Data Structures and Algorithms
Faculty of Computer Science
UPV/EHU

Programming Project

Social Network

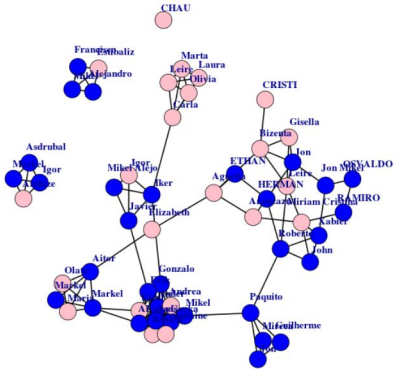
Iyán Álvarez
Davy Wellinger

January 15, 2021

Contents

1. Introduction

This is a short introduction to the **Programming Project** made by Iyán Álvarez and Davy Wellinger.



The goal of this project is the creation a functional **Social Network** that coordinates people and the relations between them. Implementing not only different basic functions, such as **loading** people and relations into the network or **downloading** it's contents into a ".txt" file, but also utilitarian functions, like simple **searching and sorting.**

In order for the user to interact with the network, we have implemented a simple **console application** that gives the user various interaction options. At the end of the project, we might implement a basic **GUI** to replace this system.

From the programming point of view, we're interested in implementing as many types of **data structures** as possible, in order to visualise their different strengths and weaknesses and improve the running cost of the program.

To further understand the data structures implemented in the final version of the project, we decided to first implement everything using **java.util's ArrayList.** This way, upon trying out the different possible data structure, we will be able to see where they excels over each other.

In the end, this Programming Project will be implemented in the most suitable way for the usage of the diverse utilitarian functions: **Searching and Sorting**. And focus on making those processes run smooth and fast.

2. First version of the Project

2.1 Classes design and method implementations

Our project is based on **four classes:**

- **Person:** Person has a lot of data/information and can have relations with other persons on the Social Network.

  Person class has **11 variables**, 3 of them are arrays, to store all the specific information of each person:

  - **identifier:** User ID's for the Social Network.
  - **name:** Person's name.
  - **surname:** Person's surname.
  - **birthdate:** Person's birthdate.
  - **gender:** Person's gender.
  - **birthplace:** Person's birthplace.
  - **home:** Person's home.
  - **studydata:** Person's study data.
  - **workdata:** Person's work data.
  - **movies:** Person's favourite movies.
  - **groupcode:** Groupcode.

  Person class implements some **methods** too:

  - **Getters:** Gets the value stored in a variable.
  - **equals():** Compares a Person with the given Object and returns true if the ID, name and surname are the same.
  - **toString():** Returns a String with the information of the person in the format: idperson,name,lastname,birthdate,gender,birthplace,home,studiedat,workplaces, films,groupcode.

- **Relation:** Relation defines the relation of two people, this relation is mutual.

  Relation class has **2 variables**, to store a relation between two people:

  - **person1:** Person related to person2.
  - **person2:** Person related to person1.

  Relation class implements some **methods** too:

  - **Getters:** Gets the value stored in a variable.
  - **equals():** Compares a relation with the given Object and returns true if the relations have the same meaning.
  - **toString():** Returns a String with the information of the relation in the format: person1,person2.

- **SocialNetwork:** SocialNetwork contains people information and the relations between them if exist.
  SocialNetwork class has **4 variables**, to save person and relations data, a scanner to read user's input and to check that only one instance exists:

  - **personList:** ArrayList that stores the Person instances of the SocialNetwork.
  - **relationList:** ArrayList that stores the Relation instances of the SocialNetwork.
  - **instance:** Controls the existance of an instance of this variable; based on the Singleton concept.
  - **sc:** Scanner to read user's input from console.

  SocialNetwork class implements several **methods** too:

  - **getInstance():** Gets instance of the Social Network, if it does not exist it creates a new one, that is going to be unique and returns it.
  - **initialMenu():** Presents an initial menu with the different choices for interacting with the social network.
  - **printInitialMenu():** Prints the choices of the initial menu.
  - **printAddPersonPeople():** Prints the choices of Add person or people.
  - **printAddRelations():** Prints the choices of Add relations.
  - **printPrintOut():** Prints the choices of Print out people.
  - **printFind():** Prints the choices of Find.

- **selectionInitialMenu():** Request user to select an option and do what was specified in the menu.
- **askForSelectionInput():** Request user for an input, it has to be a number and treats InputMismatchException.
- **addPersonPeopleSelected():** Prints Add person option and performs task.
- **addRelationsSelected():** Prints Add relation option and performs task.
- **printOutSelected():** Prints Print out option and performs task.
- **printFindSelected():** Prints Find option and performs task.
- **addPerson(data):** Adds a person to the SocialNetwork.
- **addPeopleFromFile(filename):** Adds all the people from the file to the Social Network.
- **printPeopleToConsole():** Prints all the people at the SocialNetwork to the console.
- **printPeopleToFile(filename):** Prints all the people at the SocialNetwork to a file.
- **addRelation():** Adds a relation of 2 people.
- **addRelationsFromFile():** Adds all the relations that the specified file contains.
- **existsInSocialNetwork():** Checks if one person exists in the social network given the ID (identifier).

- **SocialNetworkSimulation:** SocialNetworkSimulation simulates how the SocialNetwork works.

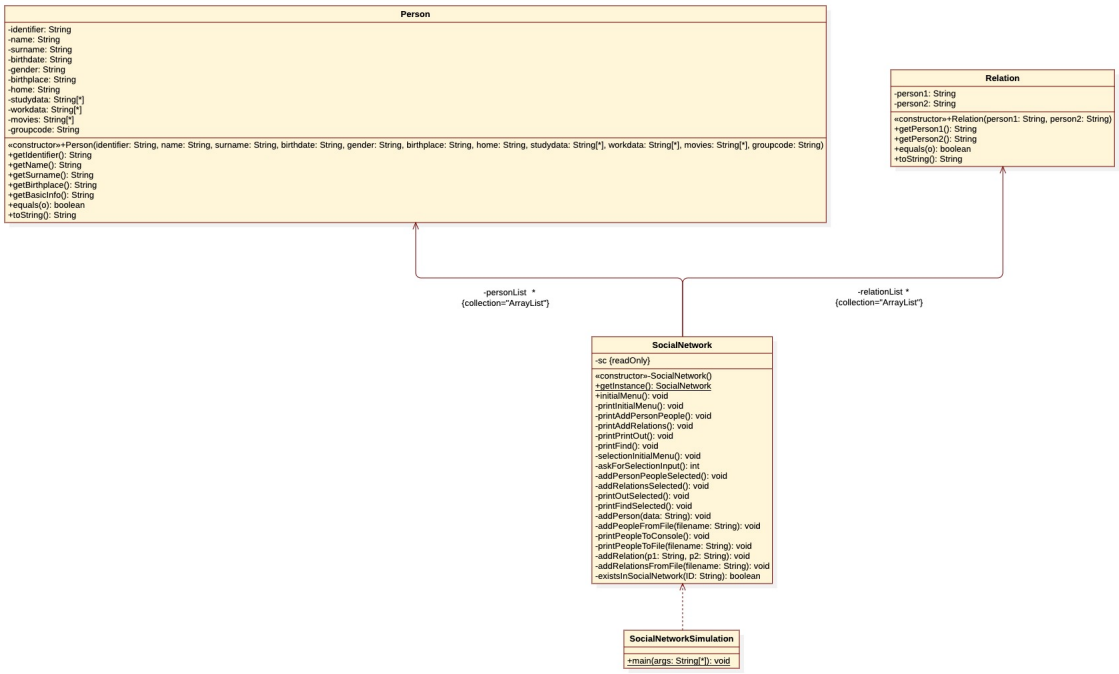
  SocialNetworkSimulation class has **no variables** and has **a main method**.

  - **main(args):** Launches SocialNetwork's initial menu and user can interact with it.

In addition, we have implemented **three exceptions.** All the exceptions extend from Java's RuntimeException, because this errors can be given on the runtime of the program, sometimes depending on the data introduced.

- **PersonAlreadyAtSocialNetwork:** If the person's that wants to be added already exists in the SocialNetwork.
- **PersonNotFoundException:** If at least one person of a relation does not exist in the SocialNetwork.
- **RelationAlreadyAtSocialNetwork:** If the relation's that wants to be added already exists in the SocialNetwork.

2.2 UML design of Social Network



2.3 Description of the data structures used in the project

As mentioned in the introductory description, up to this point, the **Milestone 1**, we have focused on creating a basic framework upon which we will build and expand all functions.

ArrayList
To make things simpler, we decided implementing first only **java.util's ArrayList** and later trying out different data structures, to easily visualize the advantages and disadvantages for our project.

The **ArrayList** class included in java's library **java.util** is easy to use and to keep track of. Being a auto-expanding data structure, we must not worry about running out of space. Similarly, the structure is easy to iterate over and its elements can be inserted anywhere and be identified and deleted easily.

3. Second version of the Project

3.1 Classes design and method implementations

Our project is based on **four classes:**

- **Person:** Person has a lot of data/information and can have relations with other persons on the Social Network.

  Person class has **11 variables**, 3 of them are arrays, to store all the specific information of each person:

  - **identifier:** User ID's for the Social Network.
  - **name:** Person's name.
  - **surname:** Person's surname.
  - **birthdate:** Person's birthdate.
  - **gender:** Person's gender.
  - **birthplace:** Person's birthplace.
  - **home:** Person's home.
  - **studydata:** Person's study data.
  - **workdata:** Person's work data.
  - **movies:** Person's favourite movies.
  - **groupcode:** Groupcode.

  Person class implements the following **methods**:

  - **Getters:** Gets the value stored in a variable.
  - **Setters:** Sets the value given in the corresponding variable.
  - **Comparators:** Compares different values of the corresponding variables.
  - **compareTo(o):** Compares a Person with the given Person by the ID.
  - **equalsStudydata(ob2):** Method for using in other methods. Compares a person by its Studydata.
  - **equalsWorkdata(ob2):** Method for using in other methods. Compares a person by its Workdata.

- **equalsMovies(ob2):** Method for using in other methods. Compares a person by its favourite movies.
- **equals():** Compares a Person with the given Object and returns true if the ID, name and surname are the same.
- **toString():** Returns a String with the information of the person in the format: idperson,name,lastname,birthdate,gender,birthplace,home,studiedat,workplaces, films,groupcode.

- **Relation:** Relation defines the relation of two people, this relation is mutual.

  Relation class has **2 variables**, to store a relation between two people:

  - **person1:** Person related to person2.
  - **person2:** Person related to person1.

  Relation class implements some **methods** too:

  - **Getters:** Gets the value stored in a variable.
  - **compareTo(o):** Compares a Relation with the given Relation by the person1, and if needed by person2.
  - **equals():** Compares a relation with the given Object and returns true if the relations have the same meaning.
  - **toString():** Returns a String with the information of the relation in the format: person1,person2.

- **SocialNetwork:** SocialNetwork contains people information and the relations between them if exist.
  SocialNetwork class has **4 variables**, to save person and relations data, a scanner to read user's input and to check that only one instance exists:
  - **personList:** ArrayList that stores the Person instances of the SocialNetwork.
  - **relationList:** ArrayList that stores the Relation instances of the SocialNetwork.
  - **instance:** Controls the existance of an instance of this variable; based on the Singleton concept.
  - **sc:** Scanner to read user's input from console.

  SocialNetwork class implements several **methods** too:
  - **getInstance():** Gets instance of the Social Network, if it does not exist it creates a new one, that is going to be unique and returns it.
  - **initialMenu():** Presents an initial menu with the different choices for interacting with the social network.
  - **printInitialMenu():** Prints the choices of the initial menu.
  - **printAddPersonPeople():** Prints the choices of Add person or people.
  - **printAddRelations():** Prints the choices of Add relations.
  - **printPrintOut():** Prints the choices of Print out people.
  - **printFind():** Prints the choices of Find.
  - **printSearch():** Prints the choices of Search.
  - **selectionInitialMenu():** Request user to select an option and do what was specified in the menu.
  - **askForSelectionInput():** Request user for an input, it has to be a number and treats InputMismatchException.
  - **addPersonPeopleSelected():** Prints Add person option and performs task.
  - **addRelationsSelected():** Prints Add relation option and performs task.
  - **printOutSelected():** Prints Print out option and performs task.
  - **printFindSelected():** Prints Find option and performs task.
  - **searchPersonPeopleSelected():** Method that interacts with the user to find people by attribute.
  - **addPerson(data):** Adds a person to the SocialNetwork.
  - **addPeopleFromFile(filename):** Adds all the people from the file to the Social Network.

- **printPeopleToConsole():** Prints all the people at the SocialNetwork to the console.
- **printPeopleToFile(filename):** Prints all the people at the SocialNetwork to a file.
- **addRelation(p1, p2):** Adds a relation of 2 people.
- **addRelationsFromFile(filename):** Adds all the relations that the specified file contains.
- **existsInSocialNetwork(ID):** Checks if one person exists in the social network given the ID (identifier).
- **binarySearchPersonID(ID):** Makes a binary search in the ArrayList ordered by the ID. Frontend method.
- **binarySearchPersonIDBack(ID):** Makes a binary search in the ArrayList ordered by the ID. Backend method.
- **findPersonBySurname(surname):** Given a surname, finds the Person(s) in the SocialNetwork with that surname and returns them in an ArrayList of Person.
- **findFriendsBySurname(surname):** Given a surname, returns a String with the friends of the user(s) with that surname.
- **printFriendsBySurnameToConsole(surname):** Given a surname, prints the friends of the user(s) with that surname in the console.
- **printFriendsBySurnameToFile(surname, filename):** Given a surname, prints the friends of the user(s) with that surname in the specified file.
- **findPersonByCity(city):** Given a city, finds the Person(s) in the SocialNetwork that have born in that city and returns them in an ArrayList of Person.
- **findPersonByCityStringBasic(city):** Given a city, returns a String with the basic info of the user(s) born in that city.
- **printPersonByCityToConsole(city):** Given a city, prints the user(s) basic info that has born in the given city in the console.
- **printPersonByCityToFile(city, filename):** Given a city, prints the user(s) basic info that has born in the given city in the specified file.
- **findPersonBetweenDates(year1, year2):** Given two dates, finds the Person(s) in the SocialNetwork born between the given years an return them in a sorted ArrayList of Person. Pre: Year 1 <= Year2.
- **findPersonBetweenDatesString(year1, year2):** Given two dates, finds the Person(s) in the SocialNetwork born between the given years an return them in a sorted ArrayList of Person. Pre: Year 1 <= Year2.

- **printPersonBetweenDatesToConsole(year1, year2):** Given two dates, prints the user(s) basic info that has born between the given years.
- **printPersonBetweenDatesToFile(year1, year2, filename):** Given two dates, prints the user(s) basic info that has born between the given years in the specified file.
- **findPersonByCityStringMore(city):** Given a city, returns a String with more info of the user(s) born in that city.
- **findPersonByCityResidentialString():** Finds the Person(s) in the SocialNetwork that have born in the same city as the ID's given in the file residential.txt in an String. Pre: The file residential.txt must be on folder files/.
- **printPersonByCityResidentialToConsole():** Prints the user(s) more info of the ones that have born in the given ID's birthplace in the console.
- **printPersonByCityResidentialToFile(filename):** Prints the user(s) more info of the ones that have born in the given ID's birthplace in the specified file.
- **splitPersonByMovies():** Splits Person(s) by individual favourite movies in a HashMap, where the keys are the favourite movies, and the values an ArrayList of Person that have that favourite movie in common.
- **getPersonListMovies(movies):** Obtains the list of Person(s) that share that favourite movie in common, if exist.
- **getPersonListMoviesString(movies):** Obtains a String of Person(s) that share that favourite movie in common, if exist.
- **printPersonListMoviesToConsole(movies):** Prints the user(s) that share the specified favourite movies basic info in the console.
- **printPersonListMoviesToFile(movies, filename):** Prints the user(s) that share the specified favourite movies basic info in the specified file.
- **splitPersonByMoviesString():** Splits Person(s) by individual favourite movies in a HashMap, where the keys are the favourite movies, and the values an ArrayList of Person that have that favourite movie in common.
- **printPersonByMoviesToConsole():** Prints all the collections of favourite and the user(s) basic info that share each collection to console.
- **printPersonByMoviesToFile(filename):** Prints all the collections of favourite and the user(s) basic info that share each collection in the specified file.
- **sortPersonList(attribute):** Method that sorts the list containing all users by a chosen parameter.
- **findAdjacentSearch(index, searching, attribute):** Method that creates a list of all indexes in the peopleList that match the search.

- **searchPersonList(searching, attribute):** Method that searches the list containg all users for a certain Person that has the desired attribute(s). Only one attribute at a time, but in case of a array type Attribute, the user must write a single string, containing all desired strings speparated by ";". Prints the found person(s).
- **printSearchedPersonList(searching, attribute):** Simply prints all results of a search. This is the method used for searching for the user.
- **getIndex(p):** Returns the index of a person in the list.
- **equalsMovieCollection(ob1, ob2):** Method for using in other methods. Compares a person by its favourite movies. All movies have to be the same / it has to be the same collection.
- **containsMovieCollection(c, m):** Method checks if a list of movie collections contains a certain collection.
- **getMovieCollections():** Method that gets all different movie collections from the network.

- **SocialNetworkSimulation:** SocialNetworkSimulation simulates how the SocialNetwork works.

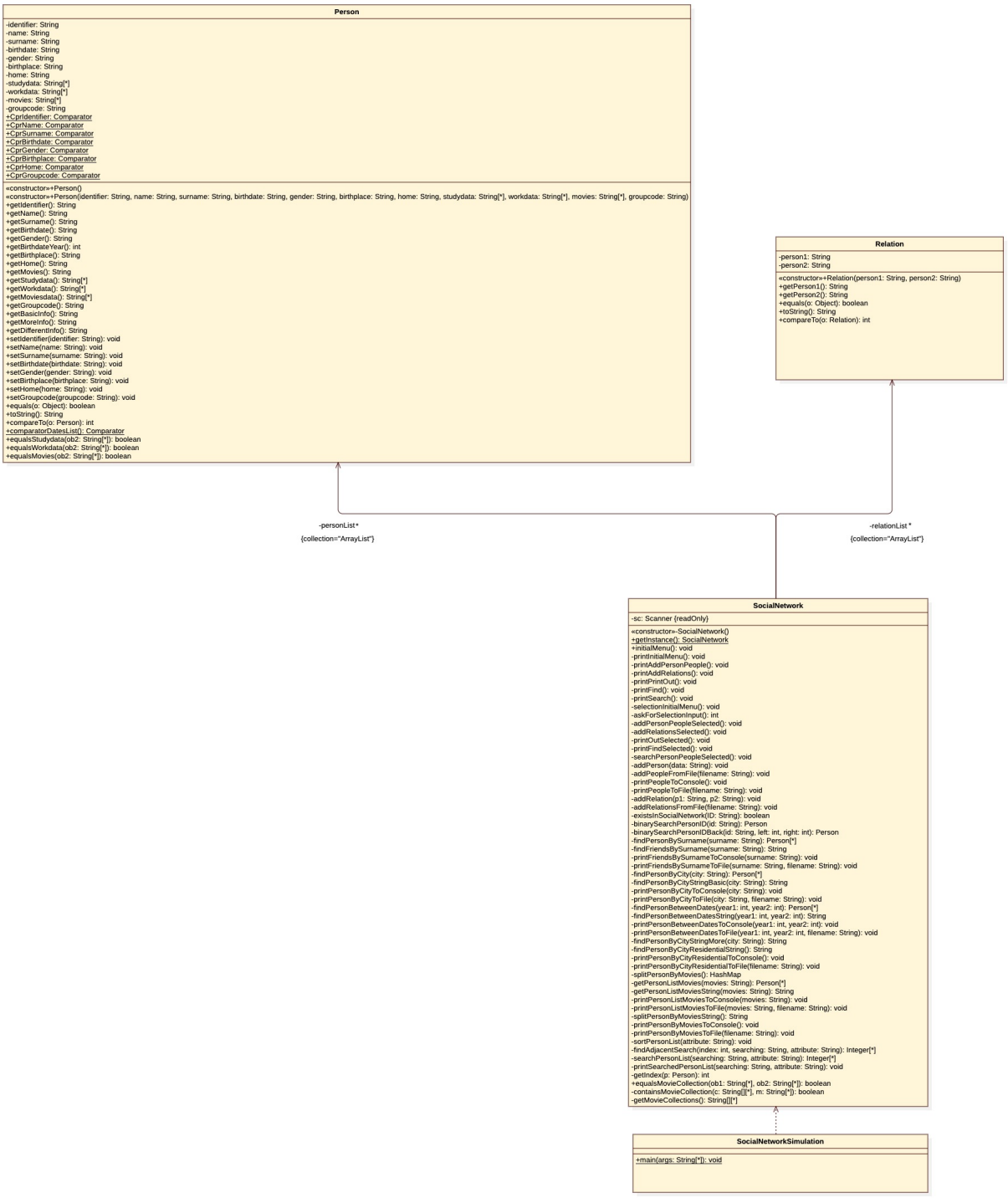
  SocialNetworkSimulation class has **no variables** and has **a main method**.

  - **main(args):** Launches SocialNetwork's initial menu and user can interact with it.

In addition, we have implemented **three exceptions.** All the exceptions extend from Java's RuntimeException, because this errors can be given on the runtime of the program, sometimes depending on the data introduced.

- **PersonAlreadyAtSocialNetwork:** If the person's that wants to be added already exists in the SocialNetwork.
- **PersonNotFoundException:** If at least one person of a relation does not exist in the SocialNetwork.
- **RelationAlreadyAtSocialNetwork:** If the relation's that wants to be added already exists in the SocialNetwork.

3.2 UML design of Social Network



3.3 Description of the data structures used in the project

ArrayList

As mentioned in the introductory description, up to this point, the **Milestone 1**, we have focused on creating a basic framework upon which we will build and expand all functions.

To make things simpler, we decided implementing first only **java.util's ArrayList** and later trying out different data structures, to easily visualize the advantages and disadvantages for our project.

The **ArrayList** class included in java's library **java.util** is easy to use and to keep track of. Being a auto-expanding data structure, we must not worry about running out of space. Similarly, the structure is easy to iterate over and its elements can be inserted anywhere and be identified and deleted easily.

ArrayList (sorted)

During the process of coding the features included in the second milestone of our project, **Milestone 2**, we came across a decision point: What data structure is best for our purposes? Or in other words, which data structure gives us an **efficient use of space** while maintaining the **cost of utilitarian functions low**?

In order to answer those questions we started digging deeper into **Sorting and Searching Algorithms**, studying the applications of **Quick Sort** and **Binary Search** with special care, as it became clear quickly that those were the algorithms we wanted to use at this stage of the project.

First, we coded and implemented two special methods, one with the purpose of **sorting our ArrayList of people by any attribute** and the other performing **binary search by attribute on that sorted list**. A part from that, we coded a method that would find the adjacent people that fit the search criteria and other auxilary methods, such as printing and processing. With these methods, the user can now find any person(s), that fit a given search criteria in a certain attribute category. Although this wasn't specifically asked for, we thought it would be something that a user should be able to do while working with a **Social Network**. These methods also came in handy while implementing other features of the second Milestone.

After implementing the mentioned **Searching and Sorting Algorithms**, it was apparent that **sorting** our list of people every time we needed to search for something was not ideal, as it would suppose a cost of **O(n log n)** + **O(log n)**, as opposed to **O(n)**, when we maintain the list sorted in a single way and linearly iterate over it. Finally, we decided to implement the **search by attribute methods** as an **option for the user** and for further use, but maintain our list of people **always sorted by the attribute "identifier"**.

Array

Generally we tried to use simple **arrays** wherever we could, especially as **return types** of methods for handy post-usage.

LinkedList

A part from **ArrayLists** as our main data structures for storing people and their relations, we got used to using **LinkedLists** for bigger amounts of data, obtained e.g. between auxiliary method calls or storing data obtained from searches for further processing.

HashMap

For the last feature included in the **second Milestone**, we immediately thought of using a **HashMap** for storing the different "movie-classes" along with the people belonging to them, using the **"movie-classes" as keys.**

4. Final version of the Project

4.1 Classes design and method implementations

Our project is based on **three classes:**

- **Person:** Person has a lot of data/information and can have relations with other persons on the Social Network.

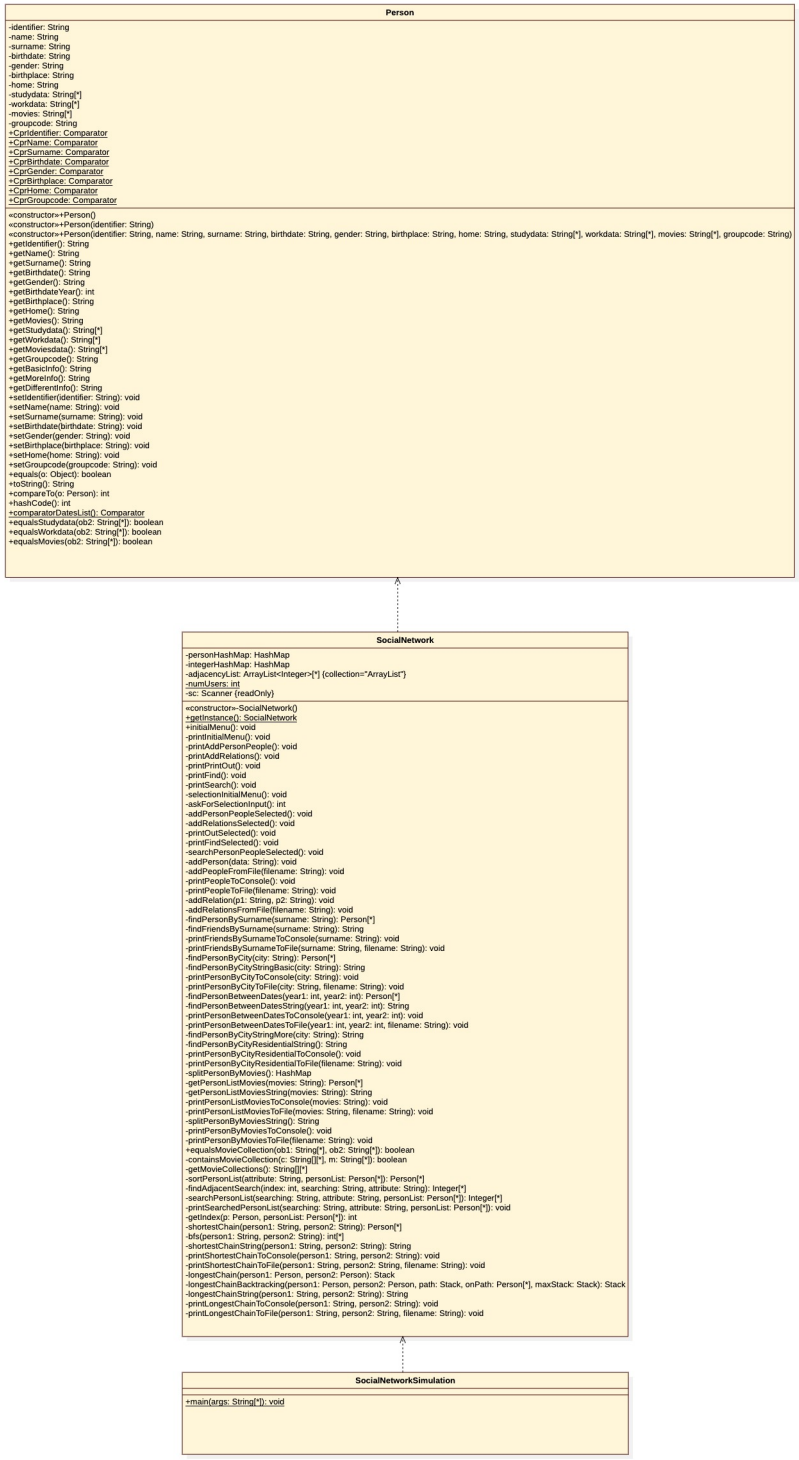
- **SocialNetwork:** SocialNetwork contains people information and the relations between them if exist.
- **SocialNetworkSimulation:** SocialNetworkSimulation simulates how the SocialNetwork works.

In addition, we have implemented **three exceptions.** All the exceptions extend from Java's RuntimeException, because this errors can be given on the runtime of the program, sometimes depending on the data introduced.

- **PersonAlreadyAtSocialNetwork:** If the person's that wants to be added already exists in the SocialNetwork.
- **PersonNotFoundException:** If at least one person of a relation does not exist in the SocialNetwork.
- **RelationAlreadyAtSocialNetwork:** If the relation's that wants to be added already exists in the SocialNetwork.
- **RelationDoesNotExist:** If the relation does not exist in the SocialNetwork.

All the documentation of the last version of the project can be seen in the JavaDoc.

4.2 UML design of Social Network



4.3 Description of the data structures used in the project

ArrayList

As mentioned in the introductory description, up to this point, the **Milestone 1**, we have focused on creating a basic framework upon which we will build and expand all functions.

To make things simpler, we decided implementing first only **java.util's ArrayList** and later trying out different data structures, to easily visualize the advantages and disadvantages for our project.

The **ArrayList** class included in java's library **java.util** is easy to use and to keep track of. Being a auto-expanding data structure, we must not worry about running out of space. Similarly, the structure is easy to iterate over and its elements can be inserted anywhere and be identified and deleted easily.

ArrayList (sorted)

Some previous method implementations keep using the sorted ArrayList.

Array

Generally we tried to use simple **arrays** wherever we could, especially as **return types** of methods for handy post-usage.

LinkedList

A part from **ArrayLists** as our main data structures for storing people and their relations, we got used to using **LinkedLists** for bigger amounts of data, obtained e.g. between auxiliary method calls or storing data obtained from searches for further processing.

HashMap

For the last feature included in the **second Milestone**, we immediately thought of using a **HashMap** for storing the different "movie-classes" along with the people belonging to them, using the **"movie-classes" as keys.**

Besides that, for the Graph implementation used, we have map every Person to a number and every number to a Person, this way we can improve many operations in **constant time**, for example, to know if a Person exists in the Social Network or to check the friends of a given Person.

Graph

For the last points we have changed the main data structure of the Social Network to a Graph based on adjacency list. This adjacency lists contains all the relations between the users in the Social Network. This structure is indexed, this way we can access any element in constant time. Each number corresponds to a Person that can be obtained mapping the number with the **HashMap**.

5. Notes of the Meetings

5.1 1st milestone

For the 1st milestone of the project we held 4 meetings:

- **First meeting**
  In our first meeting, we organised the points that we had to work at on different issues at GitHub, and we set up all the environment and the repository on GitHub to work simultaneously on the programming project. We discussed the possible implementation structures, how to organise people, relations and how to implement the menu and some other methods. Apart from that, we decided to develop a individual approach of the Social Network and the classes design and to implement it with our personally preferred approach, to later combine our ideas if needed.

- **Second meeting**
  In the second meeting we compared both of our implementations. We analyzed each other's approaches in some of the topics of the Social Network, and discussed which ideas were the best, combining everything into one project and decided to implement one of them to the master branch of the programming project. We decided to make a fully robust Social Network, treating any exceptions that might be thrown at run-time. This meeting was very productive since we learnt a lot and saw each other's approach.

- **Third meeting**
  In our third meeting, after finishing the implementation of the first milestone of the programming project, we started to work on the documentation and made some experiments with the simple GUI mentioned above. We finally decided to remodel the menu, and let the GUI be an option for a later implementation of the project.

- **Fourth meeting**
  In the fourth meeting we made a final review of the project, and finished the documentation of the 1st milestone of the programming project. We have discussed different options for next implementations and uploaded all the material to eGela.

5.2 2nd milestone

For the 2nd milestone of the project we also held 4 meetings:

- **First meeting**
  In our first meeting regarding the second milestone, we briefly reviewed our implementation of milestone 1 and started discussing what would be our best course of action from here on. Of course we wanted to change and implement new data structures, since it was obvious we needed to make changes in order to improve our spacial and running costs.

  We stayed true to our proposal from the very beginning of the project: Starting with an arraylist as a base and make improvements from there. While searching and evaluating our options, the most interesting was definitely the idea of working with sorted data structures. As for the non-linear approach, we chose the tree structure, specifically the binary tree with which we could perform binary searches. The linear approach led us to a sorted arraylist, which would also allow efficient search algorithms. At this point of the process, we didn't yet address the ideal data structure for handling the relations.

  After running some calculations, we came to the conclusion that a binary tree would be more efficient at inserting and deleting, as all operations are of cost O(log n), but ultimately we decided to implement the sorted arraylist version, since the main focus of this project lies on searching and sorting and we would know how to handle it better for more efficient and optimized programming. As with the first Milestone, we both implemented our own code in different branches for later comparison and *cherry-picking*.

- **Second meeting**
  We held our second meeting relatively soon in proportion to the time between meetings in the first milestone. While Iyán had made improvements to the basic code and implemented the sorted arraylist motioned before, Davy had implemented two master methods for sorting and searching by attribute. After a brief discussion about alternatives to the sort and search implementation, we quickly discovered that keeping our arraylist sorted in one unique way, by the attribute "identifier" of the Person class, would be the more efficient approach for handling the features asked for in the second milestone. Thus we implemented the sort and search methods and their auxiliary methods as a sort of bonus to the project and went with the singly sorted arraylist.

- **Third meeting**
  In our third session we came together in order to find and kill bugs in the code. Mainly we wanted to take care of all possible exceptions that a user could cause and checked the functionality of all methods. Our final review was dedicated to the new features of milestone 2 where it became apparent that a more appropriate data structure for storing and handling the relations was necessary. Although we hadn't seen it in class yet, we discovered that a graph, implemented with an adjacency matrix, could be an efficient approach as it is usually used to link vertices with non-weighted edges in network-like problems. With this data structure it would be very easy and efficient to find and distinguish relations between users and e.g. to find cliques. However, due to the lack of time we decided to implement this further on, in milestone 3.

- **Fourth meeting**
  In our last meeting for this milestone, we committed and pushed our code to GitHub and merged everything together, obtaining the current build. We finished up the documentation and submitted everything to eGela.

5.3 3rd milestone

For the 3rd milestone of the project we held 3 meetings:

- **First meeting**
  In our first meeting, we talked about the complexity of the new tasks and thought about how to implement them. We first thought of creating a graph at runtime to perform the tasks that needed BFS or DFS. But we thought that creating a graph always that we wanted to find a chain of relation was not the best option.

  We had different points of view and each worked on different versions to, later on, compare and decide the best implementation. The two different approaches were on the one side, using the graph data structure only for the relations and on the other side, changing the whole implementation of the data structures used for storing the people and relations to *HashMap* and *Graph*. We tried out both version of the project in order to later decided between one of them.

- **Second meeting**
  For the second meeting we discussed the different implementations. Davy had implemented a Graph class with Vertexes that were connected to each other, giving a nice and clean code used for storing only the relations of the people. Iyán changed both, the data structure used to implement the list of people and the one of relations. Namely, changing the up to know used ArrayList's to a Graph-based in an adjacency list, and accessing to two HashMaps to had a better efficiency looking to the runtime. After some tests to check that both versions were working correctly and a brief discussion about both approaches, we decided to implement Iyán's version, because in general some of the methods were reduced in time complexity, and we didn't need to use a different class to organize all the data. Moreover, we were using more interesting data structures and how to use them correctly such as the Graph's adjacency list and the HashMap's.

  We decided to work on the next points, in different versions as we have done previously and meet again to discuss the last points of the project.

- **Third meeting**
  In our third session, we came together to discuss the new code, at first glance none of us achieved to fulfil the goal of having both methods done. We had some difficulties with the backtracking concepts. Although the concept was clear, at the time of programming we did not know how to implement it correctly. Moreover, we thought that we were not going to achieve any of the last two points (longest path and cliques). Finally, Davy's last-minute implementation worked and we included the *longestPath()* method in the final build. In order to do that, we made some changes to fit the project version of Iyán because Davy had been working on his own version, for higher proficiency. Finally, we

could not implement the last point of the project because of lack of time, although we did manage to write good code, that sadly didn't return the right results.

6. Conclusions

Conclusion

Concluding, it is no understatement to say that we learned a lot about data structures and algorithms during this course and the making of this project. Although there where some difficulties, ultimately, we were able to implement all the required methods and utilities, except for the last one. During the process of coding, we also got to deepen our knowledge about the **java language** and improved our skills and workflow using the **git and GitHub**.

Personally we have to say that we enjoyed working on this project a lot. We think that a project of this size is definitely a great addition to our repositories and curricula, for companies to check-out in the future.

Main Merits

Efficient use of Data Structures. Robust project structure (high use of exceptions). Clean and easy-to-read code. Singleton implementation.

Work Distribution

In our first meeting we quickly determined the way we wanted to work together. We both thought that in order for each of us to draw the most experience out of the project, we should both implement everything - each with our own approaches. This means that the final version of the project is a combination of many of our ideas and implementations. Due to this, we would say its a clear 50/50 distribution.

Afterthought

We hope that anyone who sees this project, gets to learn from it and that some of our implementations may be of help to you for your own projects.

7. Bibliography

- **Data Structures and Algorithms - eGela (UPV/EHU)**
- **Introduction to Algorithms, Third edition - Thomas H. Cormen**
- **Data Structures and Algorithms Specialization - Coursera**
- **StackOverFlow**
- **Hackerrank**
- **GeeksForGeeks**