**Design by Contract Programming Techniques**

***-Project documentation-***

Student: Cocoş Tudor-Adrian

Group: 30423

**Contents:**

1. Objective of the assignment
2. Problem analysis
   1. Assumptions
   2. Modelling
   3. Scenarios
   4. Use cases
3. Design
   1. Design choices
   2. UML diagram
   3. Class design
   4. Algorithms
   5. User Interface
4. Implementation
5. Testing
6. Results
7. Conclusions
8. Bibliography

**1) Objective of the assignment**

The main objective of this assignment is to design and implement an application for a banking system, using Design by Contract Programming Techniques.

The secondary objectives are the steps needed to be taken to achieve the main objective. First of all, we need to analyse the concept of a banking system and what it implies: persons, as clients, possessing accounts. After that we need to break down the problem to implementable features like classes and methods needed. The next steps include class design, method implementation, User Interface development and testing. Finally, the results are evaluated, and we can think of further improvements that can be made to the existing project.

**2) Problem analysis**

*a) Assumptions*

Given the assignment, we assume that the application should have 2 separate views: one of the clients and one of their accounts as the data fields that each type of view is holding are totally different.

*b) Modelling*

To make the data as simple as possible there are a few fields to describe each entity. A client is made of a CNP (personal identification code), a name, age and address, while an account is defined by an IBAN (the bank account identification code), an account type (Savings Account or Spending Account) and the amount of money that the account is currently holding.

*c) Scenarios*

The main scenario is when it is only deal with the appropriate data and when there are no logical or data integrity problems that might be faced. This includes making no mistakes when inputting the fields of any entity for any type of action that is performed on the respective entity. As a result, the right data will be placed and represented in the correct spot at all times.

Other scenarios start with human error in inputting the data in the fields of the GUI. If one field is left out or the data inputted is not valid, it will be signalled by the application. Also, by implementing the program using Design by Contract, the data will be tested to see if it passes the pre- and postconditions for every type of action: for example, when adding a person, we check if it really is a new person or that one is a client already and can be found in the bank “database”.

*d) Use cases*

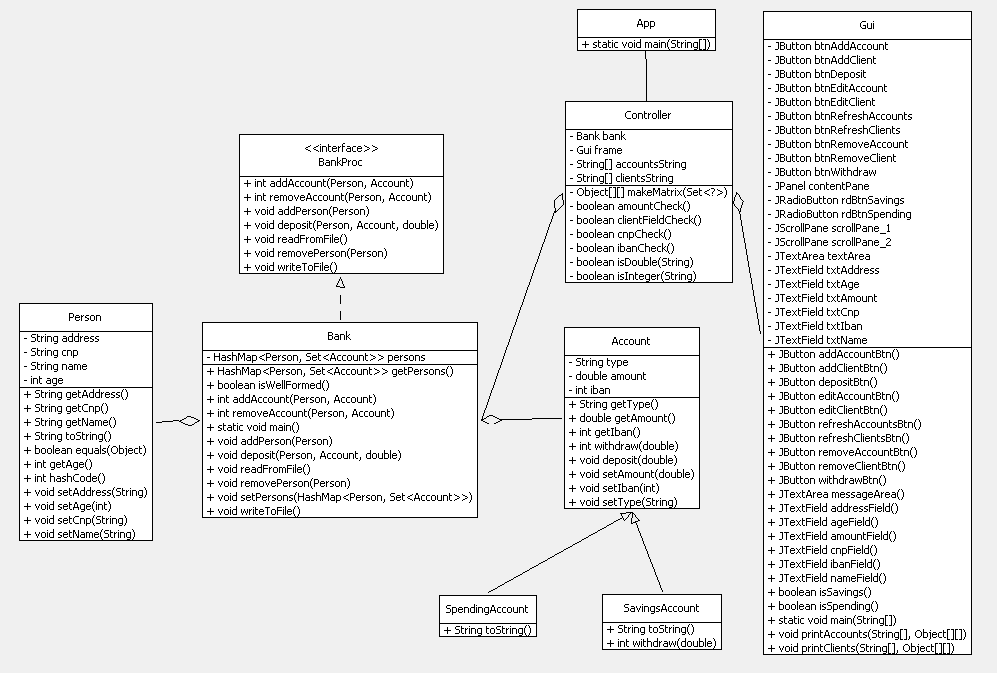
The application may be used to input the personal data of a new client, updating some of his personal data or, if he or she is no longer a client of the bank, to be able to delete his account. When considering the accounts that a certain client may hold, the user should be able to open a new account, update the amount of money inside of the account manually if needed, and finally to delete the account if the client or bank chooses to do so. When opening an account, the user may choose between Savings Account and Spending Account. The difference between the two of them is that when you open a Savings Account you need to start that account with a certain sum of money (essentially, entering the savings program from the creation of the account), while this criterion doesn’t apply to the Spending Account, which is just opened with a 0.0 RON sum inside. As another method to be differentiated, when the client withdraws money from the Savings Account, he is taxed extra for taking money off a savings scheme.

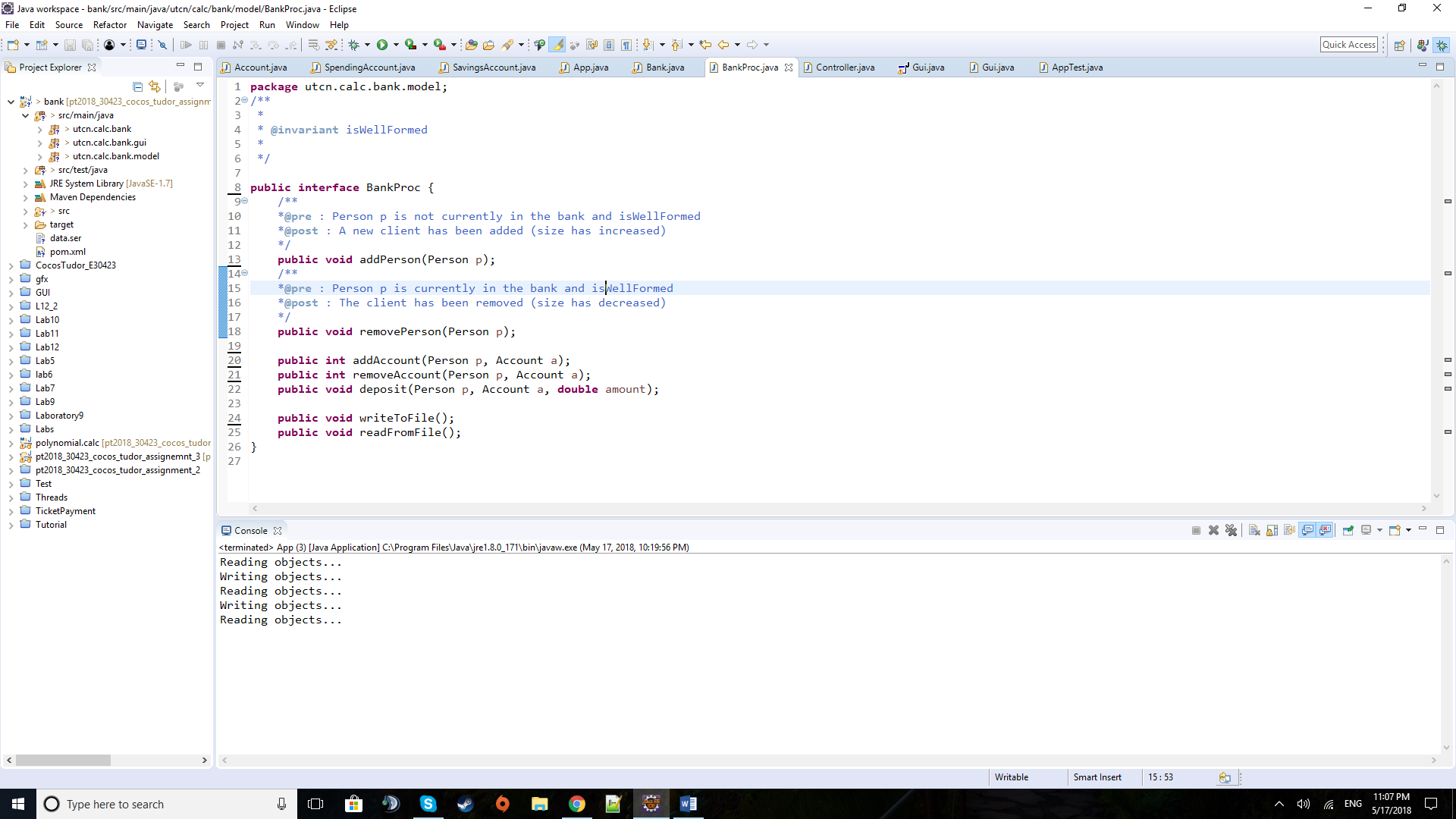
**3) Design**

*a) Design choices*

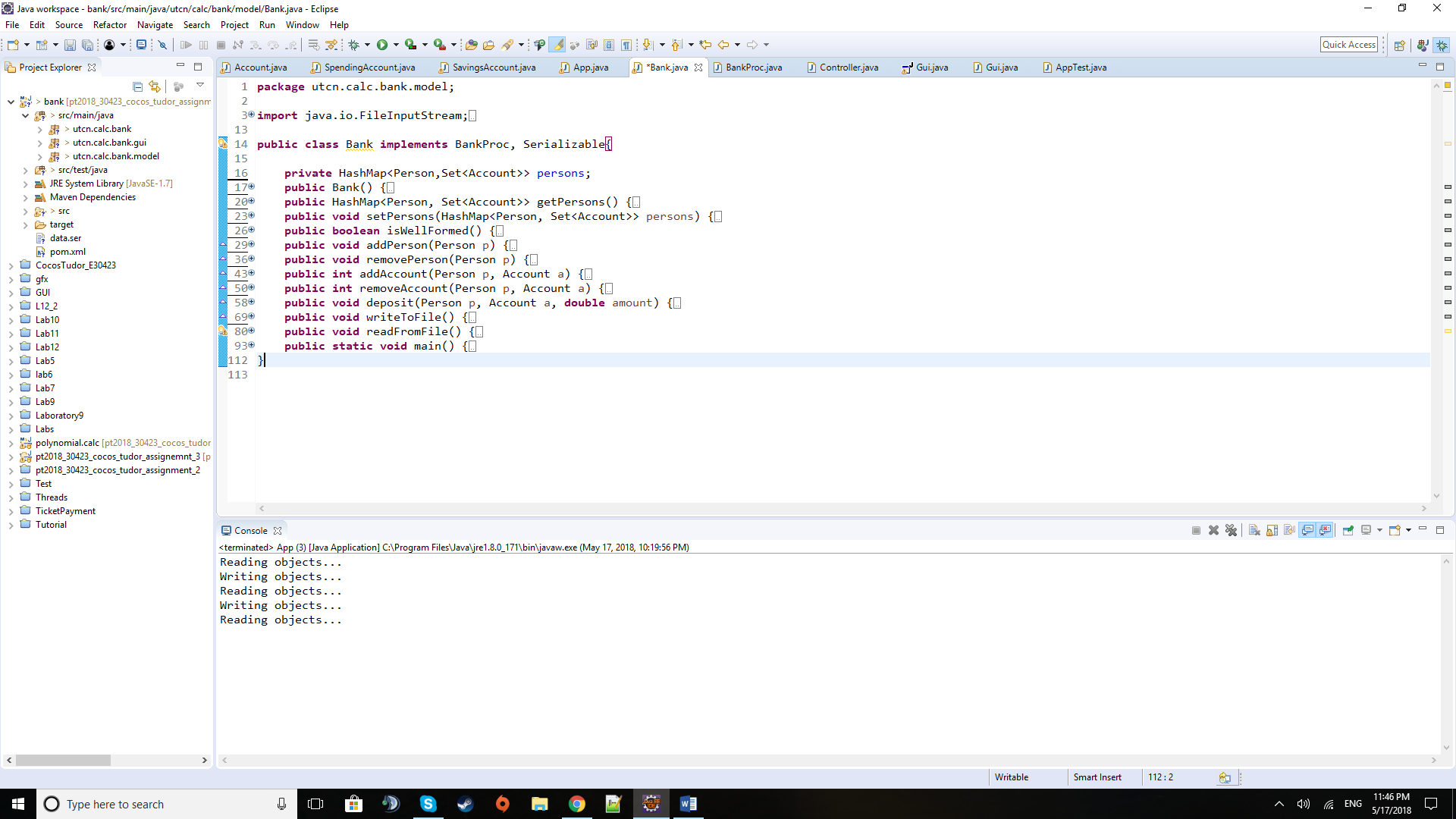
The design of the application starts with the diagram provided in the homework description, to which the graphical part is added, followed by the requirements for the task at hand.

The Bank class holds the data of the bank in a Hash Map of Persons, each with their own Set of Accounts. Here we can find the methods of adding/removing a client and editing their personal details. Additionally, there are methods defined to open a new account, edit the amount of an existing one or removing an old account. For making it possible to save this data to a file from which it is read afterwards, another two methods were defined specially for this kind of task. The headers of all these methods are found, of course, in the Bank Proc interface. The Person class is where the fields of a client are defined and found. While the same is said for the Account class, it is extended by two additional classes: Savings Account and Spending Account where we can find overwritten methods intended to be used to better differentiate between the two kinds of accounts.

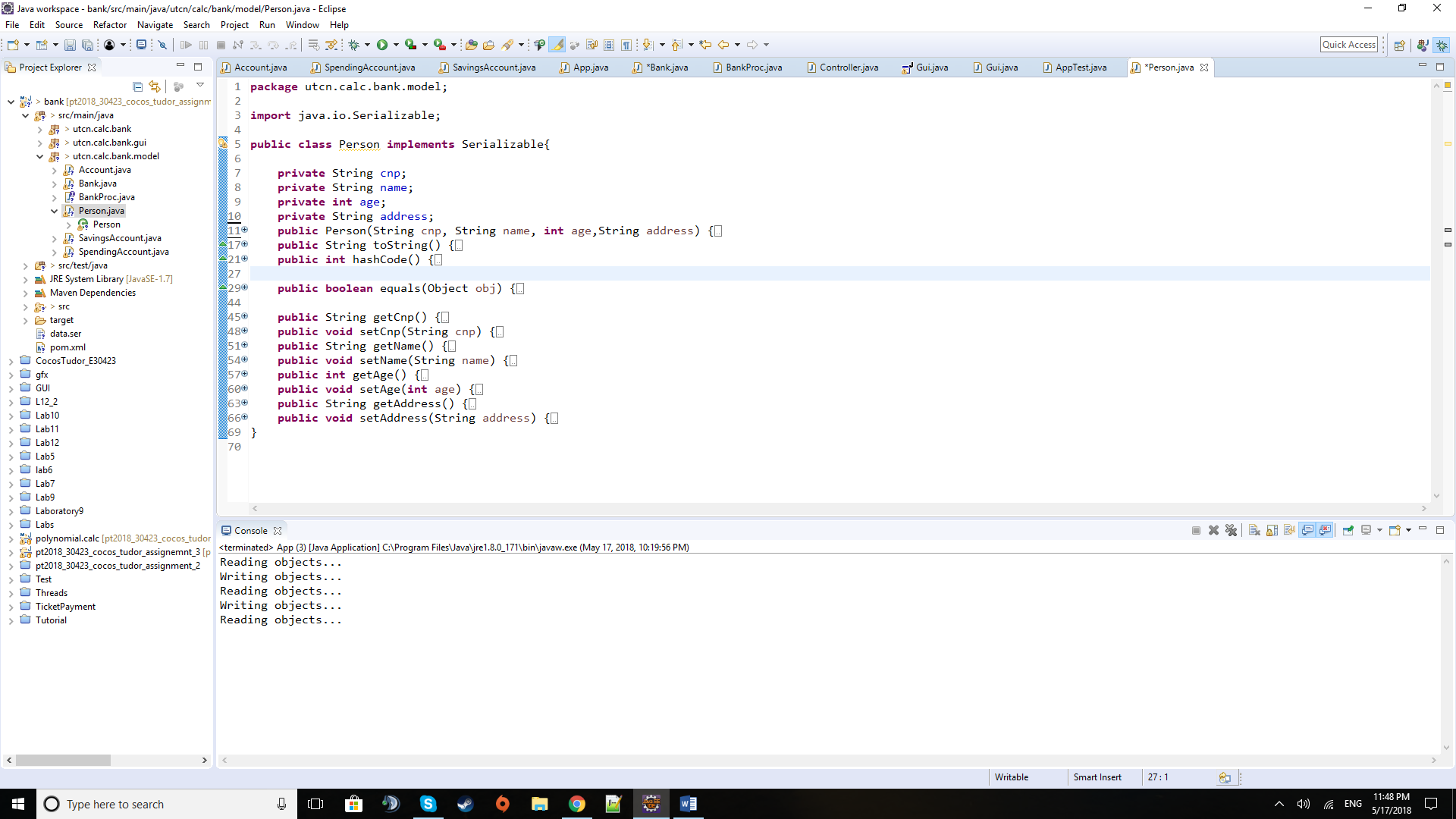
*b) UML Diagrams*

*c) Class design*

***BankProc class***



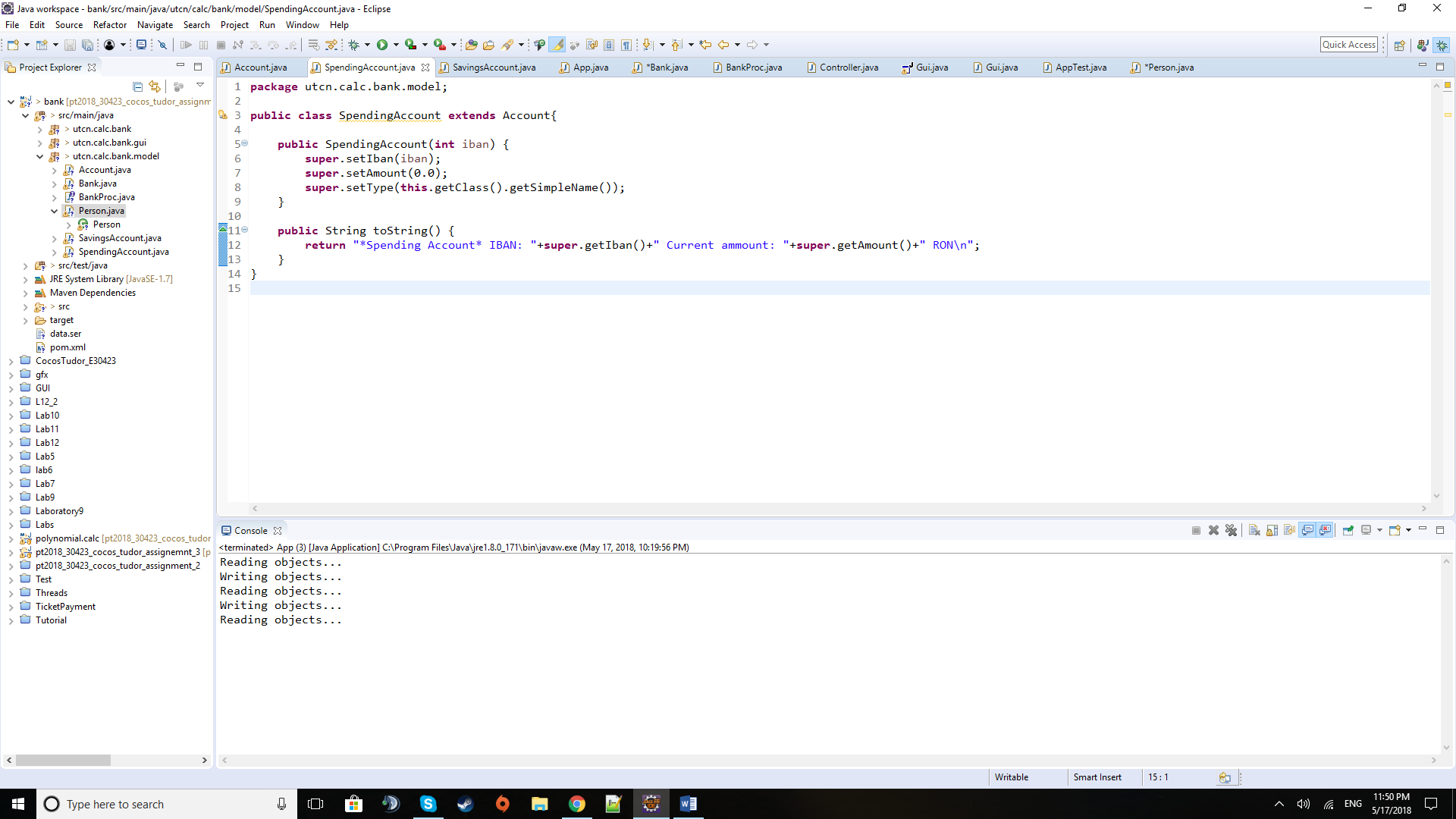
***Bank class***



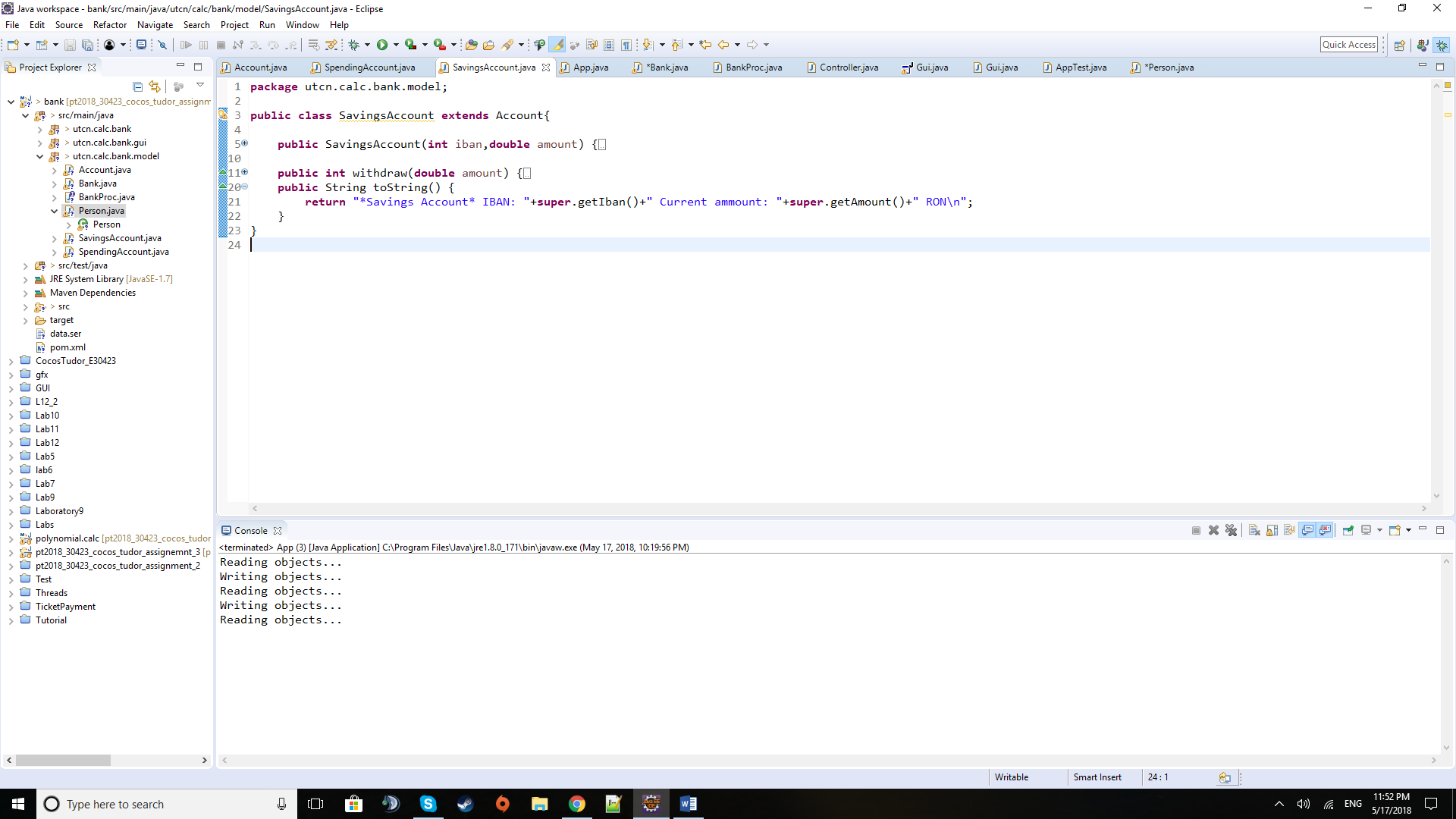
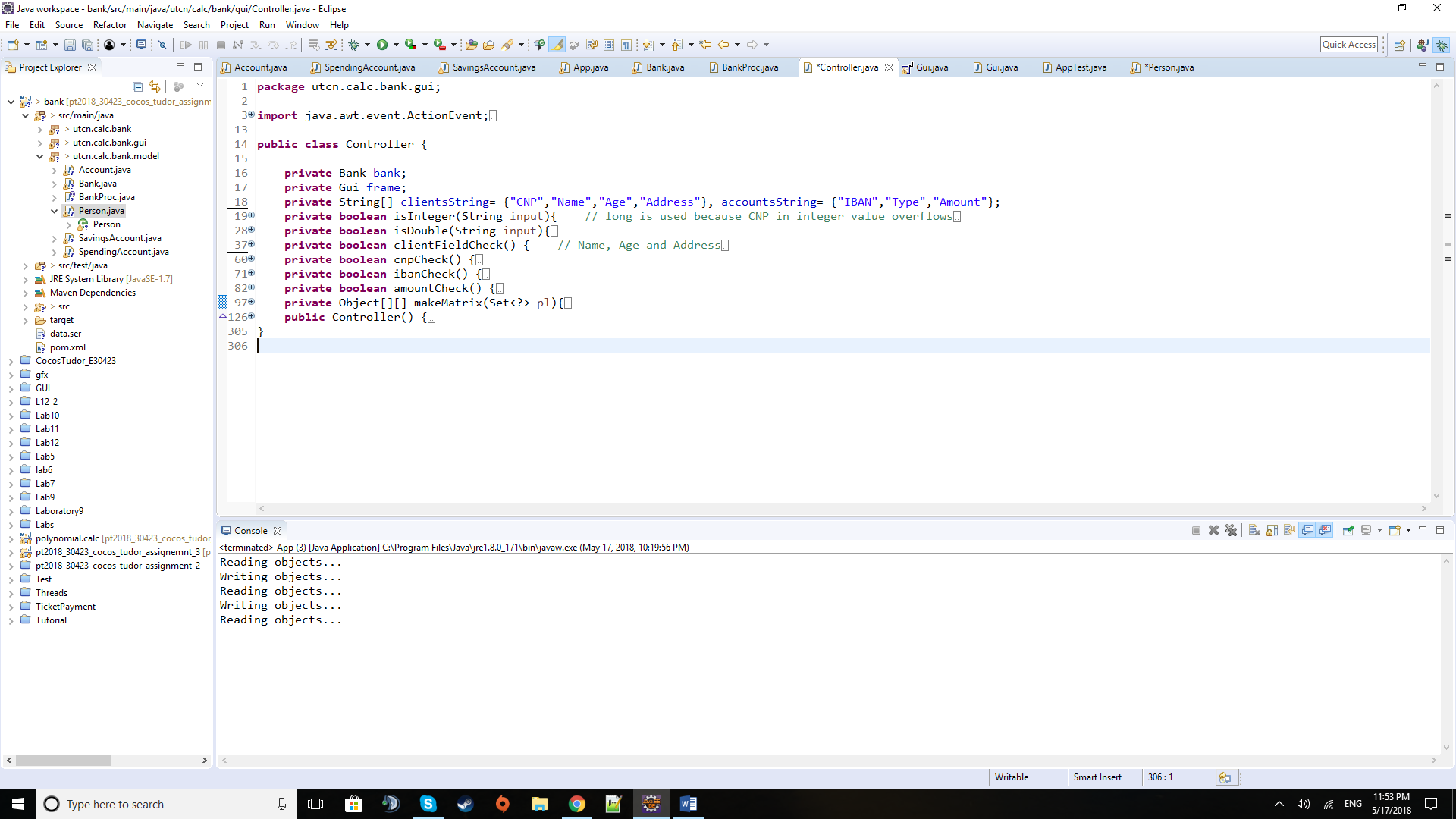
***Person class***



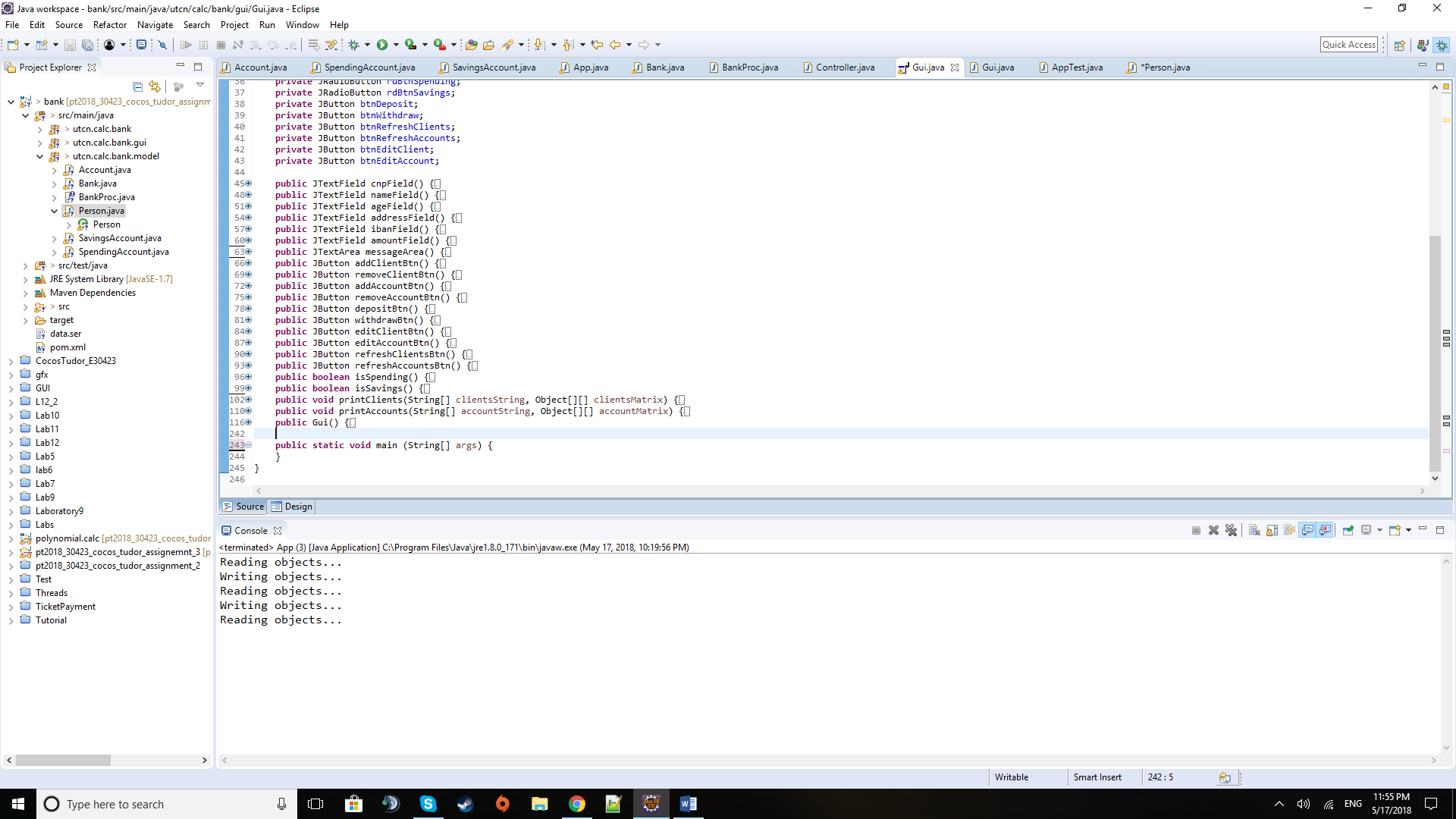
***Account class***



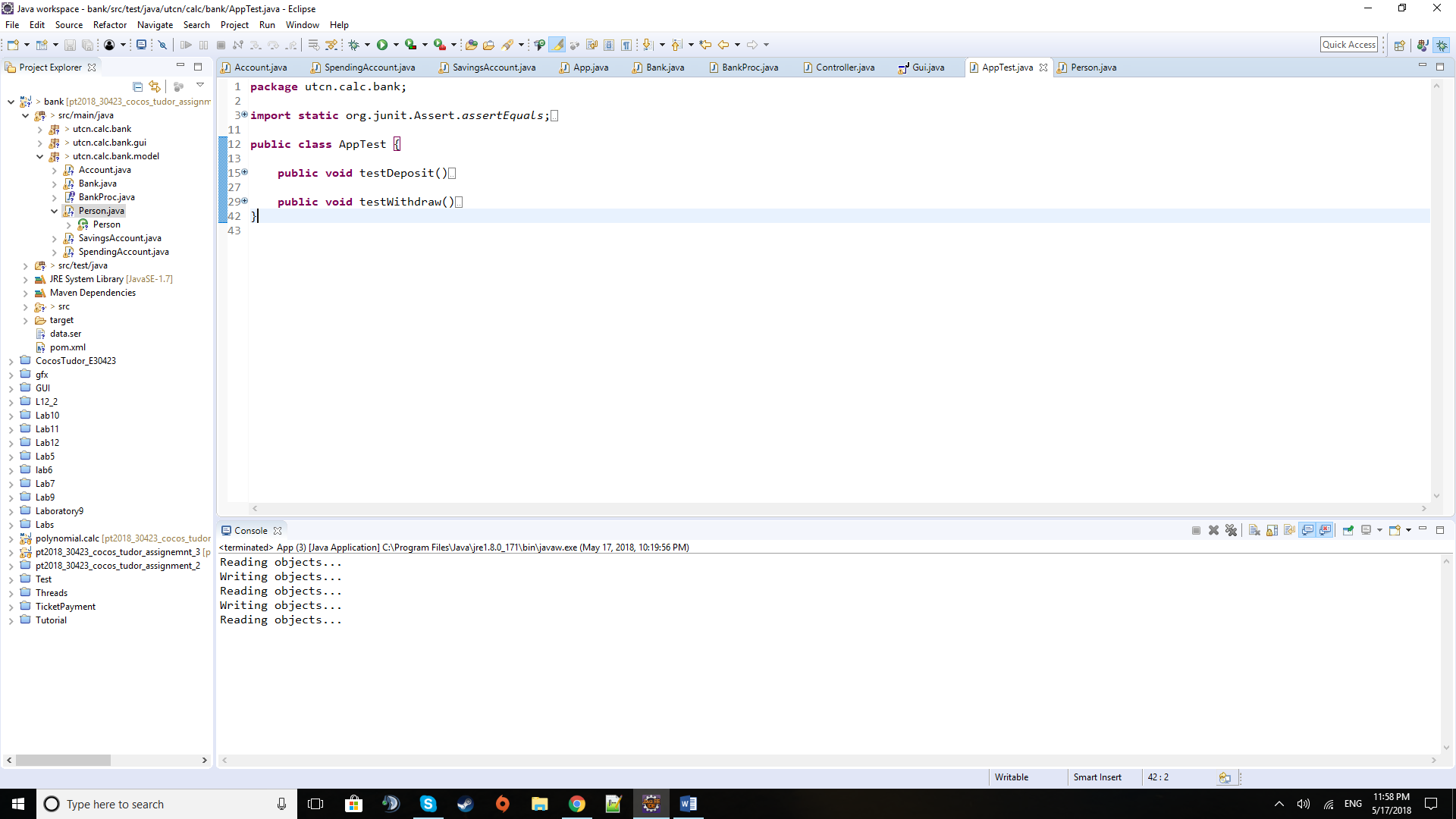
***SpendingAccount class***

***SavingsAccount class***

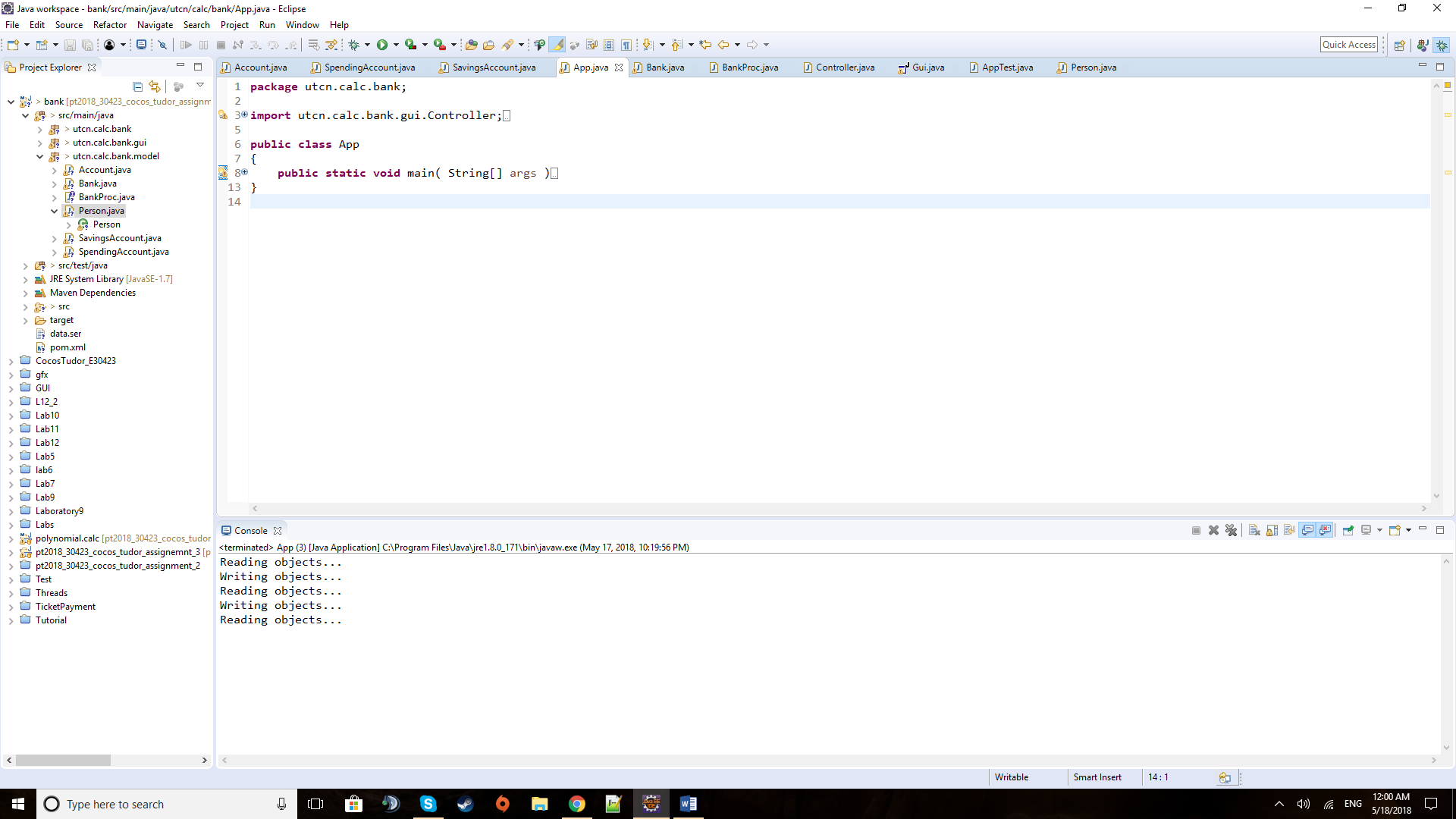
***Controller class***



***Gui class***



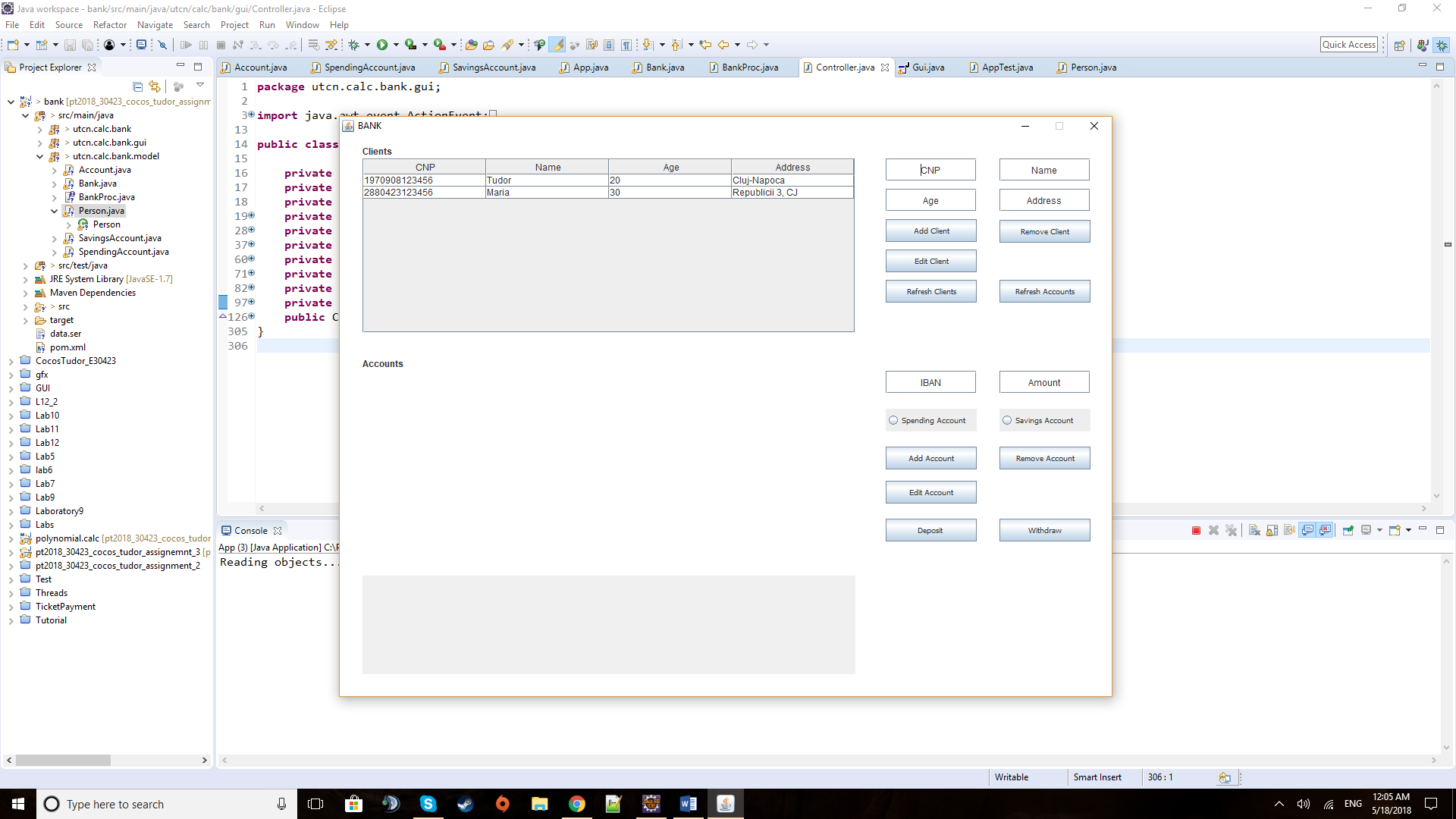
***AppTest class***



***App class***

*d) Algorithms*

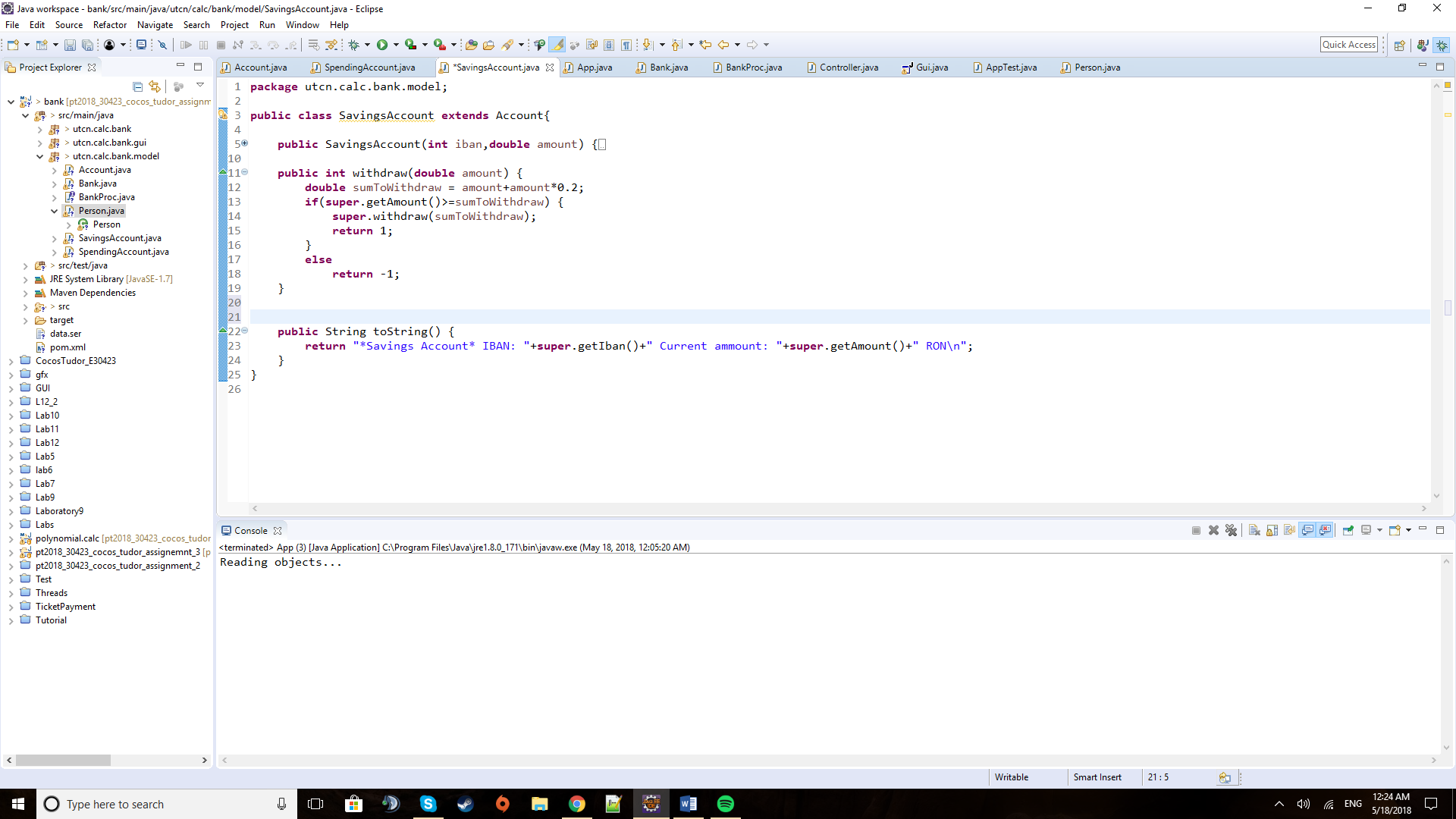
As for algorithms there are no special mathematical-based algorithms like in the polynomial processing assignment. The usual ones imply iterating through sets of clients ( persons ) or accounts, but the special ones this time are the ones used for hashing a certain entity using one of their fields and the pre- and postconditions that need certain boolean retrieving methods as they are implemented using assert. Reflection of class fields is also used in order to obtain the data for the tables that are displayed with information regarding persons that are clients of the bank, as well as their accounts. The algorithms will be presented and explained in the Implementation section below.

*e) User Interface*

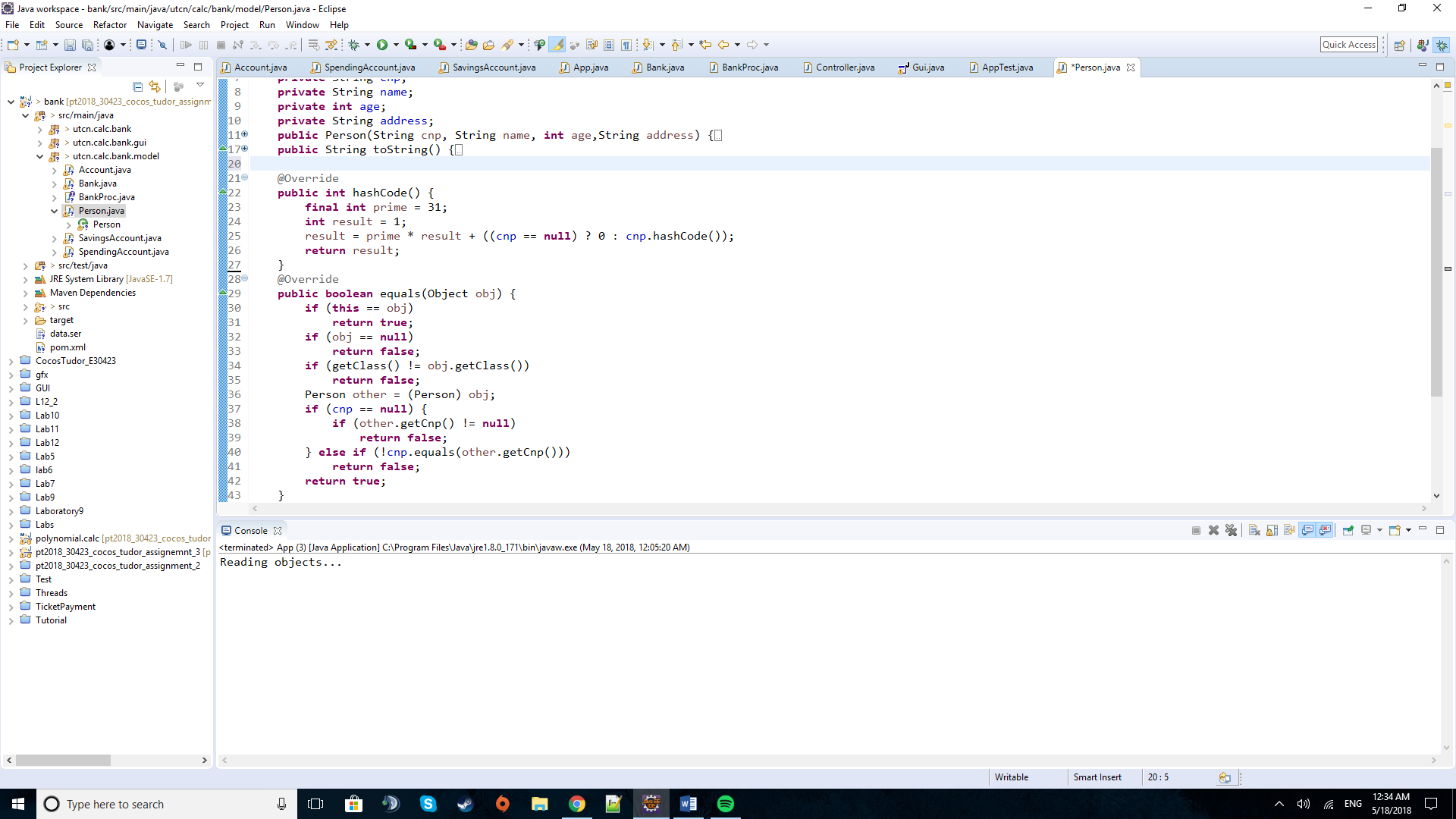
This is the window that the user is prompted to when the application is started. It contains several types of graphical elements, each with its own specific use. On the left, going from top to bottom, there are 2 tables and one text area. The two tables, as the two labels on top of each one of them suggest, display the appropriate data from their respective entities in the right fields. When the application is started only a list of all the current clients of the bank, that is read from the file, are shown. If you insert one of their CNPs then you will be able to display the accounts linked to their unique personal code. In case something happens, the user is notified via messages that are displayed in the grey text area below. On the right side of the window there are groups of text fields and buttons used to interact with the application. The top four text fields are used for the personal details of the client that will be added or removed, or of whose personal details may be subject of change via the edit option. The refresh clients will reinitialise the table of clients for the current bank data while the refresh accounts will refresh the table of accounts for the client who is chosen by typing his or her CNP in the appropriate field. The two text fields in the middle are used for inputting data for the accounts. They are complemented by the two radio buttons that are used to select between the two types of accounts that can be created for this given bank type. The cases when the user may forget to select one or may select both of them at the same time have been handled, so one must select only one type once. The last buttons are used for depositing and withdrawing money from an account for which we have inputted the CNP of the person and the IBAN of the account.

**4) Implementation**

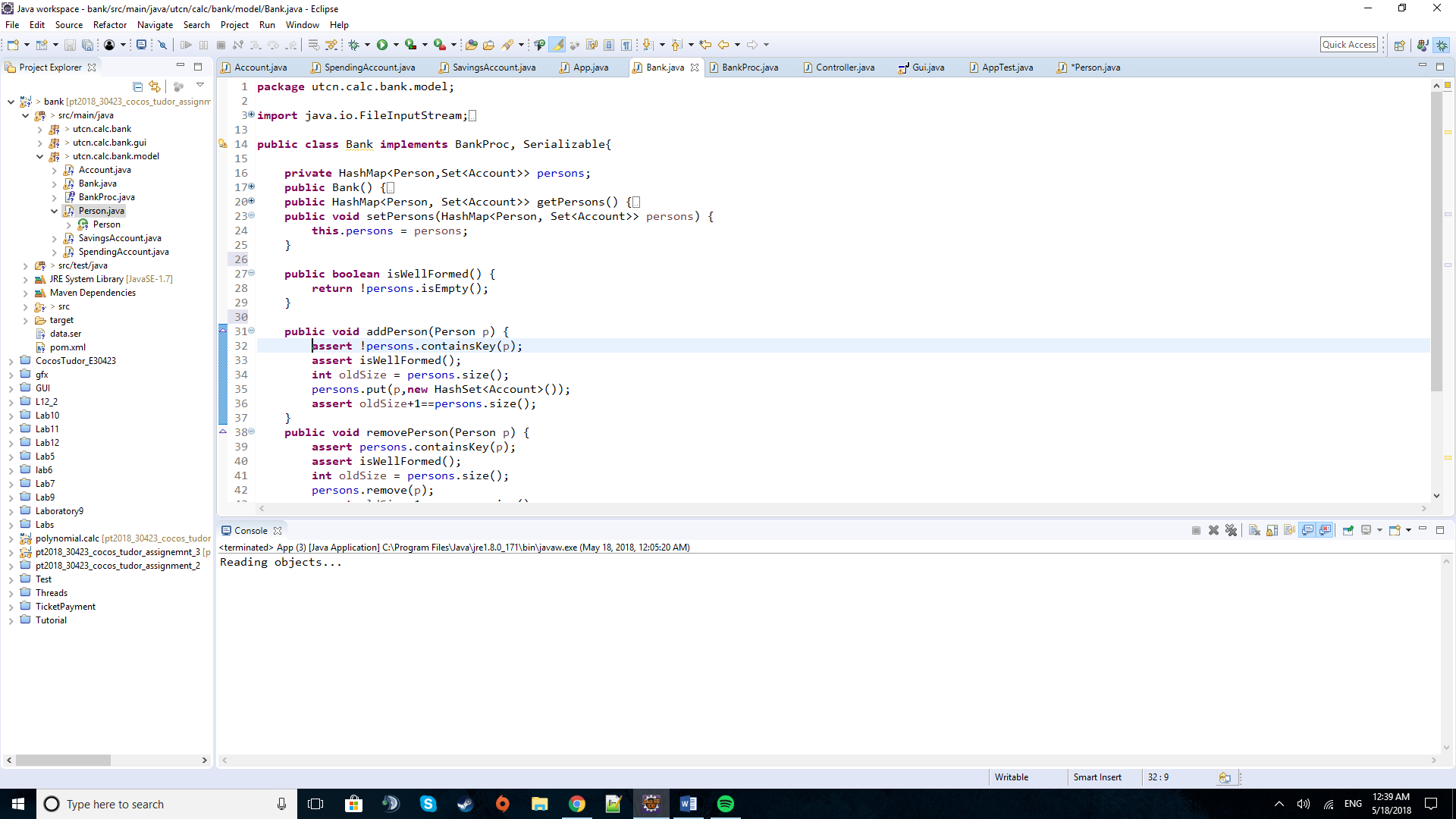
I will describe here the most relevant pieces of code and especially those that are not that common and straight-forward (unlike getters, setters, and some basic constructors).



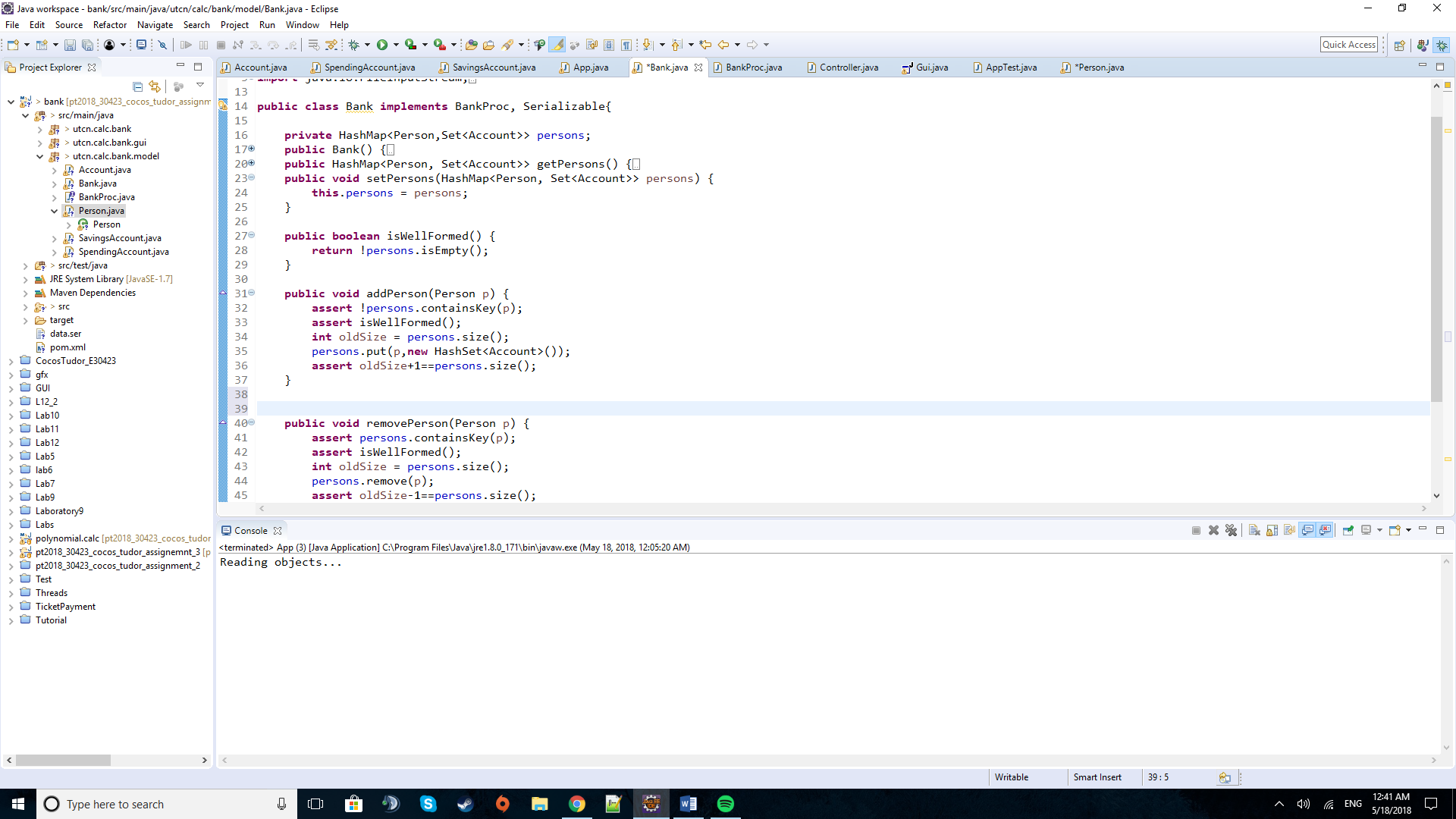
The method above is used in the Savings Account to tax the clients extra for withdrawing money from savings schemes. This is one way in which the two types of accounts are differentiated.



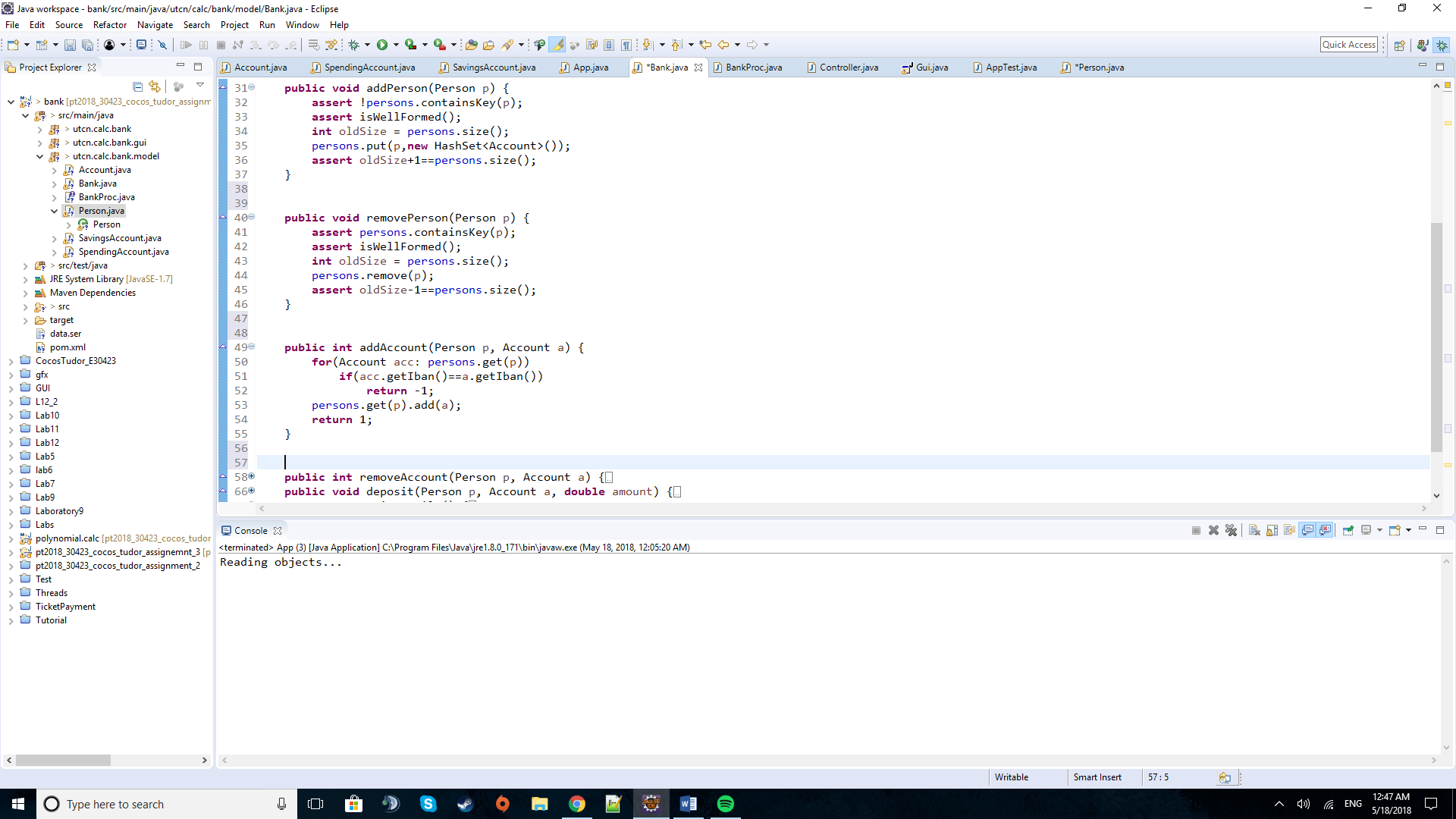
These are the methods of hashing and equals for the Person class. They both centre around the CNP of a client as it is unique for every person (in Romania, at least. Other countries have Social Security Numbers or other types of identifying a person by a unique feature).

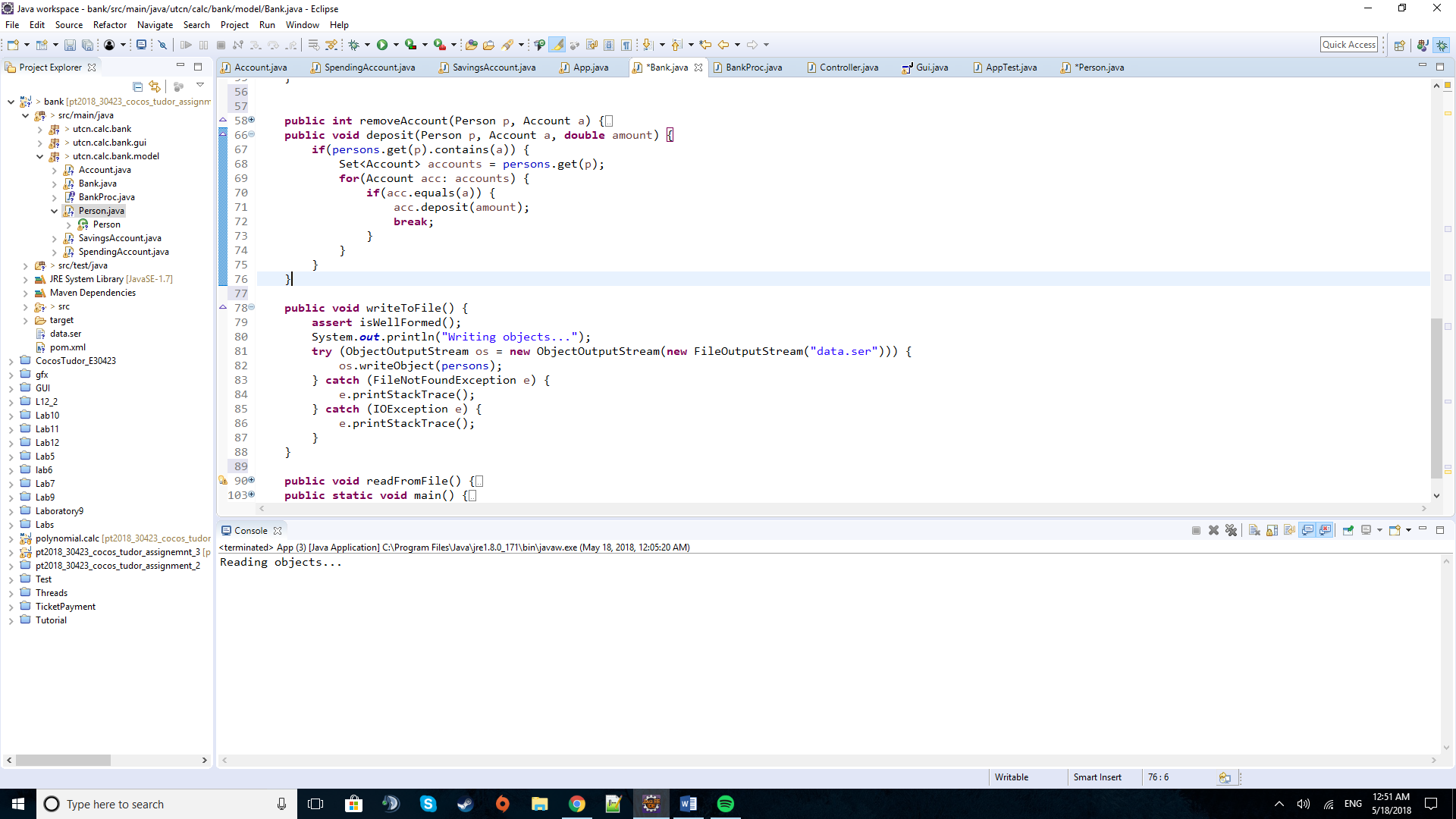


This is the method to check if the bank class remains “well formed”. In my case, I just chose to check if the persons hash map still exists and has at least one person, but there may be many ways this could be implemented.

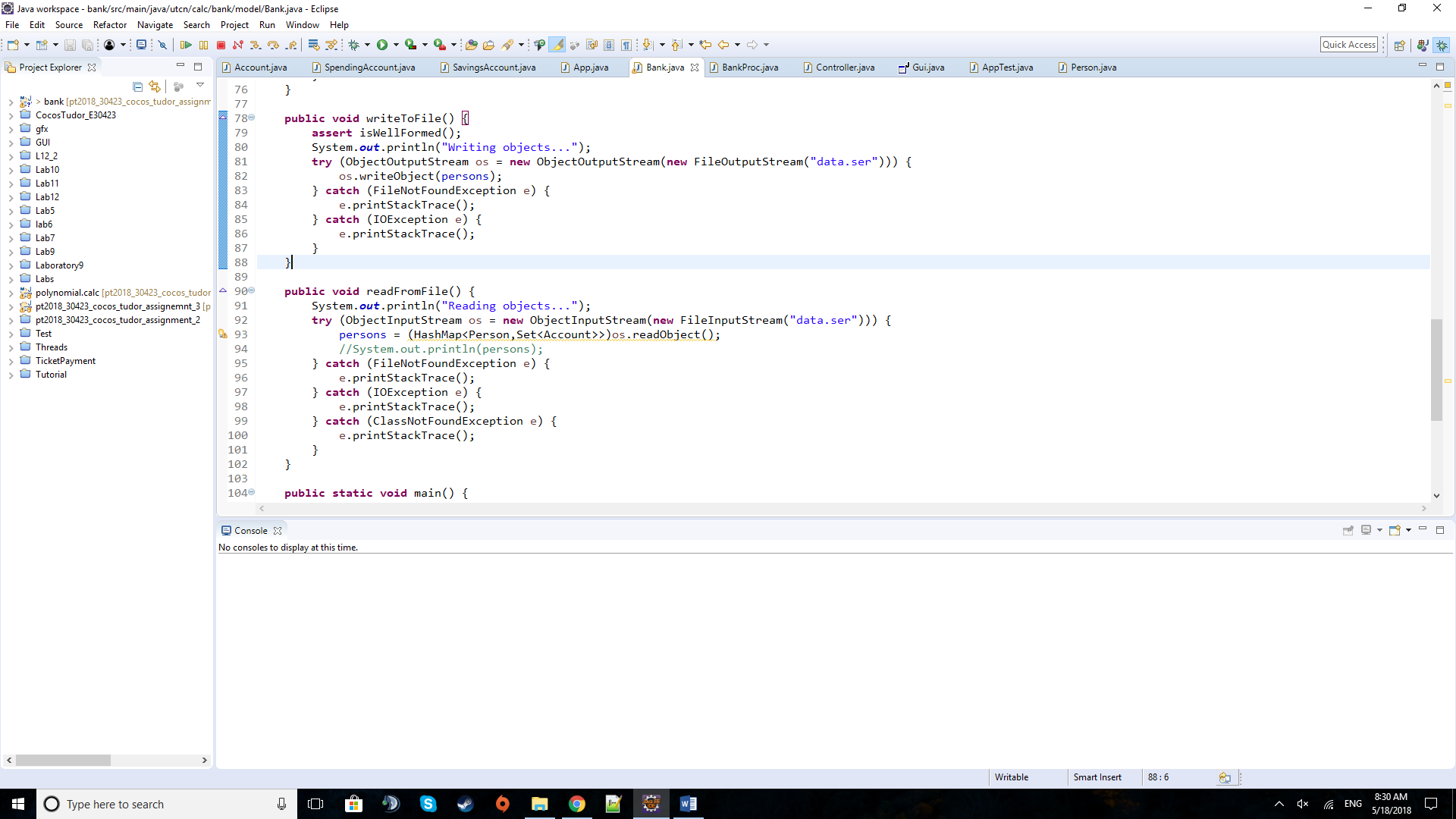


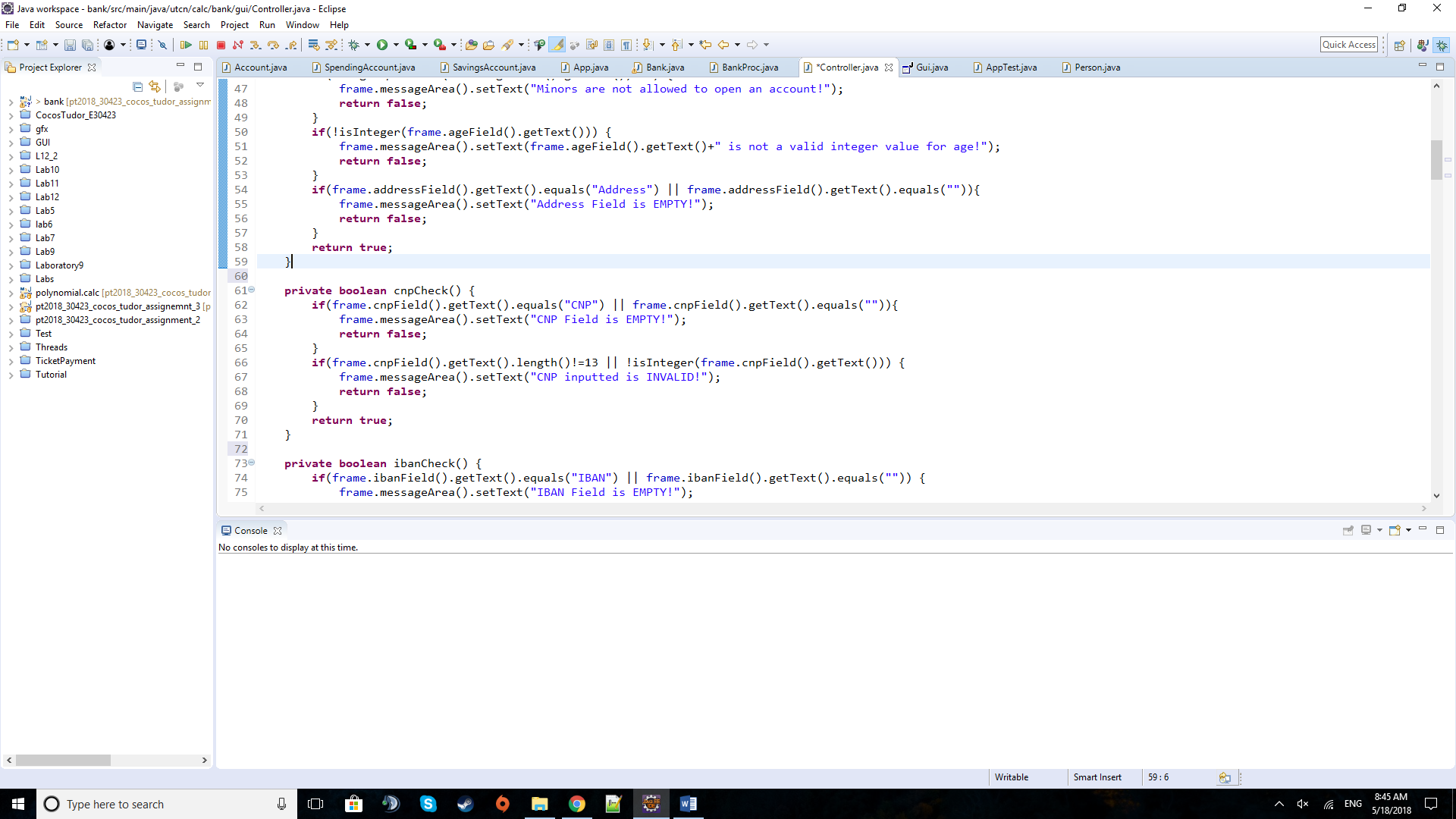
When adding a person, we check as a precondition if the person is already in the bank data base and if the data base is well formed. After that we insert the new client with an empty set of accounts and in the end check to see if it was really added by simply comparing the old and the new size of the hash map.

In a similar fashion, the removal of a person is done, with minor tweaks on the conditions and checks.

The opening of a new account is done by firstly checking if the given person already has an account with this IBAN. If it passes, the account is simply added to the set of accounts of the client.

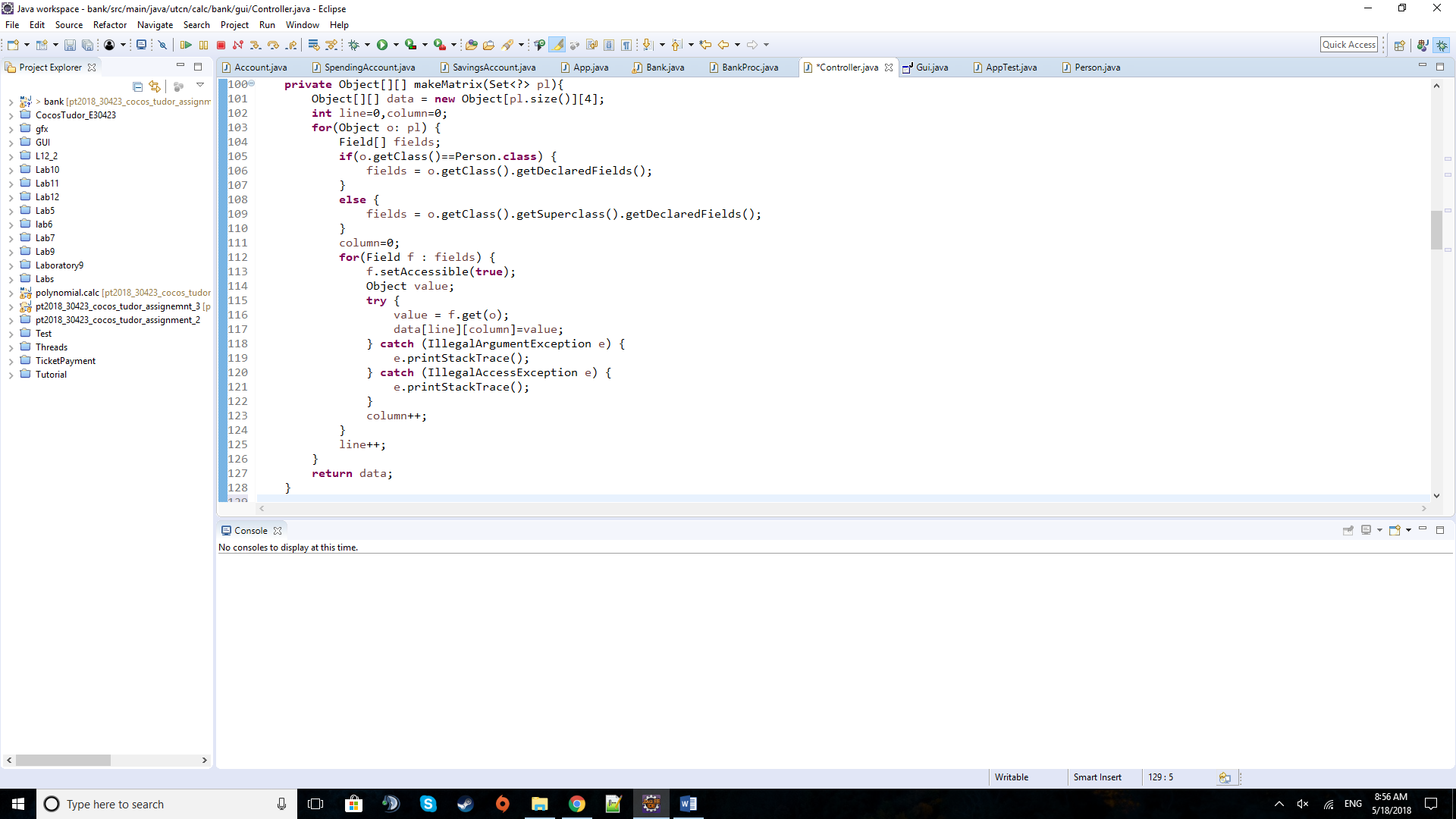
The method above is the one used to write the whole hash map to the data file by the way of Serialization.

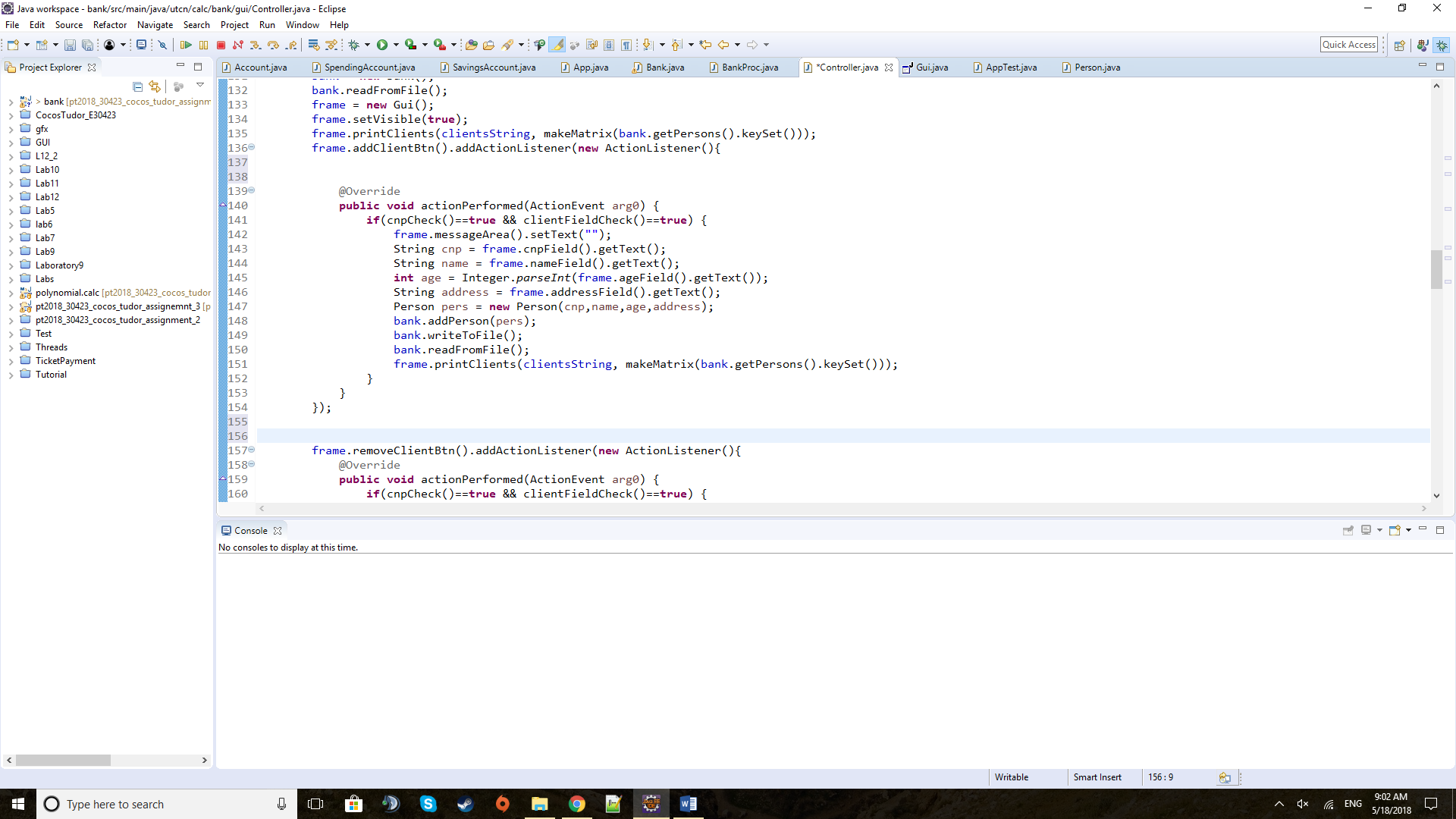


The method for reading the bank data from a file is really similar to the writing one and overwrites the previous hash map with the one in the file.

The method above is used to check if the text field for inputting the CNP has been, firstly, edited by the user and then if it represents a number made of 13 digits.

In a similar fashion the other fields are checked in order to alert the user if he or she forgot to input one of the details for a person or account or if one of the text fields holds inappropriate data for the required entity field.

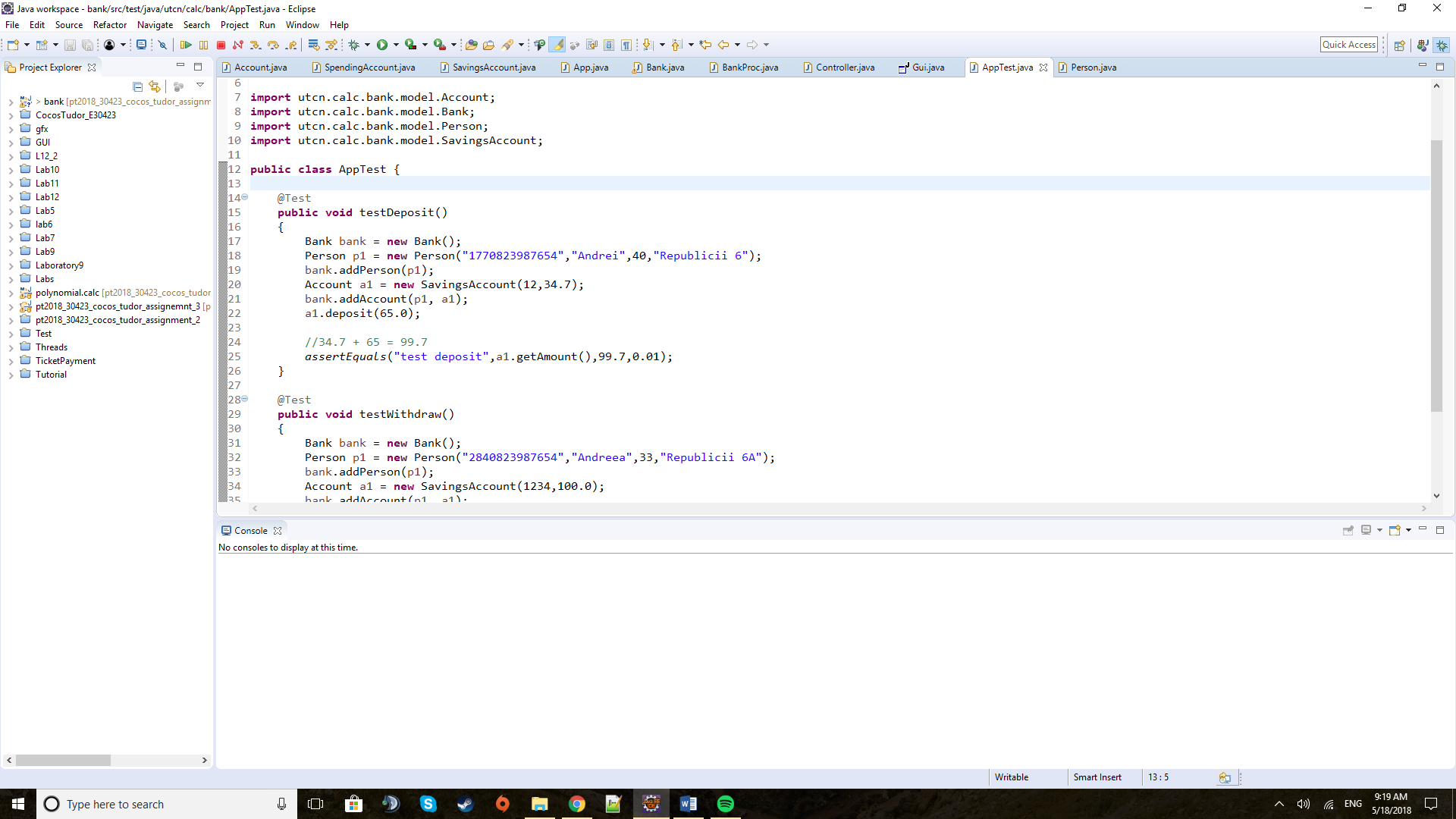


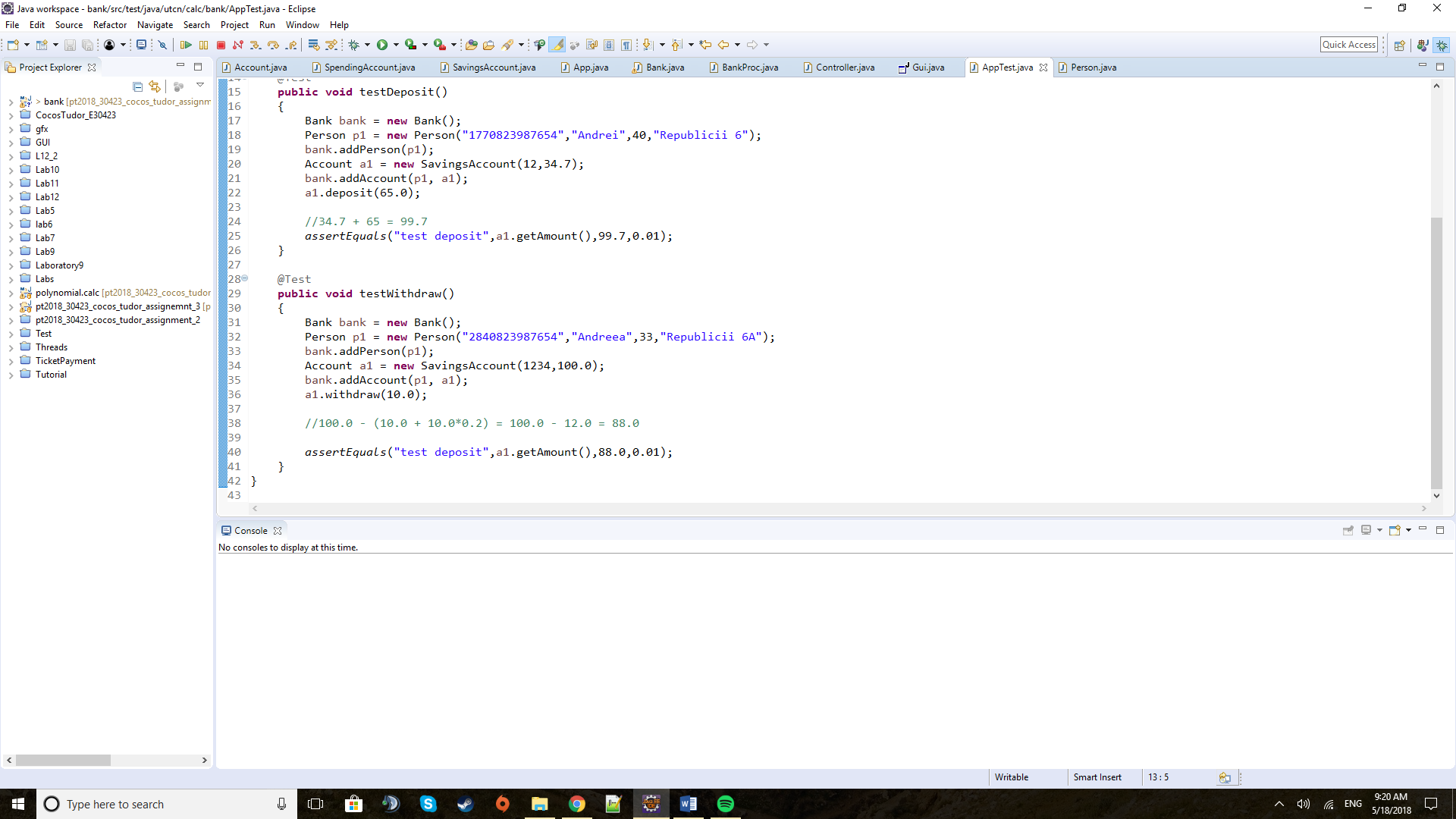
The method of make Matrix is taken from the previous assignment. It uses reflection to take a set of a certain entity and generate the matrix of objects needed to be placed in the tables of the respective entities. If the set is made of Persons, then the Object types are of that class but if the set is made of Accounts then we need to get the child classes of Spending or Savings Account.

The method above is one of the Action Listeners. It responds to the pressing of the Add Person button. After checking the fields for appropriate data, it constructs the person and calls the Bank method of adding a person to the data base. After that, it backs-up the changes made and overwrites the bank hash map of clients and their accounts. The last line refreshes the clients table, showing the new client too.

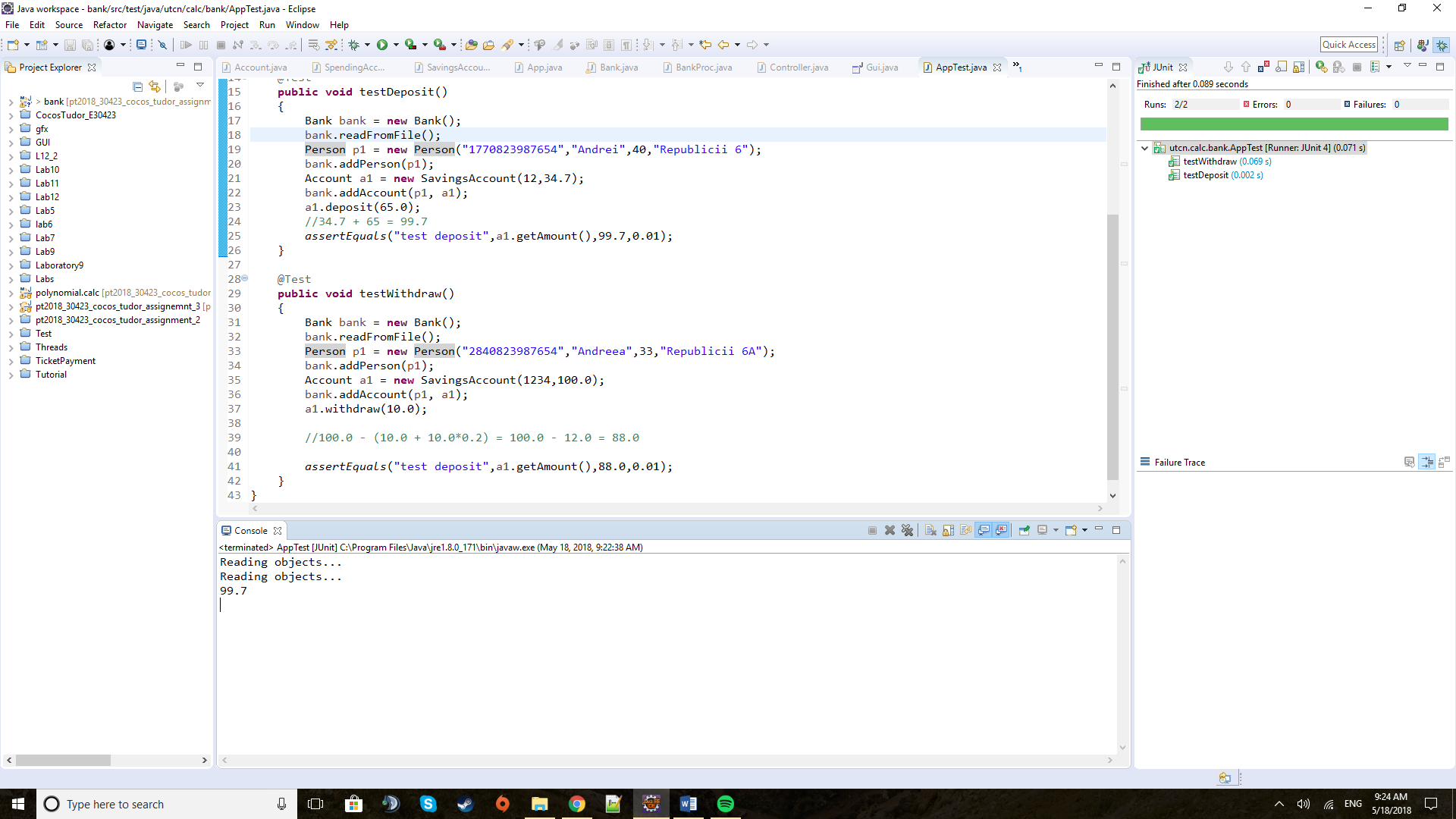
**5) Testing**

The testing for this project was foremost done by running the application and testing the application for any kind of use cases that may appear by human or program logic error. In the end, as a requirement was asking for, a test class was made. It tests for example if the application deposits and withdraws the appropriate amounts from a given account.





**6) Results**



The results came as expected, having above, in the print screens taken of the test methods, the explanation for the expected results in commented lines.

**7) Conclusions**

This assignment proved to be useful in learning how to implement Serializable and also how to use hash maps and sets.

As further development there might be a date system implemented. It will make more sense then to tax more the clients that withdraw money from a savings scheme only for a limited time span like 6 months after opening the account. After that he or she may be able to withdraw once per month any amount without commissions for example. Also additional fields may be added to both clients or accounts.

**8) Bibliography**

Tutorials from: <https://www.caveofprogramming.com/categories/java-collections-framework/index.html>

And : <https://www.caveofprogramming.com/java-video/transient.html>

For any issues regarding exceptions: <https://stackoverflow.com/>

For details about the methods and fields of predefined classes: <https://docs.oracle.com/javase/8/docs/api/overview-summary.html>

An example used to generate a simple JTable and place it in a JScrollPane: <https://www.thoughtco.com/how-to-create-a-simple-table-2033894>

For the UML diagram I used an application: <https://sourceforge.net/projects/code2uml/>