Carleton University - Department of Systems and Computer Engineering

SYSC-2004 - Object-Oriented Software Development - Fall 2015

Assignment 3

Part I – A quick spring cleaning first:

a) Now that we have Producer and Consumer as concrete classes of users, the User class itself only really serves as a way to supply them with some common methods and variables and shouldn't directly be instantiated. Make the User class abstract.

b) We now know that is better, whenever possible, to defer the choice of a data structure to when it is instantiated, and not when the variable is declared; i.e., it is better to do:

```
List<String> a = new ArrayList<>();
```

than:

```
ArrayList<String> a = new ArrayList<>();
```

This way, if we decide to switch from an ArrayList to, say, a LinkedList, we only need to make the change in one place, since both ArrayList and LinkedList implement the List interface. And if we do make such a change, there is no impact on client code that may rely on the type of this variable. So replace ArrayList variable declarations (including instance variable declarations) with List declarations everywhere. This shouldn't take long!

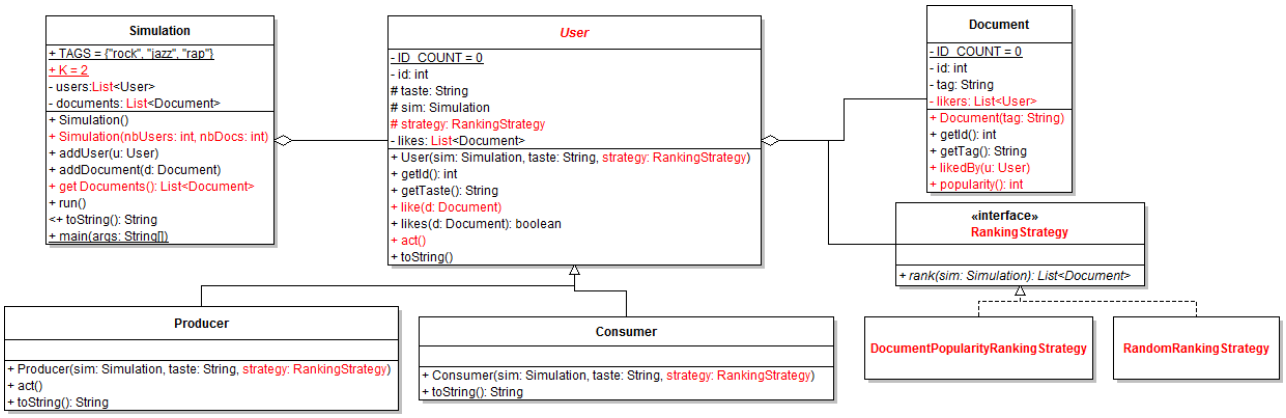
Part II – So far when performing a search, the simulation has returned all the documents. But usually, when users search for documents they really only want the *k* most relevant ones (where k is an arbitrary number). This is similar to how, when searching in Google, you only really pay attention to the first page worth of results, or only even the first few hits of that first page. In this assignment we want to provide two different strategies for choosing the top k documents.

Given that a user object doesn't know the tag of a document until it evaluates it, and it doesn't know the taste of other users or how they even decide to "like" a document, how can it retrieve the most k relevant documents? One strategy is to rank the documents according to their *popularity*, i.e. the number of "likes" they have. This might work well if most users have the same taste and they show it by "liking" those documents that matched their taste. It might not work so well if there is a great disparity in tastes among users.

Another strategy we will use is simply a "random" strategy, which will simply shuffle the list and return the top k. This strategy doesn't look very smart, but can act as a convenient baseline to compare the other strategies with.[1]

All strategies have in common the fact that they have to rank the documents contained in the simulation, and so we come up with a RankingStrategy interface that needs to be implemented by any concrete ranking strategy. This way we can "plug-in" the ranking strategy for a user by passing it as an object:

[1] To be perfectly honest with you, we thought of including a more interesting strategy as well (based on the users' "like" similarity), but it would have complicated things too much for the purpose of this assignment. If you are interested in investigating this, feel free to contact the instructor!



The rank() method defined in RankingStrategy takes the simulation as a parameter, so it can access the lists of documents and users (we also provide additional getters in class Simulation to make this possible). rank() then has to return the top k documents, where k is a constant defined in class Simulation.

Users will now have an instance variable referring to the strategy they're going to use, which will be passed to the constructor so it can be initialized. Now when a user wants to search for documents, it will ask the strategy to rank() the documents in the Simulation and return the top k. The search() method in Simulation is no longer needed and can now be safely retired.

To implement the popularity strategy described above, you will need to *sort* documents. While sorting algorithms aren't necessarily difficult to implement from scratch, the sort() method in ArrayList conveniently provides one for you (since JDK1.8!). To be able to do so, sort() requires a way to *compare* two documents. We have talked in class about Java's Comparable and Comparator interfaces, and their use in the sort() method. You will therefore have to implement a comparison method for popularity. We will leave you with a bit of freedom as to how you design this part.

To calculate the popularity of a document, i.e. the number of "likes" it has, you will also need to maintain a list of "likers" in class Document. This list needs to be updated every time a user "likes" a document.

The implementation of the random strategy is fairly straightforward. You might need to investigate the Java API to find a convenience method that can do most of the job for you!

One final point: in the constructor of Simulation that takes a certain number of users and documents as parameter (i.e. Simulation (int nbuser, int nbdoc)) , when populating the simulation by creating users at random, you need to choose the strategy for each user. For the purpose of this assignment, let's just systematically make producers use the random strategy and consumers use the similarity strategy. But I hope you can see how much flexibility our design provides in setting strategies.

Part III – No coding task is complete until you have made sure (as much as possible) that the code is working correctly! Create a unit test in which you populate a simulation with a few users and documents (not randomly! use the default constructor of Simulation), and make a user search for documents using the popularity strategy. Make use of assertions to check that the right documents are returned in the search and the other ones are left out. You can make use of BlueJ's test recording capability (which we demoed in class) or write it by hand (you saw examples of unit tests in the lab about Complex.plus()). Section 7.4 of the textbook also explains how to do this.

Now our burning question is: which strategy ultimately works best? We might address this issue in the next assignment!

Deliverables

From within BlueJ, create a JAR file containing your code (Project menu, click "Create JAR File"). When the "Create Jar File" dialog is displayed, make sure you select the Simulation class as the main class (this way you and the TA can execute your game directly from a command prompt), and that "Include Source" and "Include BlueJ Project" checkboxes are both checked! **Forgetting to include the source files means a mark of zero.**

Submit the resulting Jar file on cuLearn.

Due: Monday November 16th 2015 at 7pm. Late submissions will not be accepted.