because of that I had to completely close some windows and then recreate them to make sure that the tables are always up to date.

I feel like I’ve learned a lot about designing by contract while working on this assignment. I knew vaguely how to use assert in java applications but I had no idea on how to use them and could barely design a software without even considering using them. This assignment thought me a lot about abstract classes, interfaces and object oriented programming since I didn’t understand it before and I was avoiding it until I had to really use it just now. In this assignment I’ve also learned how to update a GUI, and make sure its data is always up to date, while executing it through and outside class. I can now handle the basics and I’m pretty sure that, if handled right, recursive programming can have a huge positive impact on a given software. I’m still unsure on when an error or warning will be triggered, that’s why I avoided creating the whole data access layer based on recursive programming.

One of the other big improvements in my coding abilities following this assignment is my ability to use the tables and their listeners to their full potential, I studied a lot about them while learning about mouse listeners. I’ve never had to use them before and tried mostly using the same entities as for a normal application. Being forced to write a program that uses them I spent a lot of time digging through all of its possibilities. And to my surprise this was not even that hard to learn once you understand the basic principles of listeners.

Possible improvements:

One of the biggest improvements I could bring to this program would be to find a way to use all of the methods created in the bank class and some of the methods created in the account classes since they are already there. Due to a lack of time I was unable to do this.

Another big improvement would be in the accessibility of the graphical user interface. It’s pretty hard coded in some sections and I would very much like to avoid, or to fix them. Also there are too many panels the user has to go through before reaching one desired action.

A big clarity improvement would be changing the layout in which the smaller input windows are created since they are pretty ugly right now, but I used them to save time which I used in implementing other great features. This is again not a logic problem but more of a lack of time problem.

Also one improvement would be that the log that’s created after each change in a account, could be created in a whole separate text file instead of just writing over to the console, and risking losing data in the case of an overwrite.

The bankproc class could also use some improvements since it could have some extra validations, pre condition, post conditions and invariants. Right now except for some validations to 2 of the methods in the bankproc class it’s methods are just given the ‘argument is not null’ pre condition, giving them a sense of uselessness.

The flow through the graphical user interface could also be improved. While most of the time I tried to keep the possibility of going back or canceling an action, in some cases it would just ruin the whole look of the user interface so I chose to abandon it. Together with this improvement to the user interface I could also add more color to it. Right now it looks exactly like a java low-level beginner interface, and although it has a lot of panels it could still be changed to look a lot better with the right kind of imagination and time.

# Bibliography:

Most of the project was done by myself with the help of the laboratory professor, who gave us some ideas on how to use threads and on how to make our application as thread safe as possible. The only two external information sources were used for the creation of the graphical user interface and for creating the division of polynomials:

<https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html>

For a debugging and problem solving there were a lot of searches on stack overflow but I can’t pin point exactly to which url’s I’ve used:

<https://stackoverflow.com/>