Technical University of Cluj-Napoca

Faculty of Automation and Computer Science

Computer Science Department

Year 2015-2016 Semester II

Programming Techniques

*4rd Project*

Student: Dan Toderici

Group: 30422

Contents

[Project specification 3](#_Toc449453103)

[Problem Analysis 3](#_Toc449453104)

[Modeling 4](#_Toc449453105)

[Use Case Diagram 4](#_Toc449453107)

[Design 5](#_Toc449453108)

[Relational Diagram 5](#_Toc449453109)

[Class Structure 7](#_Toc449453110)

[Implementation 8](#_Toc449453111)

[HashMap 8](#_Toc449453112)

[Design by Contract 9](#_Toc449453113)

[Bank Serializer 10](#_Toc449453114)

[Conclusion and further developments 11](#_Toc449453115)

[References 11](#_Toc449453116)

# ****Project specification****

Propose is to design and implement a system that simulates a bank management application for processing customer orders. It is going to be just like a bank database implementation with its specific functionalities and data.

The application, that resulted from design such system, can perform 7 main operations:

1) Insertion of a new client

2) Adding a new account for the client

3) Deletion of accounts

4) Deletion of clients

5) Creating a .ser file for saving changes

6) Depositing money into accounts

7) Withdraw money from accounts

6) Display clients into JTable format

7) Display all accounts for each client into JTable format

However, all these operations must be executed based on user input. Therefore, a graphical user interface is needed. The UI must provide the user methods to insert orders and to choose which operations he or she desires to be execute. Adaptive view was not required, and only admin will be allowed to view the bank info.

# Problem Analysis

Splitting the problem into smaller parts is the first main problem. Due to the need of persistence and further requirements, a package which operates on a .ser file is needed. Not to forget about designing a proper structure, this will help us much more than an xml or a basic file and it saves data easier ! Moreover, there should be extra logic for inserting or not into the .ser file or not and also to check if the input is okay or not.

Operations on the database, as expected must be in conformity with the Data Access Object pattern (DAO); this will be explained later when reaching the implementation and considerations chapters.

Views, due to my previous experience regarding Layered Architecture and Model View Controller designs, will follow the same basic rule. Views are the dumbest, but the most used. They do one thing only but perfectly, which is to show the user information and provide resources for him to pass his or her desired command.

Model,this time is better structure and is not only a recipient for the data in the file and does more complex stuff such as calculating the fee for withdraws.

I might consider Controllers my bread and butter, therefore this project cannot have weak linking and classes have high cohesion. Data must come from the ser file and be placed in the model classes, from there the Controllers provide them to views and last, but not least views make connection with the user.

# Modeling

In order to keeps a basic, but strong structure for the data fetching operations part. An interface is needed that contains the basic methods, which will be implemented: getAll(), insert(), update(), delete().but no having this exact name.

This exact diagram was required for classes :

# 

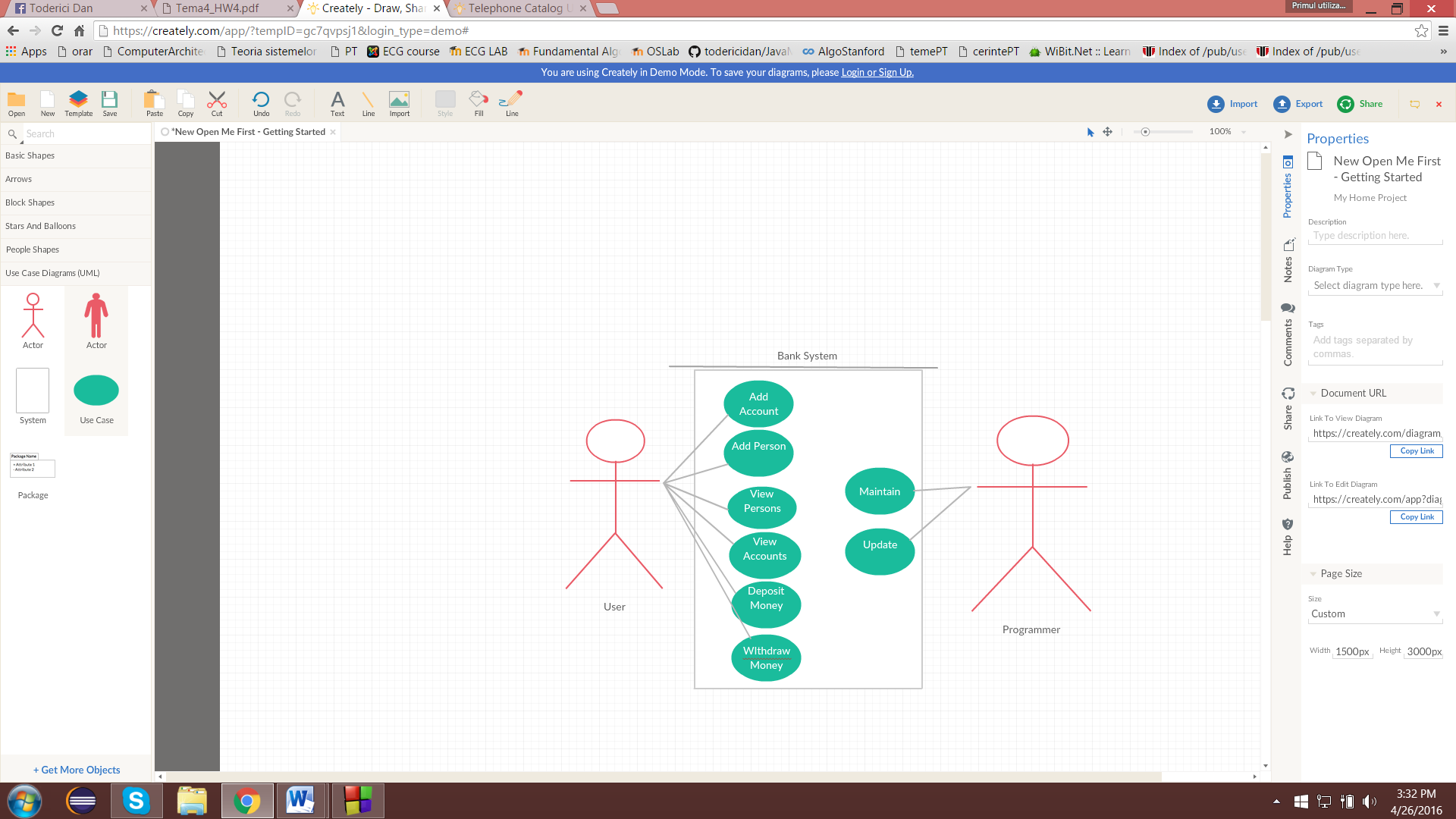
As show there must be a distinct difference between the Saving Account and the Spending Account.

## Use Case Diagram

This type of diagrams have an important role in showing how an user may interact with some features of a certain application. In software and systems engineering, a use case is a list of actions or steps, that define the interaction between a role and a sytem( in the Unified Modeling Language the role is also known as an actor) in order to achieve a goal.

In this case, there are many use case, this chapter will only display and explain some of them (for example depositing money, interrogating the bank, withdrawing money).

As far as the operation are concerned the, the user has to choose which respective button to click on and the result is immediately outputed in a textfield or a new View will be displayed.



The Use Case Diagram of the application

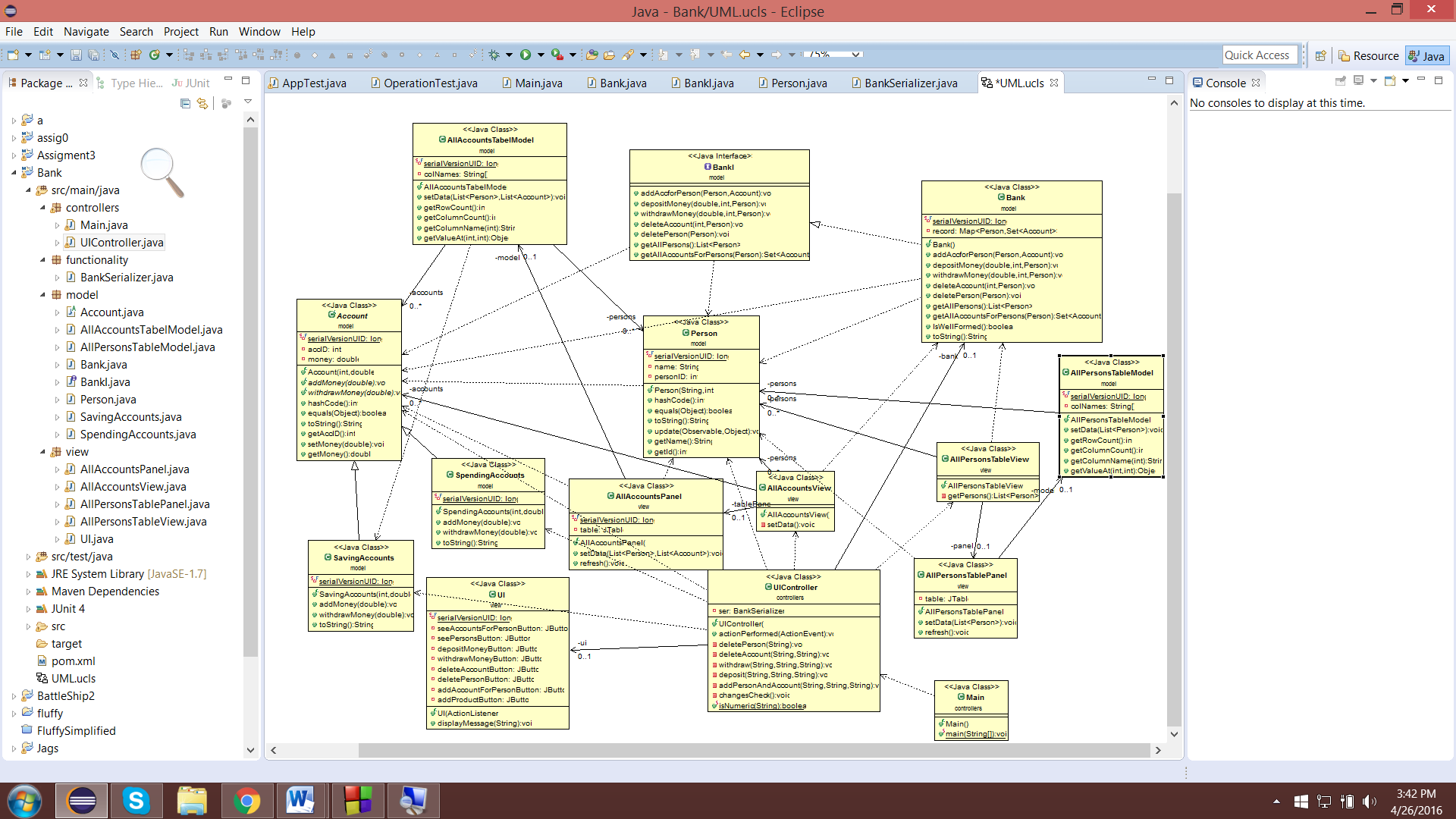
# Design

## Relational Diagram

The entity-relational model is a data model that describes aspects of a business domain or its process requirements, in such an abstract way that it can lead to being implemented in a database such as a relational database. The main components of ER models are entities and the relationships that can exist between them.

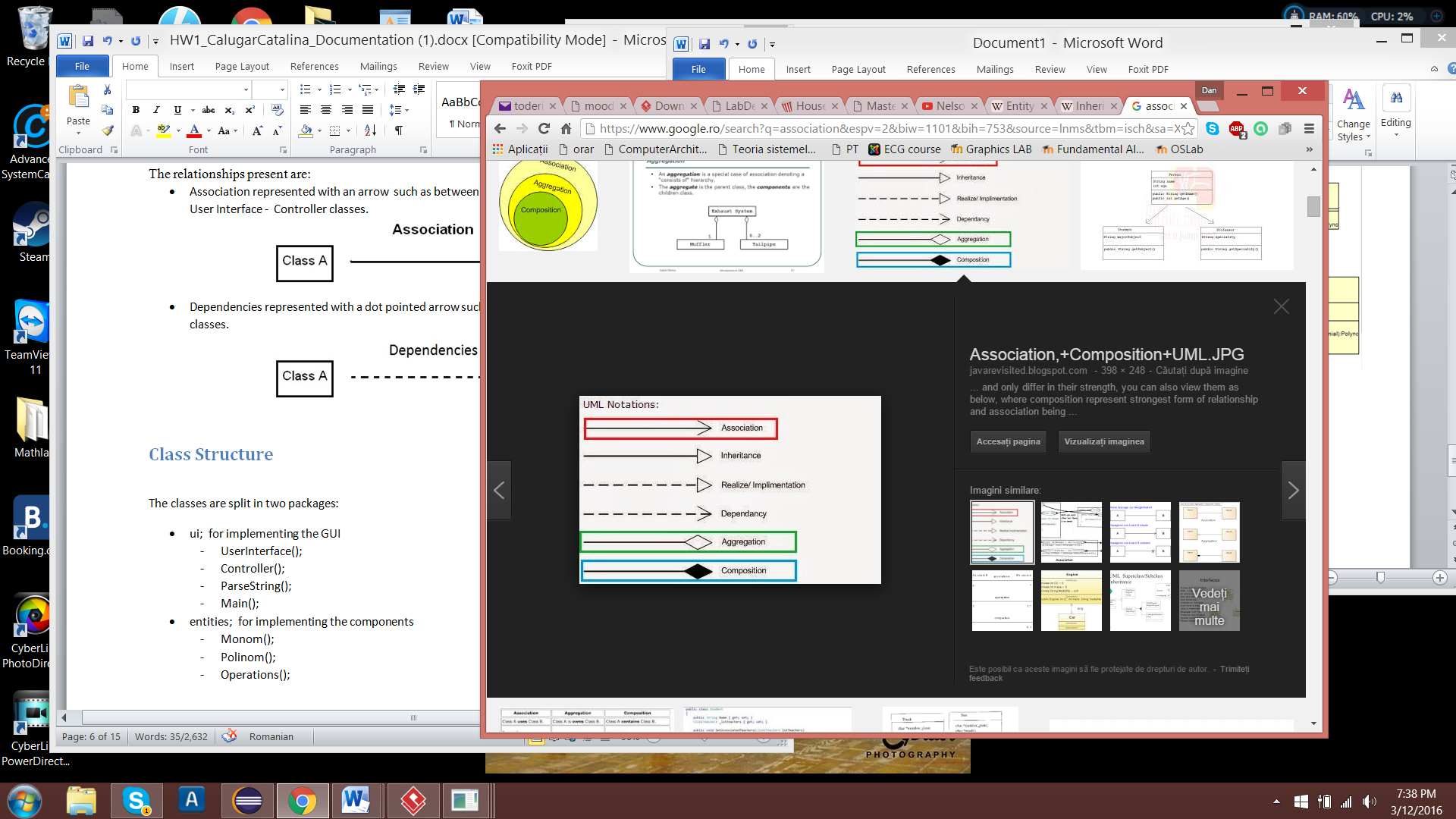
In most cases the ER design is the first step in solving any OOP problem. I have chosen to implement each operation into a separate class and also to add a class for testing. However, testing is needed only for verifying if the persistence classes work properly, because of the dimensions of the project and the number of operations done on the database, we must make sure that those classes are perfect. Also controllers, can have or not a hierarchic structures.

This is the relationship diagram containing associations and dependencies type relations:

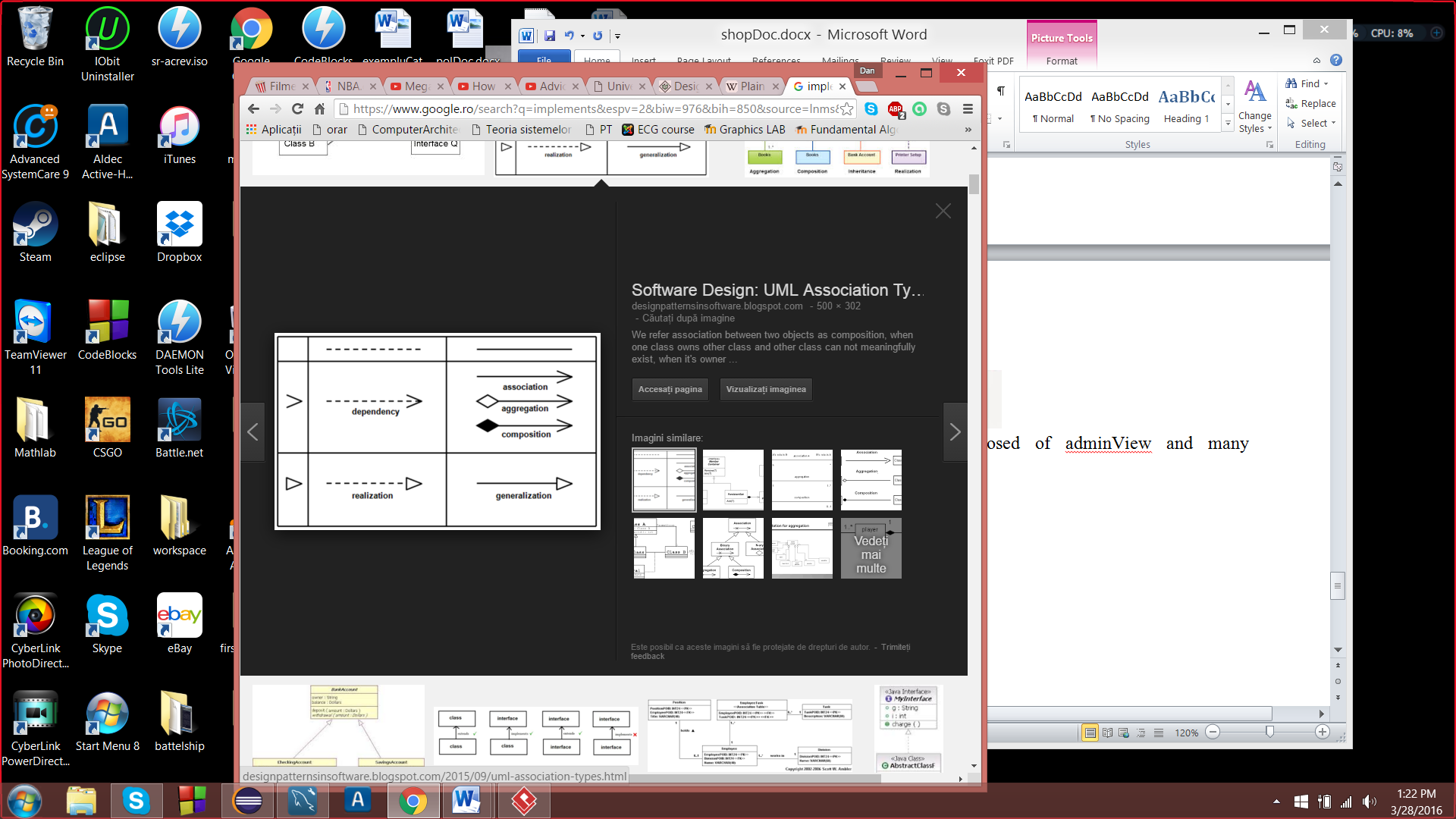


The relationships that are shown in the diagrams are:

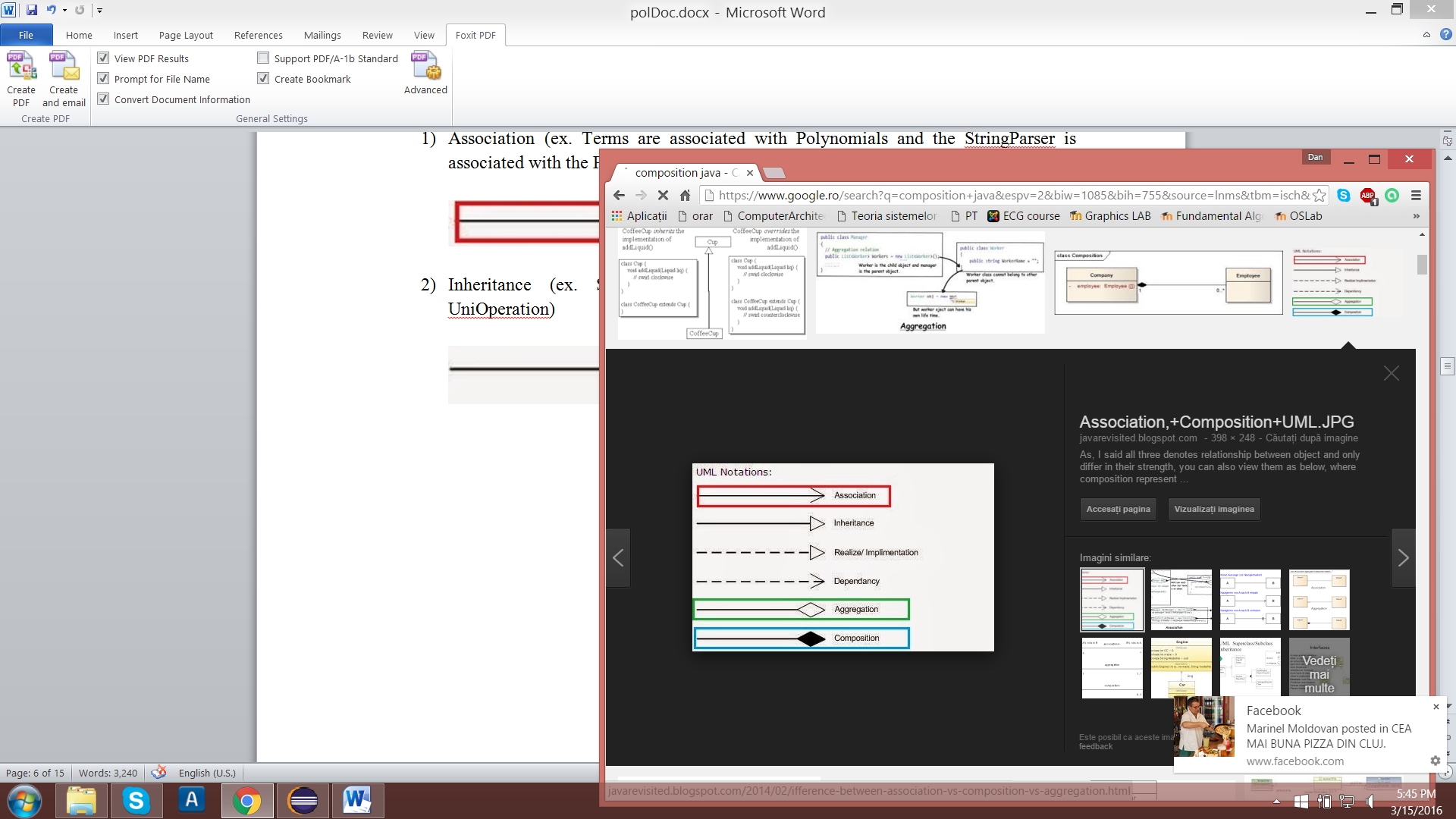
1. Association (ex. Person is associated with the AllPersonsTableModel).



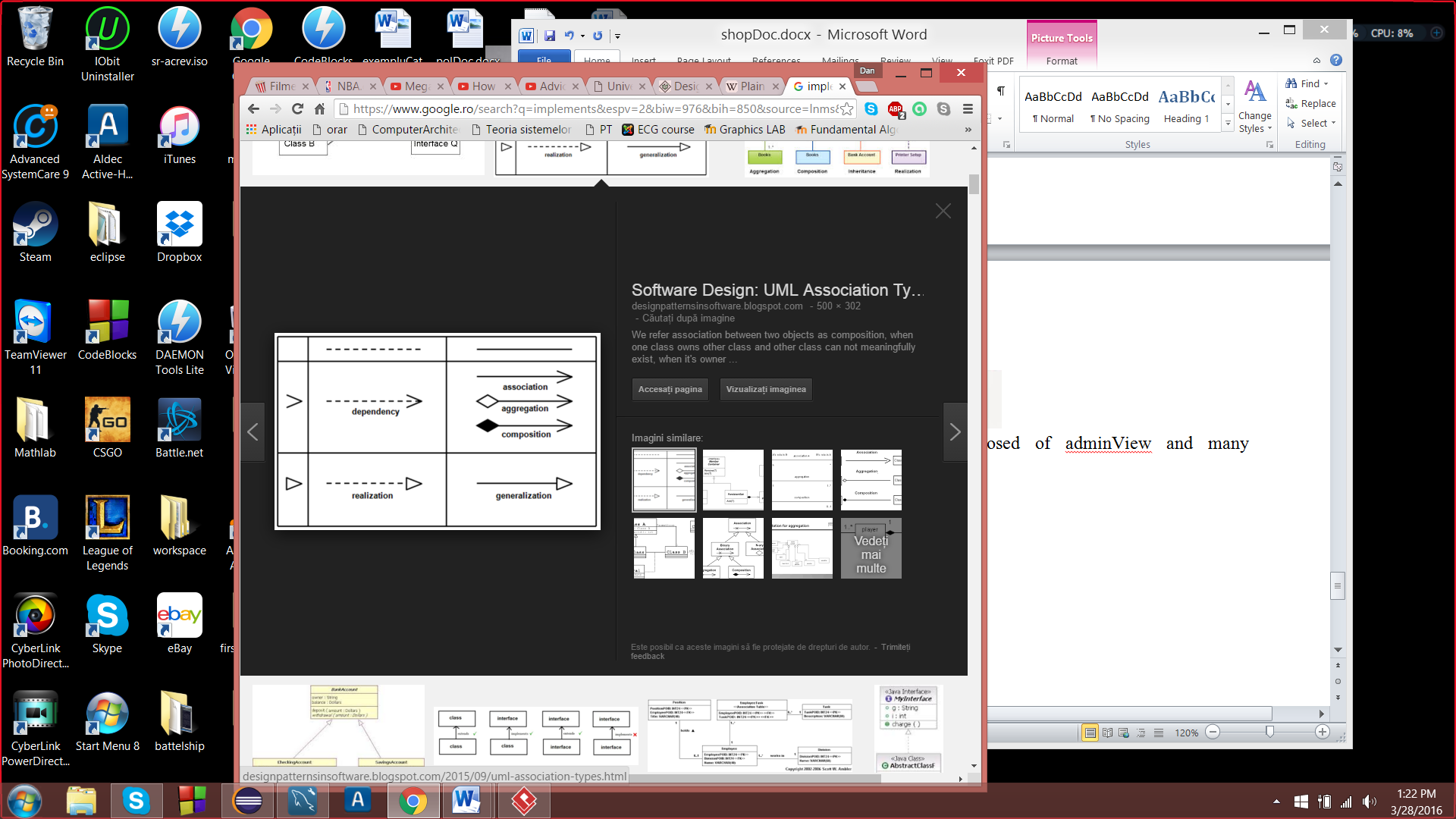
1. Implements (ex. Saving Account and Spending Account implements Account)



1. Composition (ex. UIController is composed of Ui and Bank).



1. Dependency (ex. BankSerialiazer classes depends on Bank)



# Class Structure

The classes are split into numerous packages:

1)Controllers

-Main

-Ui Controller

2)Functionality

-Bank Serializer

3)Model

-Account (Abstract Class )

-All Accounts Table Model

-All Persons Table Model

-Bank

-BankI (Interface)

-Person

-Saving Account

-Spending Account

4)View

-All Accounts Panel

-All Accounts View

-All Persons Table Panel

-All Persons Table View

-Ui

# Implementation

The only two things that are different from other project and on which I will insist are HashMap and Design by Contract

## HashMap

public class **HashMap<K,V>**

extends [AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)<K,V>

implements [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<K,V>, [Cloneable](https://docs.oracle.com/javase/8/docs/api/java/lang/Cloneable.html), [Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html)

Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

This implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of theHashMap instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

An instance of HashMap has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The *load factor*is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is *rehashed* (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the HashMap class, including get and put). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.

If many mappings are to be stored in a HashMap instance, creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table. Note that using many keys with the same hashCode() is a sure way to slow down performance of any hash table. To ameliorate impact, when keys are [Comparable](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html), this class may use comparison order among keys to help break ties.

**Note that this implementation is not synchronized.** If multiple threads access a hash map concurrently, and at least one of the threads modifies the map structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the [Collections.synchronizedMap](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html" \l "synchronizedMap-java.util.Map-) method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

Map m = Collections.synchronizedMap(new HashMap(...));

The iterators returned by all of this class's "collection view methods" are *fail-fast*: if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a [ConcurrentModificationException](https://docs.oracle.com/javase/8/docs/api/java/util/ConcurrentModificationException.html). Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

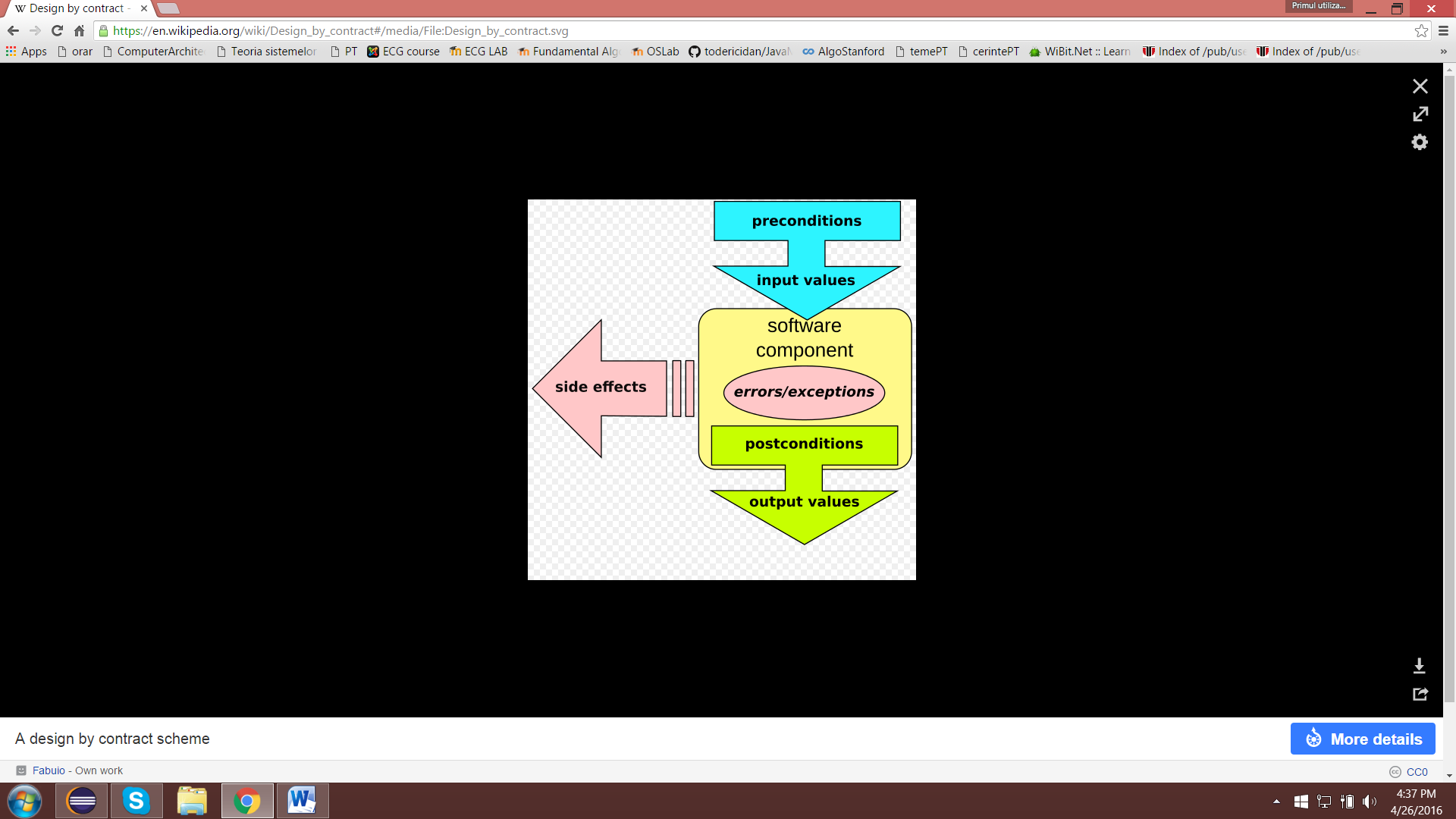
This class is a member of the Java Collections Framework.

In the case of this project, Hash Map has use in the Bank class for storing Persons and Accounts and the hash functions where generate by Eclipse.

## Design by Contract

Design by contract (DbC), also known as contract programming, programming by contract and design-by-contract programming, is an approach for designing software. It prescribes that software designers should define formal, precise and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with preconditions, postconditions and invariants. These specifications are referred to as "contracts", in accordance with a conceptual metaphor with the conditions and obligations of business contracts.

The DbC approach assumes all client components that invoke an operation on a server component will meet the preconditions specified as required for that operation. Where this assumption is considered too risky (as in multi-channel client-server or distributed computing) the opposite "defensive design" approach is taken, meaning that a server component tests (before or while processing a client's request) that all relevant preconditions hold true, and replies with a suitable error message if not.



In this case, this type of implementation assures the composition of the Bank and maintaince the isWellFormed property. This means that there is no person without an account or an account without a person in the Bank object.

## Bank Serializer

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Classes ObjectInputStream and ObjectOutputStream are high-level streams that contain the methods for serializing and deserializing an object.

# Conclusion and further developments

Arriving to the last point of this presentation, I personally considered this project a good exercise for many things, such as layered pattern design implementation, MVC design and DAO classes creation and usage. Though, I firstly tried to implement a setVisibility() method that would help me in switching between views, however when putting it in Controller nothing seemed to work. I left my thoughts behind and I have fallen in the disgrace of opening each view and setting setDefaultCloseOperation(JFrame.DISPOSE\_ON\_CLOSE).

Having more than one problem in hand, the persistence class came as a relief and helped me a lot in implementing my functionalities. In addition, I could have done many more functions based on such architecture, but maybe in the near future or in another project.

Some improvements and future development plans would be:

1. Check the formats of the inputted data if they have the right format, using regex statements or simple methods.
2. Creating a server, for more than one user to use this application at a time and for real time data to be transmitted
3. Have a multi-thread design for fetching and storing data
4. Design a better user interface with much more functionalities and prettier view
5. More notifications for the user, based upon his or her actions.

# References

<http://www.tutorialspoint.com/java/java_serialization.htm>l

<https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html>

<https://en.wikipedia.org/wiki/Design_by_contract>