Documentation of the Programming Project

Group Members: Imanol Maraña and Borja Moralejo

Date: Monday, 19 October 2020

Subject: Data Structure and Algorithms

Degree: Computer Science

Academic Year: 2nd

School: Facultad de Ingeniería Informática UPV

Table of Contents

1. Introduction

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.

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:

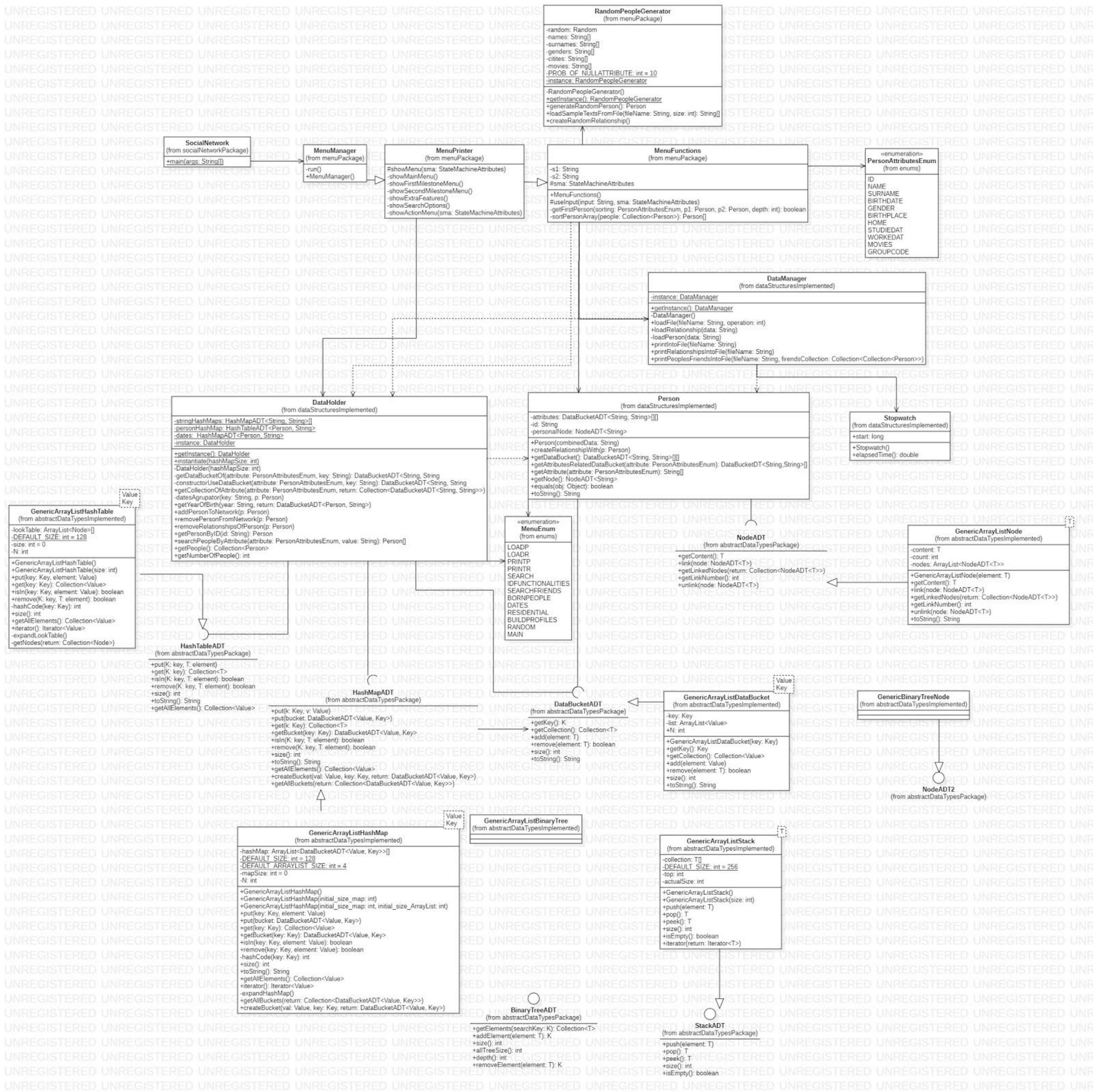
- ❖ Search friends using the person's surname. If there are several people with the same surname, print out a list of their friends (if any). Print into console or file.
- ❖ Given a city, get all people who were born there. Retrieve Person's id and surnames
- ❖ On the social network: retrieve the people 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


RandomPeopleGenerator (from menuPackage)
-random: Random
-names: String[]
-surnames: String[]
-genders: String[]
-citites: 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
Value Key
GenericArrayListHashTable (from abstractDataTypesImplemented)
-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>)
«enumeration» MenuEnum (from enums)
LOADP
LOADR
PRINTP
PRINTR
SEARCH
IDFUNCTIONALITIES
SEARCHFRIENDS
BORNPEOPLE
DATES
RESIDENTIAL
BUILDPROFILES
RANDOM
MAIN
NodeADT (from abstractDataTypesPackage)
+getContent(): T
+link(node: NodeADT<T>)
+getLinkedNodes(return: Collection<NodeADT<T>>)
+getLinkNumber(): int
+unlink(node: NodeADT<T>)
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
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
Value Key
GenericArrayListDataBucket (from abstractDataTypesImplemented)
-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
GenericBinaryTreeNode (from abstractDataTypesImplemented)
NodeADT2 (from abstractDataTypesPackage)
Value Key
GenericArrayListHashMap (from abstractDataTypesImplemented)
-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
T
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

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.

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. 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.

5. Conclusions

By taking part in this project, we’ve learnt how to deal with different types of data management, such as data buckets or binary trees. We’ve also learnt to make a successful teamwork using github, a resource that allows us to share the same project on different computers.

6. Annexes

6.1. User guide

This is the user guide of our programming project:

The user has 3 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. This has extra features of the programming project such as random people generation or an advanced people search

  3.1. 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.

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

    3.2.1. 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.

    3.2.2. Remove person by ID:

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

    3.2.3. Show friends of the given ID

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

  3.3. Generate random people or relationships

    3.3.1. 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.

    3.3.2. Generate random relationships

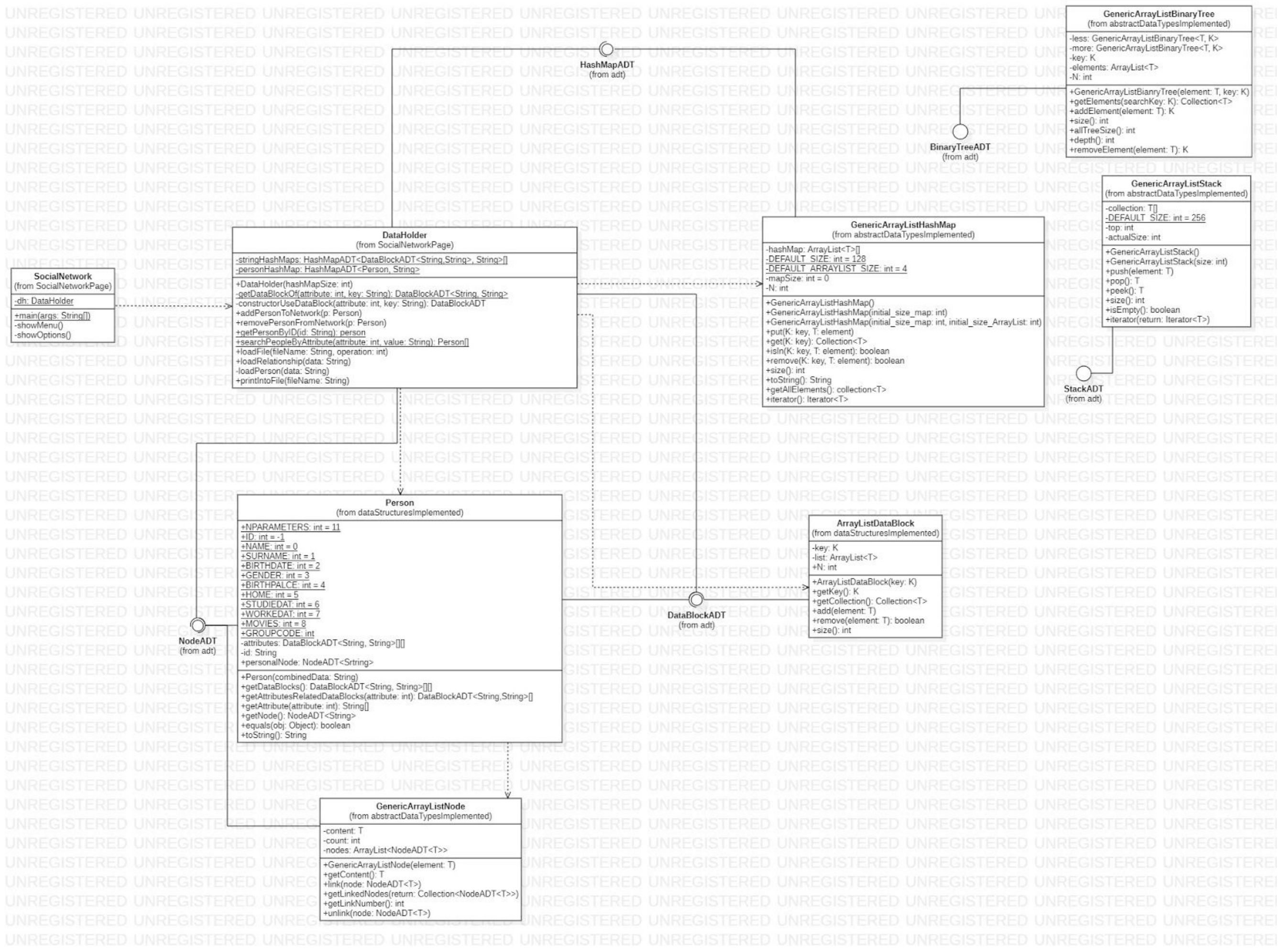
    Input a number and that will be the number of relationships that the program will try to generate successfully. Sometimes the randomness isn't good enough and some friendships are repeated and because of that the randomly generated people are always less or equal than the input.

  3.4. Show number of people in the network

6.2. UML

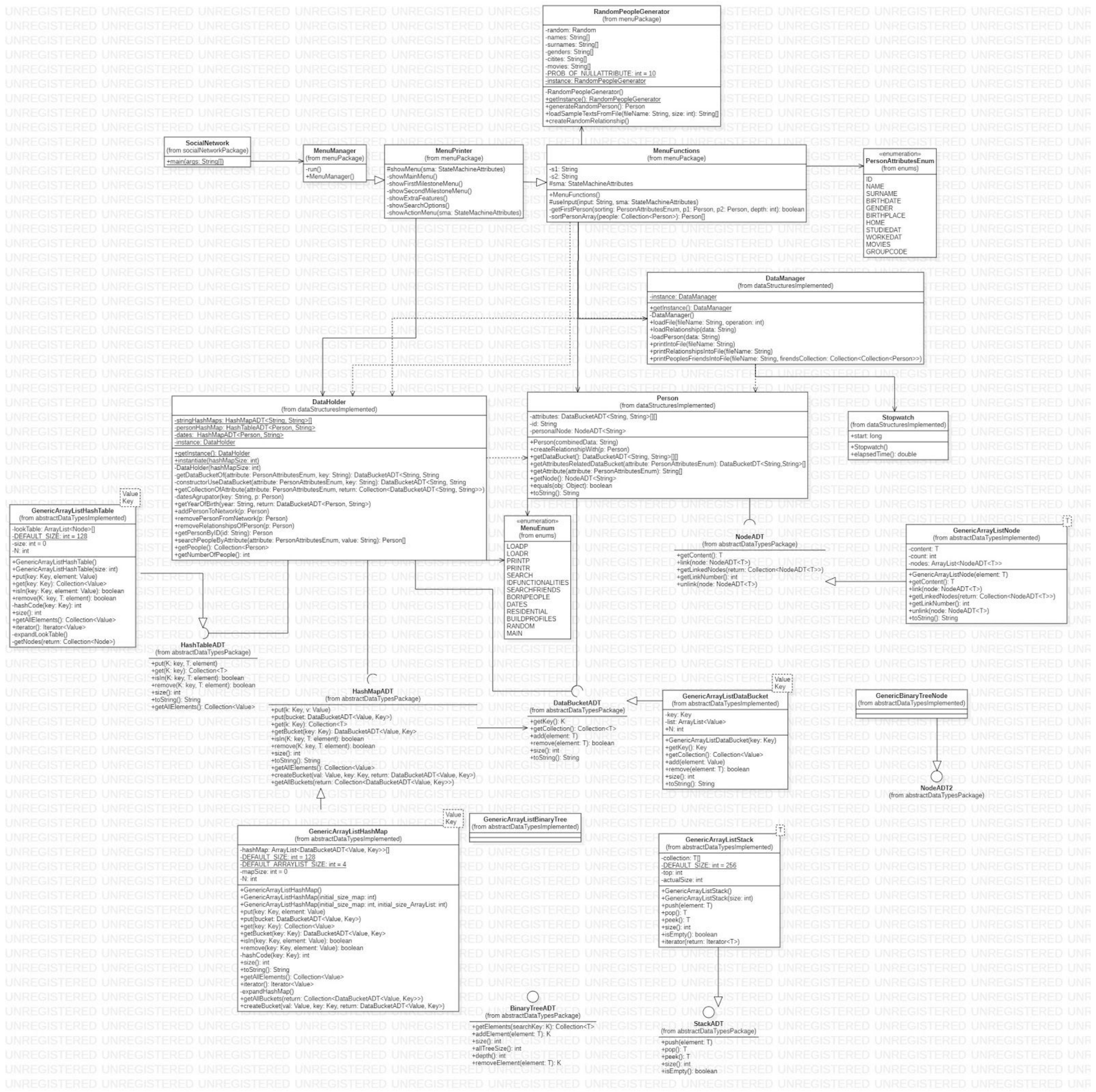
Version 1:

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


GenericArrayListBinaryTree
(from abstractDataTypesImplemented)
-less: GenericArrayListBinaryTree<T, K>
-more: GenericArrayListBinaryTree<T, K>
-key: K
-elements: ArrayList<T>
-N: int
+GenericArrayListBianryTree(element: T, key: K)
+getElements(searchKey: K): Collection<T>
+addElement(element: T): K
+size(): int
+allTreeSize(): int
+depth(): int
+removeElement(element: T): K
HashMapADT
(from adt)
BinaryTreeADT
(from adt)
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>)
SocialNetwork
(from SocialNetworkPage)
-dh: DataHolder
+main(args: String[])
-showMenu()
-showOptions()
DataHolder
(from SocialNetworkPage)
-stringHashMaps: HashMapADT<DataBlockADT<String,String>, String>[]
-personHashMap: HashMapADT<Person, String>
+DataHolder(hashMapSize: int)
-getDataBlockOf(attribute: int, key: String): DataBlockADT<String, String>
-constructorUseDataBlock(attribute: int, key: String): DataBlockADT
+addPersonToNetwork(p: Person)
+removePersonFromNetwork(p: Person)
+getPersonByID(id: String): person
+searchPeopleByAttribute(attribute: int, value: String): Person[]
+loadFile(fileName: String, operation: int)
+loadRelationship(data: String)
-loadPerson(data: String)
+printIntoFile(fileName: String)
GenericArrayListHashMap
(from abstractDataTypesImplemented)
-hashMap: ArrayList<T>[]
-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(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<T>
+iterator(): Iterator<T>
StackADT
(from adt)
Person
(from dataStructuresImplemented)
+NPARAMETERS: int = 11
+ID: int = -1
+NAME: int = 0
+SURNAME: int = 1
+BIRTHDATE: int = 2
+GENDER: int = 3
+BIRTHPALCE: int = 4
+HOME: int = 5
+STUDIEDAT: int = 6
+WORKEDAT: int = 7
+MOVIES: int = 8
+GROUPCODE: int
-attributes: DataBlockADT<String, String>[][]
-id: String
+personalNode: NodeADT<Srtring>
+Person(combinedData: String)
+getDataBlocks(): DataBlockADT<String, String>[][]
+getAttributesRelatedDataBlocks(attribute: int): DataBlockADT<String,String>[]
+getAttribute(attribute: int): String[]
+getNode(): NodeADT<String>
+equals(obj: Object): boolean
+toString(): String
NodeADT
(from adt)
DataBlockADT
(from adt)
ArrayListDataBlock
(from dataStructuresImplemented)
-key: K
-list: ArrayList<T>
+N: int
+ArrayListDataBlock(key: K)
+getKey(): K
+getCollection(): Collection<T>
+add(element: T)
+remove(element: T): boolean
+size(): int
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>)

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

Version 2:



RandomPeopleGenerator
(from menuPackage)
-random: Random
-names: String[]
-surnames: String[]
-genders: String[]
-citites: 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
Value
Key
GenericArrayListHashTable
(from abstractDataTypesImplemented)
-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>)
«enumeration»
MenuEnum
(from enums)
LOADP
LOADR
PRINTP
PRINTR
SEARCH
IDFUNCTIONALITIES
SEARCHFRIENDS
BORNPEOPLE
DATES
RESIDENTIAL
BUILDPROFILES
RANDOM
MAIN
NodeADT
(from abstractDataTypesPackage)
+getContent(): T
+link(node: NodeADT<T>)
+getLinkedNodes(return: Collection<NodeADT<T>>)
+getLinkNumber(): int
+unlink(node: NodeADT<T>)
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
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
Value
Key
GenericArrayListDataBucket
(from abstractDataTypesImplemented)
-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
GenericBinaryTreeNode
(from abstractDataTypesImplemented)
NodeADT2
(from abstractDataTypesPackage)
Value
Key
GenericArrayListHashMap
(from abstractDataTypesImplemented)
-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)
T
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>)
BinaryTreeADT
(from abstractDataTypesPackage)
+getElements(searchKey: K): Collection<T>
+addElement(element: T): K
+size(): int
+allTreeSize(): int
+depth(): int
+removeElement(element: T): K
StackADT
(from abstractDataTypesPackage)
+push(element: T)
+pop(): T
+peek(): T
+size(): int
+isEmpty(): boolean

6.3. 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 has a coverage of 80% of the project, the uncovered classes unused implemented ADTs.

Search in the project for more information.