REQUIREMENTS

1 March 2019

DOMAIN

The domain of this project are people who are subscribers to a music streaming service and their friends on this service. By nature of the data, we will limit the domain to the individuals who were subscribers on Deezer in late 2017 in Hungary, Croatia, and Romania, however the results can be extrapolated to include individuals of any music streaming service.

When dealing with returning the common music tastes in a friend group, we will further limit the friend group to be the group of friends the person has on the music streaming service (not necessarily who they have as friends in the real world).

Individuals using It Takes Two to Tango will benefit from the service by being able to see which music tastes they have in common with their friends, which friends share the most similar music tastes as them, and which friends (of friends) like a certain genre of music.

FUNCTIONAL REQUIREMENTS

Module

Genre

Genre.java will be an ADT that takes a String and returns a type for all 84 possible music genres.

Uses

None

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
Genre	*s: String*	Genre	StringNotGenreException
toString		String	
equals	*g: Genre*	Boolean	

Access Routine Semantics

Genre(*s*):

- Reads a String representing a genre and converts into an instance of Genre.
- A StringNotGenreException is raised if the String file does not correspond to any of the 84 supported genres.

toString():

- Returns a String representation of the Genre.

equals(*g*):

- Returns a boolean of true if the instance of Genre is equivalent to the genre of *g*; false otherwise.

Module

UserID

UserID.java will be an ADT that takes a String and returns a UserID.

Uses

None

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
UserID	*s: String*	UserID	
toString		String	
equals	*u: UserID*	Boolean	

Access Routine Semantics

UserID(*s*):

- Reads a String representing a user's id and converts into an instance of UserID.

toString():

- Returns a String representation of the UserID.

equals(*u*):

- Returns a boolean of true if the instance of UserID is equivalent to the genre of *u*; false otherwise.

Module

Reader

Reader.java will be a module with static methods that read CSV and JSON files. It will return arrays corresponding to the data it reads, and will be used to read information from the dataset.

Uses

Genre
UserID

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
readCSV	*s: String*	[UserID, [UserID]]	IOException
readJSON	*s: String*	[UserID, [Genre]]	IOException

Access Routine Semantics

readCSV(*s*):

- Reads a CSV file in the format of the dataset where there are 2 columns: one for node_1 and the other for node_2. In each row, the ID for node_1 and node_2 are given and the sementatic meaning is that the user whose ID is represented by node_1 is friends with the user whose ID is in node_2.
- readCSV(*s*) reads a file associated with the string *s*, and returns an array of arrays of length 2 which have the UserID and another array of UserID which represents their friends. It has the form of [[UserID1, [UserID1's friends]], [UserID2, [UserID2's friends]]...]
- An IOException is raised if the CSV file is not in the correct format as seen in the dataset or if the file could not be open.

readJSON(*s*):

- Reads a JSON file in the format of the dataset where each String ID representing a user is associated to a list of 84 possible genres.
- readCSV(*s*) reads a file associated with the string *s*, and returns an array of arrays of length 2 which have the UserID and another array of Genre which represents their music tastes.
- An IOException is raised if the JSON file is not in the correct format as seen in the dataset or if the file could not be open.

Module

Listener

Listener.java will be an Abstract Data Type that represents the user. It will have static variables that store user data, including their music tastes, and friend groups. Every user of It Takes Two to Tango will have their own Listener class, which will serve as their profile.

Uses

Genre
UserID

Syntax

Exported Constants
None

Exported Access Programs

Routine name	In	Out	Exceptions
Listener	*u: UserID*, *gs: [Genre]*	Listener	
getUseriD		UserID	
toString		String	
equals	*l: Listener*	Boolean	

Access Routine Semantics
Listener(*u, gs*):
- Reads a UserID and an array of Genre and returns a Listener.

getUserID():
- Returns the UserID of the listener

toString():
- Returns a String representation of the Listener.

equals(*l*):
- Returns a boolean of true if the instance of Listener is equivalent to the listener of *l*; false otherwise.

Module
Friends

An abstract data type which defines the friends of the user. There will be a final String variable that states that a friend's name, and the genres that friend listens to.

Uses
Genre
UserID
Listener

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
Friends	*l: Listener, us: [UserID]*	Friends	
addFriend	*u: UserID*		
removeFriend	*u: UserID*		FriendNotFoundException
toString		String	
isFriend	*u: UserID*	Boolean	
friendsWhoLike	*g: Genre*	Friends	
friendsWhoLike	*gs: [Genre]*	Friends	
friendMostInCommon		UserID	NoFriendInCommonException
commonPreference	*(All friends)*	[Genres]	
commonPreference	*us: [UserID]*	[Genres]	
sort		[UserID]	
searchClosest	*g: Genre*	UserID	
searchClosest	*gs: [Genre]*	UserID	
equals	*fs: Friends*	Boolean	

Access Routine Semantics

Friends(*l, us*):

- Given a Listener and an array of UserIDs associate these IDs as being friends of the Listener by returning an instance of Friends.

addFriend(*u*):

- Adds a friend to the Friends.

removeFriend(*u)*:

- Removes a friend.

toString():

- Returns a String representation of Friends.

isFriend(*u*):

- Returns true if the user associated to the UserID *u* is in Friends; false otherwise.

friendsWhoLike(*g*):

- Returns the Friends who like a certain genre *g*.

friendsWhoLike(*gs*):

- Returns the Friends who like the genres in *gs*.

friendsMostInCommon():

- Returns the UserID of the friend who shares the most similar music genres.

commonPreference():

- Returns the array of genres that are shared in common by all friends.

commonPreference(*us*):

- Returns the array of genres that are shared in common by only the friends in *us*.

sort():

- Returns an array of UserIDs of friends ranked in order of how similar their tastes are to the Listener

searchClosest(*g*):

- Returns the first friend (of friend… recursively search) that likes a certain genre *g*.

searchClosest(*gs)*:

- Returns the first friend (of friend… recursively search) that likes genres *gs*.

equals(*l*):

- Returns a boolean of true if the instance of Friends is equivalent to the friends of *fs*; false otherwise.

NON-FUNCTIONAL REQUIREMENTS

1. Execution Quality: Security
   When a user shares their data with Takes Two to Tango, they are trusting the information they enter is secured and safe. It is a requirement that their data is not accessible by any other users.

2. Evolution Quality: Reliability

The code should also be reliable. Reliability is a software characteristic which defines software that performs consistently well. The safeness of a program can be defined under reliability. It Takes Two to Tango should never compromise user information, by utilizing unsafe programming standards. For example, string interpolation should never be used directly with user data, as that could lead to injection attacks. User data should be processed to make sure it is safe first.

3. Execution Quality: Privacy

The music tastes of an individual should only be visible to people in their friend network.

4. Execution Quality: Effectiveness

This program should also be accurate to an acceptable degree. Some aspects of the code allows users to choose the level of accuracy they desire. This is done by letting them filter their friends by how many genres that have in common. The greater the number of genres they have in common, the more accurate and select the list of friends would be. The program should be able to adapt to the strictness of a user's search.

5. Evolution Quality: Testability

It is important that our code is structured in a way that supports sufficient testing. This will allow us to find faults in our software easier, while also allowing us to get instant feedback on any modifications we may make.

6. Execution Quality: Usability

As the end goal is to make a product that consumers will be able to use and enjoy, it is important that the usability of our software is high. This will be implemented through a GUI that is simple and easy to interact with. As music is a medium enjoyed by people of all ages, it's important that our software does not limit itself to being only usable by those with high technical skills.

DEVELOPMENT/MAINTENANCE

Takes Two to Tango needs to have certain characteristics to make sure that the implementation is practical. Firstly, the performance should scale well with a larger user database. This means being alert to which algorithms are expensive in terms of time complexity, and only using the more efficient ones. We will also be testing for accuracy along with efficiency.

This means we will have to use different levels of testing, starting with unit testing to ensure each part of the modules holds true for various quantities of data. Then we will need to use System level testing to verify all the methods in the module integrate properly. Lastly, we will also include Integration level testing which will be used to test the dynamic of all the

modules used together. This will ultimately involve testing all the features which are related to each other. Which is why by unit testing first will allow for easy debugging and maintenance.

In terms of the type of testing we will be doing, we will look at functional testing and destructive testing. With functional testing involves checking error messages, prompts, and other outputs. We would use this to check runtime and also and in the case an error occurs, the error handling routines. Finally, functional testing ultimately tests the accuracy of the output from different combinations of functions. Destructive testing will be used to test abnormal test cases. This includes testing for data overflow, and even memory usage. This will essentially be used to check extreme cases and see how the software reacts. All these methods work together to ensure a reliable and accurate software.