eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Facultad de Informática
Grado de Ingeniería Informática

Documentation of the programming project

Data Structures and Algorithms

Managing a Social Network

Imanol Maraña and Borja Moralejo Tobajas

15 Friday, January 2021

Table of Contents

1. Abstract

In this project we are going to develop a social network from scratch in Java programming language. We will learn in the process of making the different data structures types used in real world applications, their advantages and disadvantages. The main objectives are in a document marked as milestones and we must fulfill the specified points from them.

We have to think and design our data structure for the project, analyzing and calculating the efficiency and measuring the time that takes to finish their tasks, experimentally or by tilde approximation. After that, we will conclude a resolution and then rethink and remake our data structure if we find a better design while studying or after examining our previous designs.

In this document we are reporting our project evolution and development choices, for example our data structures changes or our algorithms functionality. Indirectly we will be reporting our learning process in this subject because we are going to explain our different versions of the project.

The JUnit test cases and UML diagrams are a helping hand explaining our work and the integrated user documentation or guide will help the user know the basic functionality.

Project development over the course

2. First version of the Project

Our first version only implements the first milestone of the description of the Programming Project and some extra sections. How it implements those parts are explained in the following paragraphs, first we will specify what our program currently does.

The first milestone is entirely implemented: It has an initial menu that you can use for interacting with the social network. It can load people and their relationships into the social network from a file inside \res folder and print the people inside the network into a file.
The other implemented sections are: You can search people by their attributes and get the other people that have the same attribute. Also, you can remove a person from the network and clean memory.

2.1. Classes design

This is a brief description of the classes used in this project. Each class has it's full documentation in /doc.

The main classes are: `SocialNetwork` and `DataHolder`

`SocialNetwork` class has the main method and the menu on it.

`DataHolder` class as the name indicates, contains the data of the social network.

`Person` class has the attributes inside and it is the object that has to be stored, as well as its attributes.

Inside the package `adt` we have all the abstract data types that we are using in the project.

Inside `abstractDataTypesImplemented` we have all the classes that are implementing the ADTs.

Inside `exceptions` package we have… exceptions classes.

2.2. Description of the data structures used in the project

We have two main types. Data structures for the attributes and data structures for the relationships. But first we are going to explain how Person class has their attributes stored.

Person class uses two data structures that are going to be explained further down, `DataBlockADT<String, String>` and `NodeADT<String>`. Person has an array of arrays of DataBlockADT for the attributes and a NodeADT for the relationships.

We have chosen this approach because if the user or person decides that it doesn't want to put that attribute, their array length for that attribute is 0 and we know that the user didn't enter any data.

For the attributes:

For the main data storage system we have the following structure:

All data is stored inside DataBlocks objects, they work as a node but they have another functionality. In our case we use a `DataBlock<String, String>`, the class has a key type and a collection element type. The DataBlock holds all elements that have the share key. For example, if we have a datablock holding the "*Donostia*" key String of the attribute home and we make an algorithm to add elements that have the same attribute, in this case, people's id that have a home in Donostia.

With this element we have all the people that are related by an attribute and if we need to search somebody related, we can just use the datablock for getting the ids' that are related. For this reason we need an algorithm and a data structure that can access efficiently to the asked datablock.

Because of that we have the following structure in DataHolder:
It contains two main variables *personHashMap* `HashMapADT<Person(value), String(key)>` and *stringHashMaps* if of the type `HashMapADT<DataBlock<String, String>(value), String(key)>[]` with a length of the number of attributes minus the id because Person class has a unique id with their own getter.

With that type of structure we can have each datablock which stores a unique attribute value and all the related ids hashed by the value that stores. So, if we search for the people that were born in Donostia, we ask the dataholder inside the *stringHashMap* hashmap array in the index assigned for the attribute *birthplace* for the datablock with the key "Donostia". Supposing that the hashmap has the asked datablock, it will return it and we can then proceed to use the second main variable, *personHashMap*.

The variable *personHashMap* is a hashmap of Person with their id (String) as keys. With this hashmap we can search efficiently the Person by their id.

To sum it up, this data structure allows a fast and efficient method for searching related people for the cost of some space, but nowadays it’s easier to expand the capacity than the speed of the processor.

For the relationships:

At the moment of loading a person into the network, a node is created and assigned to that person with that person's ID as it's only content. Therefore, we now have a node assigned to each of the persons loaded, ready to operate with them.

When we add a new friend relationship between two people, the node of each other stores the other person's ID in an ArrayList, using an alphabetical order, so a “link” is made between the two of them. Whenever we want to unlink them, we just need to delete each other's ID from the friend list. Also, there's a counter that indicates the number of links or the number of friends that someone has. If we make a link, that number increases, and if we unlink them, it decreases.

If we want to operate with the friends of anyone stored in our database, as long as the desired people are linked to each other, we can get their node, as well as their information, without any problem. To make this process faster, we previously ordered the friends ID alphabetically, so we just need to make a quick search and we would know if both of them are linked or not. After that, accessing the friends info is as easy as knowing whether they are linked or not, we need to find the link to return it after all.

This way, we can operate with the relationships of people with ease and quickness, as we have an efficient way of storing and searching for info.

2.3.Design and implementation of the methods

We are mentioning and giving a brief description of the method, for more information, please go into the javadoc documents.

In Person.java

Constructor `Person(String combinedData)`:
It decomposes the string and gets the attributes that are separated in a .csv format.

Function `getAtributesRelatedDataBlocks(int attribute)`:
Returns an array with the Datablocks with the given attribute, for example, all their homes.

In DataHolder.java

Function `constructorUseDataBlock(int attribute, String key)`:
This method is used to add the Person into the network, for example we can create a Person object individually but we need to get or create the attribute's datablock for correct network functionality.

Function `getDataBlockOf(int attribute, String key)`:
Is used for getting related people by the given attribute.

Method `addPersonToNetwork(Person p)`:
Links their attributes with the network's datablocks. It uses the constructorUseDataBlock function.

Method `loadFile(String filename, int option)`:
Tries to load the file with the given name and loads people into the network or loads their relationships based on the option parameter.

Function `getPersonByID(String id)`:
Gets the person with the given id from the hashmap, if there's no person with such id, it throws an exception.

Function `searchPeopleByAttribute(int attribute,  String value)`:
Gets the people that have the given attribute.

Method `printIntoFile(String fileName)`:
Prints all the people from the network into the filename.

3. Second version of the project

The second version of the project implements the second milestone of the description of the Programming Project. It also adds more features for the sake of completeness.

The second milestone consists of adding more functionalities to the social network. The following functionalities were implemented:

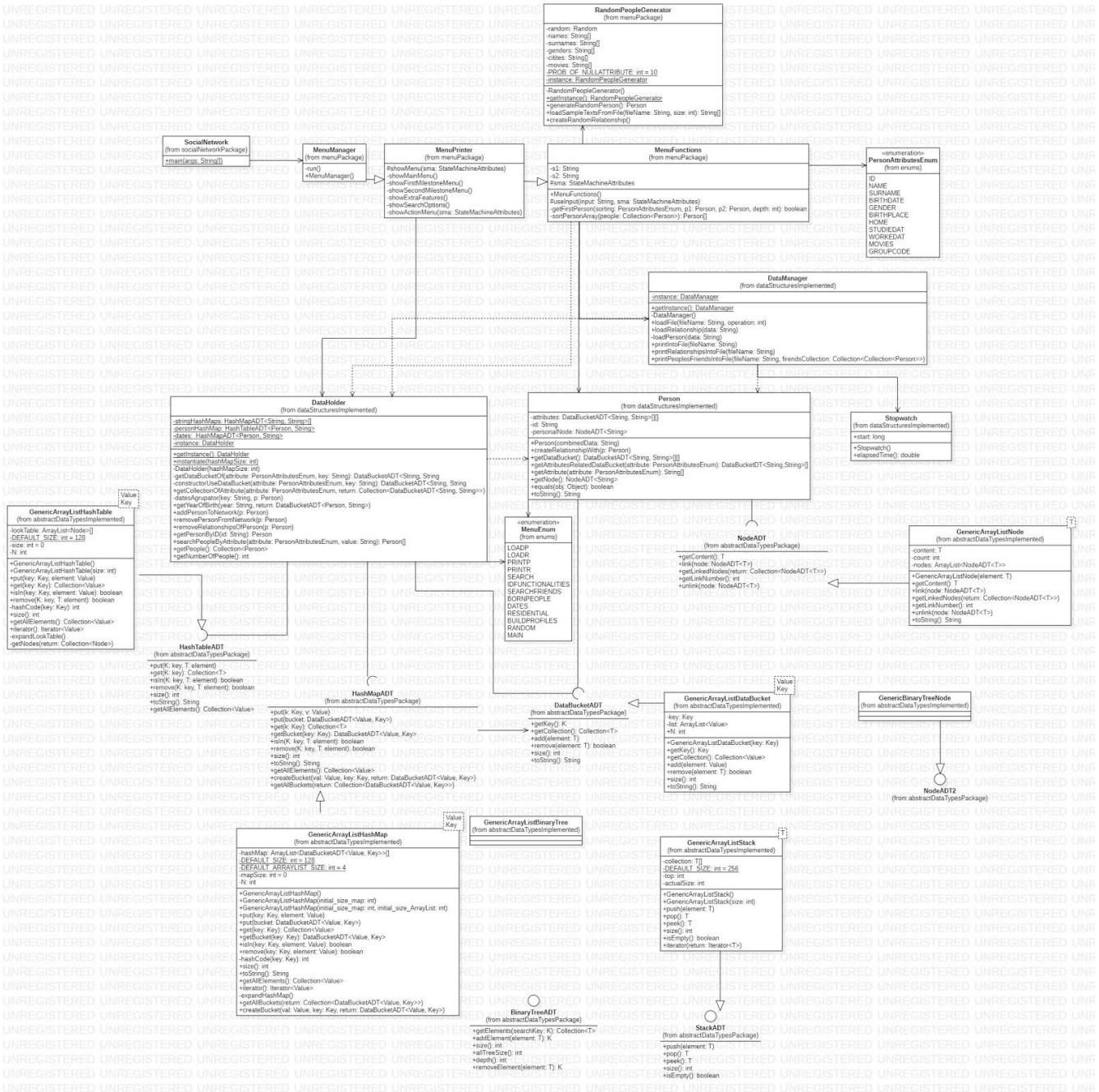
- Search friends using the person's surname. If there are several people with the same surname, print out a list of their friends (if any). Print into console or file.
- Given a city, get all people who were born there. Retrieve Person's id and surnames
- On the social network: retrieve the people who were born between dates D1 and D2, sorted by birthplace, surname, name. We consider only the "year" of the date, disregarding the values of month and day. The order relationship will be implemented according to the lexicographic order (dictionary's order) of the strings used for the attributes.
- Given a set of identifiers in a file named residential.txt, recover the values of the attributes name, surname, birthplace and studiedat of the people on the network whose birthplace matches the hometown of the people who are described in residential.txt. People whose birthplace/hometown is unknown do not affect the result of this operation.
- Two users have the same profile if they match the same collection of favorite movies. Your task is to split the users into classes with the same profile and to build a list of those classes.

The menu is completely reworked and now works using a state machine. This change makes it easier to implement more functionalities in our project. This has no effects with the interaction that the user had.

The extra features added into the project are the following ones:

- Print friends of the requested person
- Print into a file the friends relationships of the social network
- Random people files generator
- Random friendship generator
- Show the number of people in the network

3.1.Classes design



This is a brief description of the classes used in this project. Each class has it's full documentation in /doc.

The main class is: `SocialNetwork` in socialNetworkPackage

`SocialNetwork` class has the main method and starts the menu.

The package menuPackage has the following classes:

- MenuManager, the class with the menu loop.
- MenuPrint, which is the class that prints in console all the information needed for interaction to the user.
- MenuFunctions, here we have all of the functions described in the description of the Programming Project
- And finally we have RandomPeopleGenerator which its name is self explanatory.

In the enums package has the enumerations used in the programming project. MenuEnum and PersonAttributesEnum.

In dataStructuresImplemented package we have Person, DataHolder, DataManager and Stopwatch classes.

- `DataHolder` class as the name indicates, contains the data of the social network and we can access all the data from there and use it in the functions that we want. This class is a singleton.
- `DataManager` loads or prints the data from or into files. This class is a singleton.
- `Person` class has the attributes inside and it is the object that has to be stored, as well as its attributes using dataBuckets.
- Stopwatch class was taken from the resources provided by Professor Javier Dolado.

Inside the package `abstractDataTypesPackage` we have all the abstract data types that we are using in the project.

Inside `abstractDataTypesImplemented` we have all the classes that are implementing the ADTs.

Inside `exceptions` package we have the following Exception inherited classes: AlreadyOnTheCollectionException, ElementNotFoundException, EmptyCollectionException, ImpulsoryAttributeRequiredException and MenuClosedException.

In comparator package we have PersonComparator which has 3 comparators that sort by name, surname and birthplace. Linking them in order we get the comparator composite.

3.2. Description of the data structures used in the project

We have two main types. Data structures for the attributes and data structures for the relationships. But first we are going to explain how Person class has their attributes stored.

Person class uses two data structures that are going to be explained further down, `DataBucketADT<String, String>` and `NodeADT<String>`. Person has an array of arrays of DataBucketADT for the attributes and a NodeADT for the relationships.

We have chosen this approach because if the user or person decides that it doesn't want to put that attribute, their array for that attribute is null and we know that the user didn't enter any data.

For the attributes:

For the main data storage system we have the following structure:

All data is stored inside DataBuckets objects which are the buckets of the HashMap that we use, they work as a node but they have another functionality. In our case we use a `DataBucket<String, String>`, the class has a key type and a collection element type. The DataBucket holds all elements that have the share key. For example, if we have a databucket holding the "*Donostia*" key String of the attribute home and we make an algorithm to add elements that have the same attribute, in this case, people's id that are inside of the databucket have a common attribute, home in Donostia.

With this element we have all the people that are related by an attribute and if we need to search somebody related, we can just use the databuckets for getting the ids' that are related. For this reason we need an algorithm and a data structure that can access efficiently to the asked databucket.

Because of that we have the following structure in DataHolder:
It contains two main variables *personLookTable* `HashTableADT<Person`(value)`, String`(key)`>` and *attributesHashMaps* of the type `HashMapADT<String`(value)`, String`(key)`>[]` with a length of the number of attributes minus the id because Person class has a unique id with their own getter.

With that type of structure we can have each databucket which stores a unique attribute value and all the related ids hashed by the value that stores. So, if we search for the people that were born in Donostia, we ask the dataholder inside the *attributesHashMaps* hashmap array in the index assigned for the attribute *birthplace* for the databucket with the key "Donostia". Supposing that the hashmap has the asked databucket, it will return it and we can then proceed to use the second main variable, *personLookTable* .

The variable *personLookTable* is a hashtable of Person with their id (String) as keys. With this hashtable we can search efficiently the Person by their id.

The hashmap<Person, String> dates holds people born in the given year.

To sum it up, this data structure allows a fast and efficient method for searching related people for the cost of some space, but nowadays it's easier to expand the capacity than the speed of the processor.

For the relationships:

At the moment of loading a person into the network, a node is created and assigned to that person with that person's ID as it's only content. Therefore, we now have a node assigned to each of the persons loaded, ready to operate with them.

When we add a new friend relationship between two people, the node of each other stores the other person's ID in an ArrayList, using an alphabetical order, so a "link" is made between the two of them. Whenever we want to unlink them, we just need to delete each other's ID from the friend list. Also, there's a counter that indicates the number of links or the number of friends that someone has. If we make a link, that number increases, and if we unlink them, it decreases.

If we want to operate with the friends of anyone stored in our database, as long as the desired people are linked to each other, we can get their node, as well as their information, without any problem. To make this process faster, we previously ordered the friends ID alphabetically, so we just need to make a quick search and we would know if both of them are linked or not. After that, accessing the friends info is as easy as knowing whether they are linked or not, we need to find the link to return it after all.

This way, we can operate with the relationships of people with ease and quickness, as we have an efficient way of storing and searching for info.

3.3.Design and implementation of the methods

We are mentioning and giving a brief description of the method, for more information, please go into the javadoc documents.

In Person.java

Constructor `Person(String combinedData)`:
It decomposes the string and gets the attributes that are separated in a .csv format.

Function `getAtributesRelatedDataBuckets(PersonAttributeEnum attribute)`:
Returns an array with the Databucket with the given attribute, for example, all their homes.

In DataHolder.java

Function `constructorUseDataBucket(int attribute, String key)`:
This method is used to add the Person into the network, for example we can create a Person object individually but we need to get or create the attribute's databucket for correct network functionality.

Function `getDataBucketOf(PersonAttributeEnum attribute, String key)`:
Is used for getting related people by the given attribute.

Method `addPersonToNetwork(Person p)`:
Links their attributes with the network's databuckets. It uses the constructorUseDataBucket function.

Method `loadFile(String filename, int option)`:
Tries to load the file with the given. Based on the option parameter, it may use the methods loadPerson, loadRelationship, or load residential with the information readed from the file.

Function `getPersonByID(String id)`:
Get the person with the given id from the hashmap, if there's no person with such id, it throws an exception.

Function `searchPeopleByAttribute(PersonAttributeEnum attribute, String value)`:
Gets the people that have the given attribute.

Method `printIntoFile(String fileName)`:
Prints all the people from the network into the filename.

Implementation of the points of the Programming Project:
From the first Milestone:

1. The application must present an initial menu with the different choices for interacting with the social network.
It uses a state machine menu using MenuManager, MenuPrint and MenuFunctions.

2. On the social network: take the data from a person and add him/her to the network. This function does not read any data from Console. This function receives the data as parameters and adds that information to the network.
Given a string in a .csv value the Person constructor creates a person with the given attributes.

3. Load people files into the network
This uses the method loadFile of DataManager and first creates a Person with the data and then adds the Person into the network using the method addPersonToNetwork in DataHolder. It searches the databuckets for each attribute of person with the same attributes and if it finds one it links the person databuckets with them. It has a cost of O(1) to put the person on the hashmap. If there are conflicts inside the hashmap the cost ups to O(log n)

4. On the social network: print out a listing to a text file of the people on the network.
This uses the method printIntoFile of DataManager. It uses the hashmap's toCollection() method.

5. Load relationships from files.
Given a file with the ID of 2 people in each line, creates a relationship between them.

This uses the method loadFile of DataManager with the option 1. This option reads each line of the file and uses the method loadRelationships.

When creating the relationship between the given people, it uses the createRelationship method in class Person. So, the cost of this operation is O(2log(n)) for each pair of ID in the file

From the second Milestone:

6.On the social network: given a surname, prints the ID and surname of each of the friends of all the people whose surname matches with the given one. It can be printed into a file with a custom or default name or into a console.

This method uses the method searchPeopleByAttribute from DataHolder, and in case that the user wants to print it into a file, uses the method printPeoplesFriendsIntoFile from DataManager, thus having an overall cost of O(1).

7. On the social network: given a city, retrieve all people who were born there. Information to be retrieved for each person is Person's id and Surname.

The implementation of this point is fulfilled with the use of the method searchPeopleByAttribute of DataHolder. It has a cost of O(1) with the use of hashmaps and databuckets.

8. On the social network: retrieve the people who were born between dates D1 and D2, sorted by birthplace, surname, name and by lexicographic order.

Given D1 and D2, iterates between D1 and D2 given that D1 <= D2 and using the hashmap dates retrieves the people born in the values between D1 and D2. It has a cost of O(1) and the hashmap dates is builded when adding the people into the network. For the sorting I have used an Stream processing and use it's sort function with our comparator to sort using birthplaces, surnames and names. We could have used a quicksort or a mergesort, but data stream processing technology can use GPU acceleration for sorting.

9. Given a file named residential.txt, print into the console the name, surname, home and the places of study of the people whose ID is in the file.

It uses the loadFile method's option 2, which is reading each of the lines of the file and using the method loadReasidential with them.

This point uses the method getPeopleByID and searchPeopleByAttribute from class DataHolder, which makes its cost O(1).

10.Two users have the same profile if they match the same collection of favorite movies. Your task is to split the users into classes with the same profile and to build a list of those classes.

This point is implemented using searchPeopleByAttribute method of DataHolder. It has a cost of O(1) with the use of hashmaps and databuckets. That's for the data retrieval part. For the profile building each person needs to get their data retrieved, so with n people, data retrieval cost is O(n).
Then it must sort the movies lexicographically using a quicksort method with the cost O(n log n) with n number of movies. But because people on the network don't usually have a lot of favorite movies, we can disregard that cost. Finally the sorted string is concatenated and put into a hashmap, with the movies concatenation as key and the people as values.

3.4.Changes from the previous version

- Limiting the number of attributes on some attributes, like gender or birthdate to one.
- Separating DataHolder class into DataManager class whose functionality is to load and print data.
- Transforming DataHolder and DataManager into singletons.
- Reworked the menu and now works using state machines.
- Changed the working of hashmaps, now they use databuckets in a way that is more similar to the real hashmaps.
- Created a new method in Person which makes puting some selected attributes into console easier
- Now we use Enumerators for the attributes of the people.

4. Final version of the project

The final version of the project implements the last milestone of the description of the Programming Project plus all the other versions and so this is the complete version of the programming project.

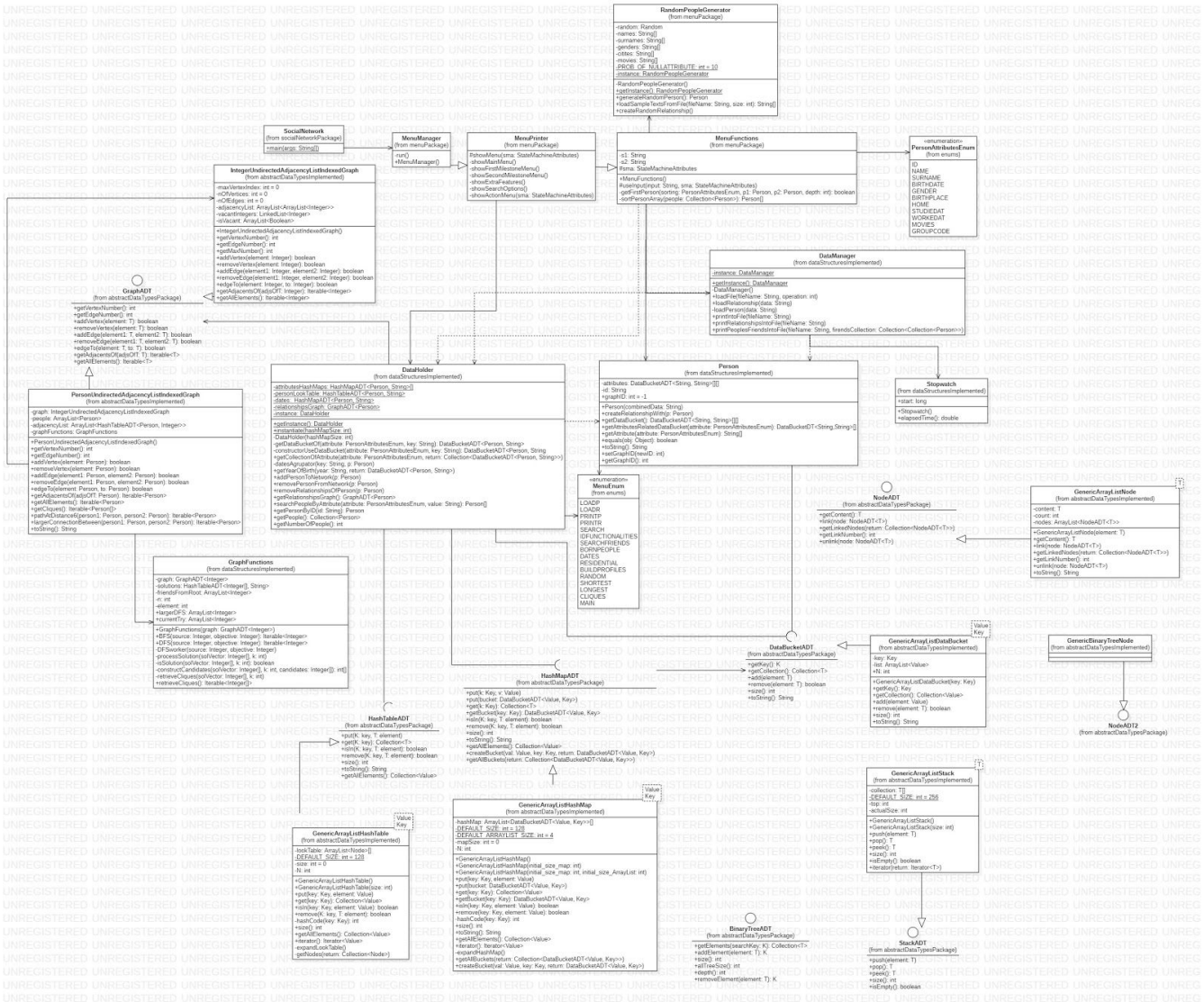
The final milestone consists of creating a graph and the last part from the description of the programming project:

- Retrieve the shortest path that relates two people by friendship with a maximum depth of 6
- Given two people return the longest chain that relates them.
- Retrieve all the cliques of people with more than 4 friends.

Some changes have been made to the data structure and the changes have been analyzed.

With JUnit test plan we have a coverage of 93% of the project. The remaining 7% are some extreme cases like introducing wrong inputs in the menu and some ADT functions that we don’t use.

4.1. Classes design



This is a brief description of the classes used in this project. Each class has it's full documentation in /doc.

The main class is: `SocialNetwork` in socialNetworkPackage

`SocialNetwork` class has the main method and starts the menu.

The package menuPackage has the following classes:

- MenuManager, the class with the menu loop.
- MenuPrint, which is the class that prints in console all the information needed for interaction to the user.
- MenuFunctions, here we have all of the functions described in the description of the Programming Project
- And finally we have RandomPeopleGenerator which its name is self explanatory.

In the enums package has the enumerations used in the programming project. MenuEnum and PersonAttributesEnum.

In dataStructuresImplemented package we have Person, DataHolder, DataManager and Stopwatch classes.

- `DataHolder` class as the name indicates, contains the data of the social network and we can access all the data from there and use it in the functions that we want. This class is a singleton.
- `DataManager` loads or prints the data from or into files. This class is a singleton.
- `Person` class has the attributes inside and it is the object that has to be stored, as well as its attributes using dataBuckets.
- GraphFunctions class has the methods of the final milestone. It uses a GraphADT<Integer> provided by the PersonGraph inside DataHolder.
- Stopwatch class was taken from the resources provided by Professor Javier Dolado.

Inside the package `abstractDataTypesPackage` we have all the abstract data types that we are using in the project.

Inside `abstractDataTypesImplemented` we have all the classes that are implementing the ADTs.

Inside `exceptions` package we have the following Exception inherited classes: AlreadyOnTheCollectionException, ElementNotFoundException, EmptyCollectionException, ImpulsoryAttributeRequiredException and MenuClosedException.

In comparator package we have PersonComparator which has 3 comparators that sort by name, surname and birthplace. Linking them in order we get the comparator composite.

4.2. Description of the data structures used in the project

We have two main types. Data structures for the attributes and data structures for the relationships. But first we are going to explain how Person class has their attributes stored.

Person class uses a data structure that is going to be explained further down, `DataBucketADT<Person, String>`. Person has an array of arrays of DataBucketADT for the attributes and an id for the symbol used in the graph.

We have chosen this approach because if the user or person decides that it doesn't want to put that attribute, their array for that attribute is null and we know that the user didn't enter any data. Using this type of data structure we can achieve the points of the programming project easily and efficiently.

For the attributes:

For the main data storage system we have the following structure:

All data is stored inside DataBuckets objects which are the buckets of the HashMap that we use, they work as a node but they have another functionality. In our case we use a `DataBucket<Person, String>`, the class has a key type and a collection element type. The DataBucket holds all elements that have the share key. For example, if we have a databucket holding the "*Donostia*" key String of the attribute home and we make an algorithm to add elements that have the same attribute, in this case, person's object references that are inside of the databucket have a common attribute, home in Donostia.

With this element we have all the people that are related by an attribute and if we need to search somebody related, we can just use the databuckets for getting their objects that are related. For this reason we need an algorithm and a data structure that can access efficiently to the asked databucket.

Because of that we have the following structure in DataHolder:
It contains two main variables *personLookTable* `HashTableADT<Person(value), String(key)>` and *attributesHashMaps* of the type `HashMapADT<Person(value), String(key)>[]` with a length of the number of attributes minus the id because Person class has a unique id with their own getter.

With that type of structure we can have each databucket which stores a unique attribute value and all the related references hashed by their unique ids. So, if we search for the people that were born in Donostia, we ask the dataholder inside the *attributesHashMaps* hashmap array in the index assigned for the attribute *birthplace* for the databucket with the key "Donostia".

Supposing that the hashmap has the asked databucket, it will return it all the related elements.

The variable *personLookTable* is a hashtable of Person with their id (String) as keys. With this hashtable we can search efficiently the Person by their id.

The hashmap<Person, String> dates holds people born in the given year.

To sum it up, this data structure allows a fast and efficient method for searching related people for the cost of some space, but nowadays it's easier to expand the capacity than the speed of the processor.

For the relationships:

This part has been reworked with minimal changes to other classes. We used to use nodes to relate each person, with each node having their own adjacent list on it. Now that we have learned what graphs are, we have created GraphADT and its proper implementations, `IntegerUndirectedAdjacencyListIndexedGraph` and `PersonUndirectedAdjacencyListIndexedGraph`.

The graph implementation was made with those two classes, IndexGraph class and PersonGraph class for short. The idea while implementing was to symbolize each Person with an Integer and let the graph do calculation with their symbols.

For that we have IndexGraph, which is an implementation using Integers as types. It's a regular graph using integers. You can add and remove elements from the graph and the graph will remember each removed element for later use. The data is stored in an arraylist which is going to be increasing in size while the social network increases in size, and if some users leave the network, their unique graphId's will be stored and then given to other new users. In order to use this graph, the developer must use getIndex() function because this is crucial for that class.

PersonGraph has an IndexGraph that uses as reference when asked to do operation with graphs, but also is an implementation of GraphADT with Person as a type. This class gives the users their unique graph id even if they don't have friends. And when the users leave, their id will be removed, but the managing of those symbols is work of IndexGraph.

In short, if people are added to the network, they will get an unique graph id which is an Integer that works as a symbol in the other class. When asked to create an edge between two elements, the two symbols are going to be used and the index graph is going to create an edge between those two symbols.

The GraphADT<Person> variable that is the graph is contained in DataHolder.java.

4.3. Design and implementation of the methods

We are mentioning and giving a brief description of the method, for more information, please go into the javadoc documents.

In Person.java

Constructor `Person(String combinedData)`:
It decomposes the string and gets the attributes that are separated in a .csv format.

Function `getAtributesRelatedDataBuckets(PersonAttributeEnum attribute)`:
Returns an array with the Databucket with the given attribute, for example, all their homes.

In DataHolder.java

Function `constructorUseDataBucket(int attribute, String key)`:
This method is used to add the Person into the network, for example we can create a Person object individually but we need to get or create the attribute's databucket for correct network functionality.

Function `getDataBucketOf(PersonAttributeEnum attribute, String key)`:
Is used for getting related people by the given attribute.

Method `addPersonToNetwork(Person p)`:
Links their attributes with the network's databuckets. It uses the constructorUseDataBucket function.

Method `loadFile(String filename, int option)`:
Tries to load the file with the given. Based on the option parameter, it may use the methods loadPerson, loadRelationship, or load residential with the information readed from the file.

Function `getPersonByID(String id)`:
Get the person with the given id from the hashmap, if there's no person with such id, it throws an exception.

Function `searchPeopleByAttribute(PersonAttributeEnum attribute, String value)`:
Gets the people that have the given attribute.

Method `printIntoFile(String fileName)`:
Prints all the people from the network into the filename.

Implementation of the points of the Programming Project:

From the first Milestone:

1. The application must present an initial menu with the different choices for interacting with the social network.
It uses a state machine menu using MenuManager, MenuPrint and MenuFunctions.

2. On the social network: take the data from a person and add him/her to the network. This function does not read any data from Console. This function receives the data as parameters and adds that information to the network.
Given a string in a .csv value the Person constructor creates a person with the given attributes.

3. Load people files into the network
This uses the method loadFile of DataManager and first creates a Person with the data and then adds the Person into the network using the method addPersonToNetwork in DataHolder. It searches the databuckets for each attribute of person with the same attributes and if it finds one it links the person databuckets with them. It has a cost of O(1) to put the person on the hashmap. If there are conflicts inside the hashmap the cost ups to O(log n)

4. On the social network: print out a listing to a text file of the people on the network.
This uses the method printIntoFile of DataManager. It uses the hashmap's toCollection() method.

5. Load relationships from files.
Given a file with the ID of 2 people in each line, creates a relationship between them.

This uses the method loadFile of DataManager with the option 1. This option reads each line of the file and uses the method loadRelationships.

When creating the relationship between the given people, it uses the createRelationship method in class Person. So, the cost of this operation is O(1) for each pair of ID in the file

From the second Milestone:

6.On the social network: given a surname, prints the ID and surname of each of the friends of all the people whose surname matches with the given one. It can be printed into a file with a custom or default name or into a console.

This method uses the method searchPeopleByAttribute from DataHolder, and in case that the user wants to print it into a file, uses the method printPeoplesFriendsIntoFile from DataManager, thus having an overall cost of O(1).

7. On the social network: given a city, retrieve all people who were born there. Information to be retrieved for each person is Person's id and Surname.

The implementation of this point is fulfilled with the use of the method searchPeopleByAttribute of DataHolder. It has a cost of O(1) with the use of hashmaps and databuckets.

8. On the social network: retrieve the people who were born between dates D1 and D2, sorted by birthplace, surname, name and by lexicographic order.

Given D1 and D2, iterates between D1 and D2 given that D1 <= D2 and using the hashmap dates retrieves the people born in the values between D1 and D2. It has a cost of O(1) and the hashmap dates is builded when adding the people into the network. For the sorting I have used an Stream processing and use it's sort function with our comparator to sort using birthplaces, surnames and names. We could have used a quicksort or a mergesort, but data stream processing technology can use GPU acceleration for sorting.

9. Given a file named residential.txt, print into the console the name, surname, home and the places of study of the people whose ID is in the file.

It uses the loadFile method's option 2, which is reading each of the lines of the file and using the method loadReasidential with them.

This point uses the method getPeopleByID and searchPeopleByAttribute from class DataHolder, which makes its cost O(1).

10.Two users have the same profile if they match the same collection of favorite movies. Your task is to split the users into classes with the same profile and to build a list of those classes.

This point is implemented using searchPeopleByAttribute method of DataHolder. It has a cost of O(1) with the use of hashmaps and databuckets. That's for the data retrieval part. For the profile building each person needs to get their data retrieved, so with n people, data retrieval cost is O(n).
Then it must sort the movies lexicographically using a quicksort method with the cost O(n log n) with n number of movies. But because people on the network don't usually have a lot of favorite movies, we can disregard that cost. Finally the sorted string is concatenated and put into a hashmap, with the movies concatenation as key and the people as values.

From the final milestone:

11.Six degrees of separation is the theory that everyone on Earth is six or fewer steps away, by way of introduction, from any other person in the world, so that a chain of "a friend of a friend" statements can be made to connect any two people in a maximum of six steps (or five intermediaries). On the social network: given two people of the network, your task is to retrieve the shortest chain that relates them

This point is implemented in GraphFunctions.java. As this function is the only one that works with BFS, the BFS has been made to fulfil the conditions of being at a distance less than 6 and to retrieve a path between a single source and a single end.

The method uses a queue to check which is the next element to use while searching for an objective integer, which would later indicate whether the path has been found or not. It also uses an array of boolean to check whether the element has already been used previously to prevent infinite loops and a final array of Integers containing a list with the previous element from which the current one proceeds.Once the first appearance of the objective has been found, a path to that point is set as the shortest path.

The method pathDistance6(Person, Person) returns an Iterable<Person> in 3 possible ways: If one of the inputs isn`t in the network or there's not a path at distance less than 6 between both of them, it will return null; If both are the same, it will return an Iterable<Person> of size 1 containing only one appearance of the person; and if it's not any of the previous, returns a path going from the first input to the second, containing at most a chain of 6 people including both the source and the objective.

12.On the social network: given two people, recover the largest chain of different people linking them (duplicate intermediaries are not allowed). Use backtracking

This point is implemented in GraphFunctions.java. The method returns an Iterable<Integer> which later will be used by LongestMethod(Person, Person) in the main graph class.

The method uses a recursive method called from a master method, which sets an initialization for the first iteration, which then implements backtracking searches for more solutions as there may be a longer connection in the graph if other connections are selected.

It works iterating again and again over all the elements of the graph until all the elements connected to the source are used. This process is made using the adjacents of every element in the graph and once that element has been used, it's added to a list where it will be stated whether that element has been used or not. If the desired element is found, there are 3 possible actions to take. If there has never been a previous encounter, it initializes the array where the solution will be placed. If it has already occurred and the size of the current try is greater than the previous one, the array is replaced with a new one containing the larger one. And if neither of these two happen, it does nothing.

This method returns null if at least one of the 2 inputs are not in the network or there's no connection at all between them; It will return an Iterable<Integer> of size 1 with the only element in it as one of the inputs; and it will return an Iterable<Integer> of size 2 or greater if neither of the previous conditions are fulfilled. This is set as a path going from the first input to the second following a path of people

13.On the social network: retrieve all the cliques of friends (crews) with more than 4 friends. A clique is a group of friends in which each person has friendship with each other. Use backtracking.

This point is implemented in GraphFunctions.java. The function retrieveCliques returns an Iterable<Integer[]> in which each Integer[] is a clique with the symbols of the people that form the clique. Then PersonGraph will take each symbol and look up the corresponding Person that it is.

The function uses the integer symbol graph of PersonGraph. In order to check whether or not a Person is in a clique, we must iterate all over the elements in the graph. Then we use backtracking to resolve the problem and the backtracking function has the Skiena structure
.
First we grab the adjacency list of the iterating element i. That list tells us about the maximum size that the clique can have, as much as the number of elements in the list, so we create an array of Integers of that size. That array is the solution vector and each Integer inside of it will be a vector from the clique.
In the backtracking part first we check if the elements in the solution vector are in fact a solution, checking if a bigger clique can be constructed. If a bigger clique can be made, then it's not a solution. If there are no more edges to check, or is the biggest clique that it can be formed, it's a solution and it's added into the solutions hashtable, which only contains unique cliques sorted by an ascending order. For example, the clique composed of vertices 1, 2, 3 and 4 or 4, 2, 1 and 3 is the same, but in backtracking we would have all the possible combinations, and we are not interested in that, we only want the unique clique. That sorting and hashtable takes care of that in the processSolution function.

The construct candidates function grabs a vertex from the friends list and then checks if that vertex has all the elements in the solution vector as adjacent. If it has all the elements as adjacents, the recursion continues.

When the iteration of all the elements in the graph has been completed, the solution is all the elements that are inside the solution hashtable. Then PersonGraph is going to create an Iterable<Person[]> for MenuFunctions.java to work.

This function is very costly and when the cliques grow in size, the more costly it becomes. Each time a clique is found, it must check for connection n times, with n the size of the clique, with each candidate. Say we have a group of 5 people and a clique of 4, and the leftover person has friendship with all but 2. We have to check n factorial times, so the cost of this function in the worst case is O(n!) which is NP. Our approach is not efficient. We end up checking for the same cliques all the time and that could be improved. For example removing combinations of the sequence of the clique.

4.4.Changes from the previous version

- The relationships are managed with a GraphADT instead of nodes.
- DataBuckets<String, String> changed to DataBuckets<Person, String> for faster functions, now it doesn't have to search for person by id.
- Implemented points 11, 12 and 13.

4.5.Analysis

We have tested different options for the selection of types and classes in the data structure and measured the time needed for the completion of the JUnit `TestingMenuTimes`.

We have tested changing the implementation of databuckets and changing the type that uses.

Type 1 is implementation with ArrayList and type String
Type 2 is implementation with ArrayList and type Person
Type 3 is implementation with BinarySearchTree and type String
Type 4 is implementation with BinarySearchTree and type Person

Type	Time
Type 1	74.072 s
Type 2	92.934 s
Type 3	46.885 s
Type 4	48.489 s

Now a brief explanation why we have chosen to stay with the worst time, the Type 2.
The most time that the program is running is while loading people, then the operation with the data and people is a small percentage but the time that the function takes is the most important part..

Type 2 takes longer loading than any other type, but then while doing operations is a lot faster, the other took 2 seconds with 150k people but this just stayed for 0.3s. So if this project is an imaginary social network, then the users would be using it and doing operations like searches and related. Loading the network is not such a big problem because it is going to be on all the time. But if the operations take long to process, the user's requests are going to pile up and the user will experience lag.

5. Conclusions

The main objective of the programming project was successfully completed, we have learned the basics of data structures and algorithms that are the base of the majority of the projects. In this report we describe the evolution of our designs and our gained experience developing projects like this.

By taking part in this project, we have learned how to implement other data structures such as hashmaps, hash tables, binary trees, graphs and recursive functions like backtracking. Their usefulness is crucial for this project and for our development as engineers.

We have also learnt to make a successful teamwork using github, a resource that allows us to share the same project on different computers. In order to create a major programming project we have to coordinate our ideas, decisions and work for the common objective of developing an efficient, working and clear project. Working in groups is difficult if you want to have everyone doing their work on their own. It must exist an organized workflow in order to achieve the common goal.

There are, however, points in the project that must be polished. The Unified Modeling Languaged diagram it's a little bit confusing due to the scale of the project and JUnit doesn't take care of all the blindspots of the project. Sturdiness hasn't been taken care of, the user can freely introduce an unvalid input and crash the execution of the program.

But on the modularity aspect, we are very sure our project can handle any kind of modulation and changes because we are using ADTs as base variables and we are using an state machine menu. If any developer wants to add a functionality it's as easy as adding an enumeration and adding a switch statement.

We think our project is pretty based on modularity of the different data structures learned in the assignment and on the efficiency of the functions.

6. Annexes

6.1. Notes of the meetings

First meeting:
We discussed the data structure that our project was going to have, we decided that it needed a fast accessible data structure that doesn't need to do many comparisons, so we thought about an indexed structure, and from there we found out that hashmaps work similar to that.

Second meeting:
We divided the work we had to do for this project. Imanol would be working on relationships and the menu and Borja on the Person and the data storage structure or system.

Third meeting:
We shared our advancements and our problems, we sorted out the majority of them.

Fourth meeting:
We explained to each other how exactly our part of the project worked.

Fifth meeting:
We discussed the new data structure that our project had to change for the second milestone. It turns out that we didn't have to change as much as we thought.

Sixth meeting:
We divided the work and we kept the main problems in common to solve.

Seventh meeting:
We have decided to leave the binary tree implementation for the next milestone.
.
Eighth meeting: We talked about the implementation of the graph, we will use a base, a GraphADT and develop versions of the graph for it, each one making their version.

Ninth meeting: We divided the work left to do and each one started developing the points.

6.2. User guide

This is the user guide of our programming project:

The user has 4 main options:

1. First milestone part. This part loads and prints data from the network or to the network.
   1.1. Load people from file:
   Loads a text file located in the res directory given file name. That file must not contain information with more than 11 attributes defined by “,”. Then, the information within the file is used to create a person, which will be added to the social network.

   1.2. Load relationships from a file:
   Loads a text file located in the res directory given file name. That file must not contain information with more than 2 attributes defined by “,”. Then, the information within the file is used to create a friend relationship between 2 people, which will be added to the social network.

   1.3. Print people into a file:
   Gets the information of all the people added into the network and prints it into a file located in the res directory with a given name.

   1.4. Print relationships into a file (Extra):
   Gets the information of all the people added into the network and prints their relationships into a file located in the res directory with a given name.

2. Second milestone part. This part has the points that the programming project description explicitly says that our project must implement.
    2.1. Point 6: Search friends by surname:

        On the social network: given a surname, get all the people that have that surname and print all the friends that they have in the social network into a file or the console. File can have a custom name or a default name. Information to be printed is ID and surname

        The user must input the people's surname and choose an option between go back, print to console, or print to file with custom or default name. If chosen custom name, must also introduce a name for the file.

    2.2. Point 7: Get people born in

        On the social network: given a city, retrieve all people who were born there. Information to be retrieved for each person is Person's name, surname, their home and were they studied

        The user only needs to input a city name.

    2.3. Point 8: Get people born between D1 and D2

        On the social network: retrieve the people who were born between dates D1 and D2, sorted by birthplace, surname, name and by lexicographic order.

        The user must give two dates in order to get the wanted functionality printed in the console.

    2.4. Point 9: Residential

        Given a set of identifiers in a file named residential.txt, recover the values of the attributes name, surname, birthplace and studiedat of the people on the network whose birthplace matches the hometown of the people who are described in residential.txt.

        Prints the name, surname, home and study places of the people whose home matches the birthplace of those that are in the Residential.txt file into the console.

    2.5. Point 10: Build profiles

        Two users have the same profile if they match the same collection of favorite movies. Your task is to split the users into classes with the same profile and to build a list of those classes.

        Choosing this option it shows from the console all the classes without any other input. The user must enter any character or string to continue with the program.

3. Third milestone or final examination

3.1. Point 11: Shortest path

Given two people's IDs, prints into the console the path that connects them if there's a path in between them that has a connection of less or equal than 6 people.

It prints the IDs of the people forming the connecting starting in the first person and ending in the last one.

3.2. Point 12: Longest path

Given two people's IDs, prints into the console the longest path that connects them using the relationships set in the graph of the network.

It prints the IDs of the people forming the connecting starting in the first person and ending in the last one.

3.3. Point 13: Cliques

Prints into the console the amount of total cliques formed of at least 4 people in the network. It will only take into account the biggest clique involving different people, so if there's a clique of 5 people, there won't be also 5 cliques of 4 people.

4. This has extra features of the programming project such as random people generation or an advanced people search

4.1. Search people by the given attribute

Given a number associated with an attribute and a value, searches for any person with the given value in the given attribute and prints its information. If the attribute is not ID, it may print more than one person.

4.2. Work with a single person. Add or remove a person by id.

4.2.1. Add person into the network

Given the information of a person in a correct order, creates and adds that person into the social network. That information must not contain more than 11 attributes, but must have at least 1 attribute: the ID. If adding more than one information to a single attribute is desired, put a ";" between the information.

4.2.2. Remove person by ID:

Given the ID, deletes a person with that ID from the network if it exists.

4.2.3. Show friends of the given ID

With the given ID the program searches for that person and prints it's friends.

4.3. Generate random people or relationships

4.3.1. Generate random people

Input a number and that will be the number of people that the program will try to generate successfully. Sometimes the randomness isn't good enough and some id's are repeated and because of that the randomly generated people are always less or equal than the input.

4.3.2. Generate random relationships

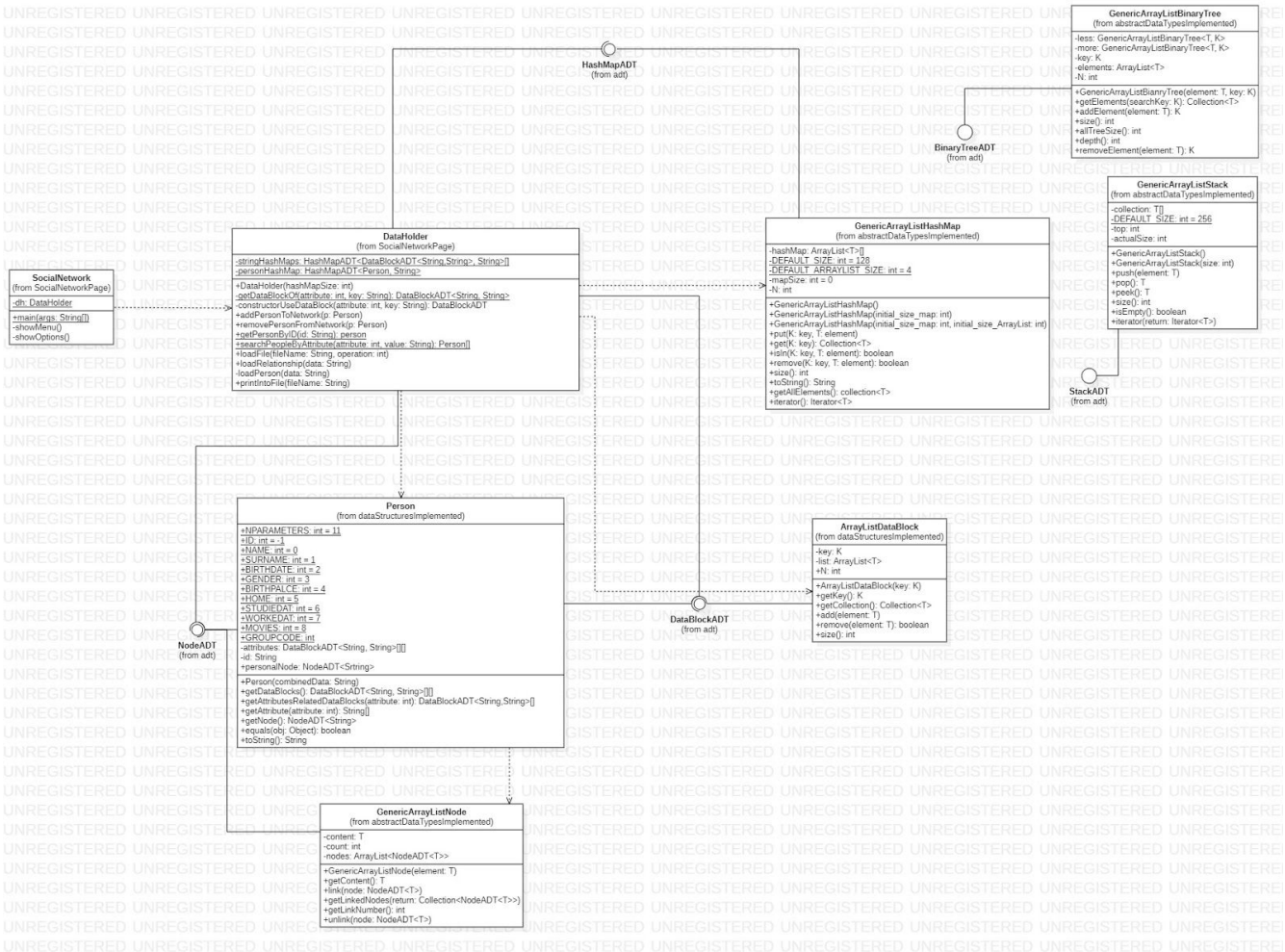
Input a number and that will be the number of relationships that the program will try to generate successfully. Sometimes the randomness isn't good enough and some friendships are repeated and because of that the randomly generated people are always less or equal than the input.

4.4. Show number of people in the network

6.3. UML

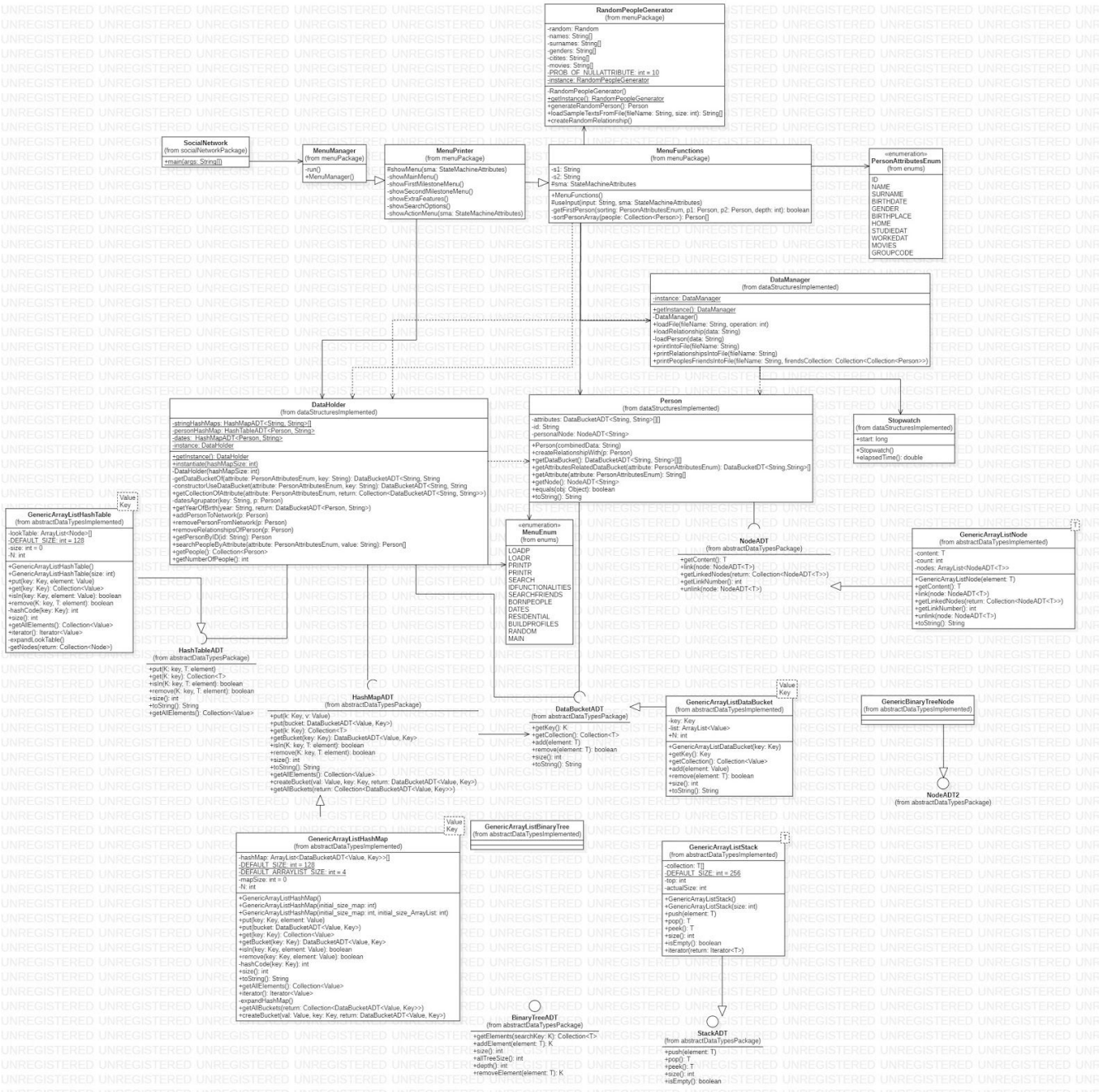
Version 1(old):

Inside the project folder we have the class diagrams. Here is a picture.

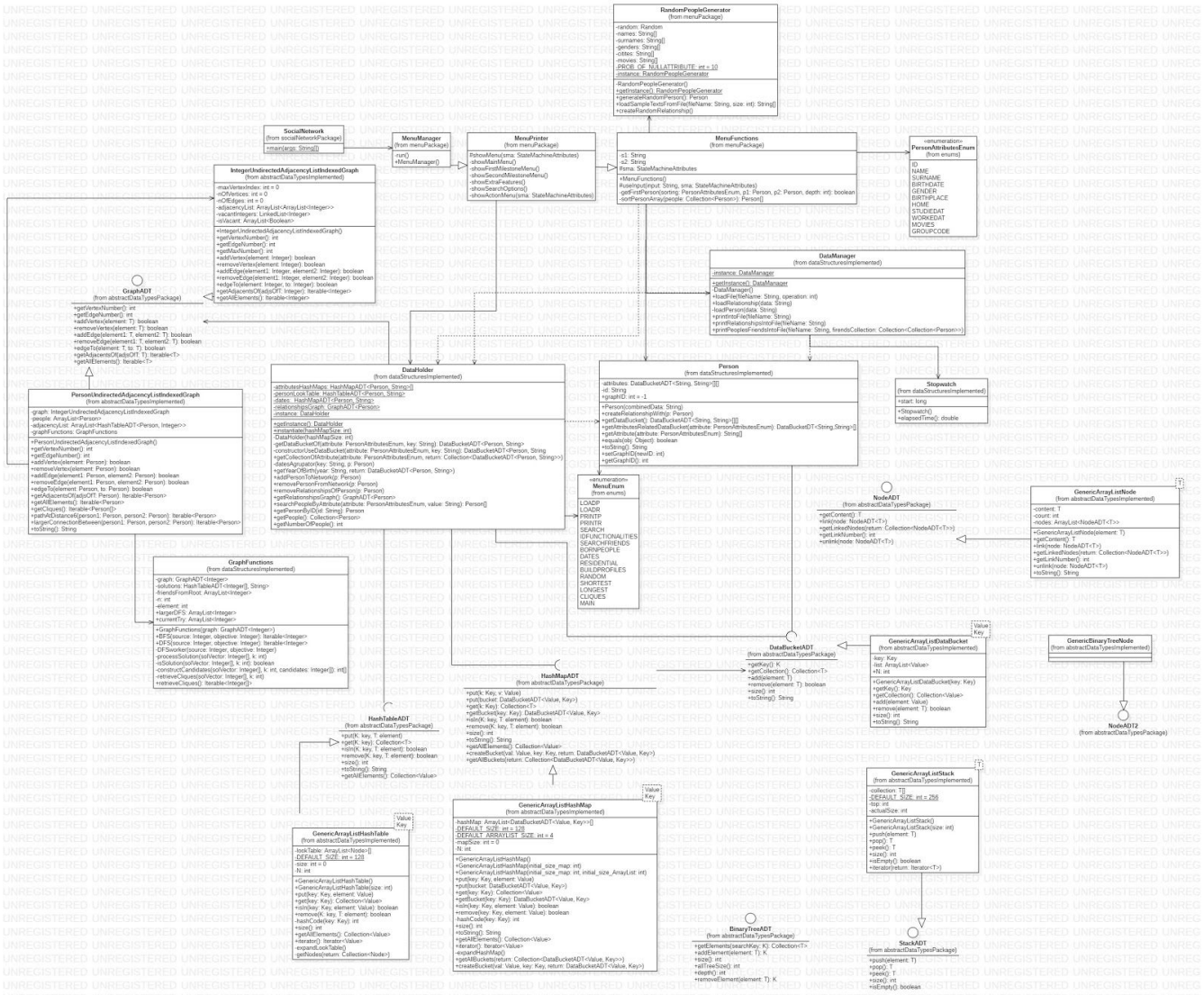


We recommend opening the .mdj file for a detailed view.

Version 2:


RandomPeopleGenerator
(from menuPackage)
-random: Random
-names: String[]
-surnames: String[]
-genders: String[]
-cities: String[]
-movies: String[]
-PROB_OF_NULLATTRIBUTE: int = 10
-instance: RandomPeopleGenerator
-RandomPeopleGenerator()
+getInstance(): RandomPeopleGenerator
+generateRandomPerson(): Person
+loadSampleTextsFromFile(fileName: String, size: int): String[]
+createRandomRelationship()
SocialNetwork
(from socialNetworkPackage)
+main(args: String[])
MenuManager
(from menuPackage)
-run()
+MenuManager()
MenuPrinter
(from menuPackage)
#showMenu(sma: StateMachineAttributes)
-showMainMenu()
-showFirstMilestoneMenu()
-showSecondMilestoneMenu()
-showExtraFeatures()
-showSearchOptions()
-showActionMenu(sma: StateMachineAttributes)
MenuFunctions
(from menuPackage)
-s1: String
-s2: String
#sma: StateMachineAttributes
+MenuFunctions()
#useInput(input: String, sma: StateMachineAttributes)
-getFirstPerson(sorting: PersonAttributesEnum, p1: Person, p2: Person, depth: int): boolean
-sortPersonArray(people: Collection<Person>): Person[]
«enumeration»
PersonAttributesEnum
(from enums)
ID
NAME
SURNAME
BIRTHDATE
GENDER
BIRTHPLACE
HOME
STUDIEDAT
WORKEDAT
MOVIES
GROUPCODE
DataManager
(from dataStructuresImplemented)
-instance: DataManager
+getInstance(): DataManager
-DataManager()
+loadFile(fileName: String, operation: int)
+loadRelationship(data: String)
-loadPerson(data: String)
+printIntoFile(fileName: String)
+printRelationshipsIntoFile(fileName: String)
+printPeoplesFriendsIntoFile(fileName: String, firendsCollection: Collection<Collection<Person>>)
DataHolder
(from dataStructuresImplemented)
-stringHashMaps: HashMapADT<String, String>[]
-personHashMap: HashTableADT<Person, String>
-dates: HashMapADT<Person, String>
-instance: DataHolder
+getInstance(): DataHolder
+instantiate(hashMapSize: int)
-DataHolder(hashMapSize: int)
-getDataBucketOf(attribute: PersonAttributesEnum, key: String): DataBucketADT<String, String
-constructorUseDataBucket(attribute: PersonAttributesEnum, key: String): DataBucketADT<String, String
+getCollectionOfAttribute(attribute: PersonAttributesEnum, return: Collection<DataBucketADT<String, String>>)
-datesAgrupator(key: String, p: Person)
+getYearOfBirth(year: String, return: DataBucketADT<Person, String>)
+addPersonToNetwork(p: Person)
+removePersonFromNetwork(p: Person)
+removeRelationshipsOfPerson(p: Person)
+getPersonByID(id: String): Person
+searchPeopleByAttribute(attribute: PersonAttributesEnum, value: String): Person[]
+getPeople(): Collection<Person>
+getNumberOfPeople(): int
Person
(from dataStructuresImplemented)
-attributes: DataBucketADT<String, String>[][]
-id: String
-personalNode: NodeADT<String>
+Person(combinedData: String)
+createRelationshipWith(p: Person)
+getDataBucket(): DataBucketADT<String, String>[][]
+getAttributesRelatedDataBucket(attribute: PersonAttributesEnum): DataBucketDT<String,String>[]
+getAttribute(attribute: PersonAttributesEnum): String[]
+getNode(): NodeADT<String>
+equals(obj: Object): boolean
+toString(): String
Stopwatch
(from dataStructuresImplemented)
+start: long
+Stopwatch()
+elapsedTime(): double
«enumeration»
MenuEnum
(from enums)
LOADP
LOADR
PRINTP
PRINTR
SEARCH
IDFUNCTIONALITIES
SEARCHFRIENDS
BORNPEOPLE
DATES
RESIDENTIAL
BUILDPROFILES
RANDOM
MAIN
GenericArrayListHashTable
(from abstractDataTypesImplemented)
Value
Key
-lookTable: ArrayList<Node>[]
-DEFAULT_SIZE: int = 128
-size: int = 0
-N: int
+GenericArrayListHashTable()
+GenericArrayListHashTable(size: int)
+put(key: Key, element: Value)
+get(key: Key): Collection<Value>
+isIn(key: Key, element: Value): boolean
+remove(K: key, T: element): boolean
-hashCode(key: Key): int
+size(): int
+getAllElements(): Collection<Value>
+iterator(): Iterator<Value>
-expandLookTable()
-getNodes(return: Collection<Node>)
HashTableADT
(from abstractDataTypesPackage)
+put(K: key, T: element)
+get(K: key): Collection<T>
+isIn(K: key, T: element): boolean
+remove(K: key, T: element): boolean
+size(): int
+toString(): String
+getAllElements(): Collection<Value>
HashMapADT
(from abstractDataTypesPackage)
+put(k: Key, v: Value)
+put(bucket: DataBucketADT<Value, Key>)
+get(k: Key): Collection<T>
+getBucket(key: Key): DataBucketADT<Value, Key>
+isIn(K: key, T: element): boolean
+remove(K: key, T: element): boolean
+size(): int
+toString(): String
+getAllElements(): Collection<Value>
+createBucket(val: Value, key: Key, return: DataBucketADT<Value, Key>)
+getAllBuckets(return: Collection<DataBucketADT<Value, Key>>)
DataBucketADT
(from abstractDataTypesPackage)
+getKey(): K
+getCollection(): Collection<T>
+add(element: T)
+remove(element: T): boolean
+size(): int
+toString(): String
GenericArrayListDataBucket
(from abstractDataTypesImplemented)
Value
Key
-key: Key
-list: ArrayList<Value>
+N: int
+GenericArrayListDataBucket(key: Key)
+getKey(): Key
+getCollection(): Collection<Value>
+add(element: Value)
+remove(element: T): boolean
+size(): int
+toString(): String
NodeADT
(from abstractDataTypesPackage)
+getContent(): T
+link(node: NodeADT<T>)
+getLinkedNodes(return: Collection<NodeADT<T>>)
+getLinkNumber(): int
+unlink(node: NodeADT<T>)
GenericArrayListNode
(from abstractDataTypesImplemented)
-content: T
-count: int
-nodes: ArrayList<NodeADT<T>>
+GenericArrayListNode(element: T)
+getContent(): T
+link(node: NodeADT<T>)
+getLinkedNodes(return: Collection<NodeADT<T>>)
+getLinkNumber(): int
+unlink(node: NodeADT<T>)
+toString(): String
GenericBinaryTreeNode
(from abstractDataTypesImplemented)
NodeADT2
(from abstractDataTypesPackage)
GenericArrayListHashMap
(from abstractDataTypesImplemented)
Value
Key
-hashMap: ArrayList<DataBucketADT<Value, Key>>[]
-DEFAULT_SIZE: int = 128
-DEFAULT_ARRAYLIST_SIZE: int = 4
-mapSize: int = 0
-N: int
+GenericArrayListHashMap()
+GenericArrayListHashMap(initial_size_map: int)
+GenericArrayListHashMap(initial_size_map: int, initial_size_ArrayList: int)
+put(key: Key, element: Value)
+put(bucket: DataBucketADT<Value, Key>)
+get(key: Key): Collection<Value>
+getBucket(key: Key): DataBucketADT<Value, Key>
+isIn(key: Key, element: Value): boolean
+remove(key: Key, element: Value): boolean
-hashCode(key: Key): int
+size(): int
+toString(): String
+getAllElements(): Collection<Value>
+iterator(): Iterator<Value>
-expandHashMap()
+getAllBuckets(return: Collection<DataBucketADT<Value, Key>>)
+createBucket(val: Value, key: Key, return: DataBucketADT<Value, Key>)
GenericArrayListBinaryTree
(from abstractDataTypesImplemented)
BinaryTreeADT
(from abstractDataTypesPackage)
+getElements(searchKey: K): Collection<T>
+addElement(element: T): K
+size(): int
+allTreeSize(): int
+depth(): int
+removeElement(element: T): K
GenericArrayListStack
(from abstractDataTypesImplemented)
-collection: T[]
-DEFAULT_SIZE: int = 256
-top: int
-actualSize: int
+GenericArrayListStack()
+GenericArrayListStack(size: int)
+push(element: T)
+pop(): T
+peek(): T
+size(): int
+isEmpty(): boolean
+iterator(return: Iterator<T>)
StackADT
(from abstractDataTypesPackage)
+push(element: T)
+pop(): T
+peek(): T
+size(): int
+isEmpty(): boolean

Final version:



6.4. JUnit

We have used the library JUnit for the test plan of our project. It tests the loading, printing and storing methods of the DataHolder. It tests the datablock's methods and attribute obtention. Said test plan has a coverage of 93% of the project.

JUnit is the base for testing that the work made is working properly. And it's even more important if we are constantly making changes in our project. For that reason our JUnit folder is full of tests. It tests the bases of the project and if the tests fails, we know that we must fix it, otherwise the whole project would crumble over it's code.

Search in the project files for more information, in /JUnitTests/test/