Dawson Movie Recommendations : Adding Books!
Project Specifications
Phase III

Due: November 13th: Worth 25% of final project mark.

Phase III involves adapting what you've done in parts 1 and 2 to include book recommendations as well as movie recommendations. We will also use library objects to optimize the code written in part 1.

The same comments regarding expectations (e.g. documentation, exceptions, git, team work, etc) apply to this phase of the assignment.

Also, be sure to leave enough time to deal with various Git related/source repository issues and **remember to designate one teammate to take care of the final submission/merge request!**

Note: Sometime before the due date, your teacher will provide feedback on the code you've submitted for phase 2, but you shouldn't wait to start phase 3.

First, one teammate should create a new branch labelled phase3 which will serve as your teams staging branch for this phase. You can branch this from the latest code in the master branch

After that, all teammates should periodically pull phase3 (even if you aren't yet ready to push to it) to keep these protected branches up-to-date locally. Whenever you are about to start a new feature branch, first checkout phase3 and pull in updates of phase3 from origin.

Part 0: Fixing the mistakes from previous parts

Make sure that you allocate a proper amount of time to fix the issues you had during phase 1 or phase 2. You should start by reading over the comments given to you. In addition to this, make sure you have thoroughly tested your code to make sure that the fixes are there. And always keep your eye out for other bugs, even if your teacher---who is human too ☺ -- didn't notice them.

Part 1: Updating PersonalizedRecommender to use HashMap

In phase 2, we did some "coding gymnastics" to make allow user ids to be the index of our various arrays. This has some serious downsides, specifically that the user ids now have to be positive (or zero) integers only. Also, imagine that the system had user ids starting in the millions (for example: if the user id was equal to your Dawson student id!), then this would be a big waste of space.

In this phase, we will use two *HashMap* to store the int[] mostSimilarUsers and the Rating[][] that you created in the last phase.

Recall that to use a *HashMap*, you must provide *two* types: the type of the "index" or "key" and the type of the value itself.

HashMap<Integer, Integer> mostSimilarUsers = new HashMap<Integer, Integer>();
Creates a map from Integer to Integer.

You can then insert things into the HashMap by using a method *put:*

mostSimilarUsers.put(3, 10); // inserts an element with a key of 3 and value of 10
mostSimilarUsers.get(3); // gets the value that 3 maps to
mostSimilarUsers.containsKey(3); // returns a boolean of true or false based on if the map contains 3 as a key or not.

- Change the int[] mostSimilarUsers to be a HashMap<Integer, Integer> instead.
- Change the Rating[][] to be a HashMap<Integer, ArrayList<Rating>> instead.
    - Hint: This is a little trickier, but you can write code such as the following:
      map.put(3, new ArrayList<Rating>()); // would mean user with id of 3 has not rated any moves yet.
      ArrayList<Rating> userThree = map.get(3); // gets the list for user3. Remember that due to aliasing that changing userThree's list will also change the list within the map!
      userThree.add(*some rating)*; // this will successfully change the list that 3 maps to due to aliasing.

Since both arrays you changed are private fields, it should not be necessary to make any other changes to the code in other classes. However, it is possible that you introduced a bug in the process of making this change (if you made an error). So you should retest your code from other classes. Hint: If you have written a good set of automated tests, you can run them now and have some confidence that things are still working well!

Optional things you can do:

- For performance reasons, it is best to create the HashMap<Integer, ArrayList<Rating>> at the beginning of the constructor before calculating the user similarities. The reason is that it is useful to use this HashMap to optimize calculating the similarities between the users.
- You can also update your userSimilarities array to be a HashMap<HashMap<Integer, Double>> instead of a double[][].
- If you've done both of the above, you can actually convert your userids back to a String (and then your HashMaps to be HashMap<String, ArrayList<Rating>> instead of HashMap<Integer, ArrayList<Rating>> This has the advantage that we could incorporate more easily a system that has a String as a userid instead (e.g. an email address as the user id)

Part 2: More interfaces and generic types

In this next section, we will add some infrastructure to the code by adding several classes/interfaces. We will be writing all the classes/interfaces to make our code as in the UML diagram that follows the specification. The idea is so we can write a book recommendation engine with as little work as possible. To do this, we'll need to rearrange a few things first though. Most of the below should be cutting/pasting existing code that you wrote in the previous parts, but there is some new code to write.

- Within the package recommendation.interfaces, define a *generic* recommender interface *IRecommender*. Because genre based recommendations only make sense in the context of movies (not all sorts of things will have a genre), this interface should have one method: recommend(int userId, int n). Because you want this to be able to work on all sorts of items---Movie, Book, etc, you should define the interface to be generic by including <T> in the header.

public interface IRecommender<T>

This means that you can use T throughout the interface. The recommend method, should return a T[]. When you create an object, just like with ArrayList, you'll provide the type.

- Within the package recommendation.movies, define an interface *IMovieRecommender* that *extends* the IRecommender interface. *IMovieRecommender* is *not* generic, meaning that when you extend it, you'll need to write

  public interface IMovieRecommender extends IRecommender<Movie>

  This means that IMovieRecommender is extending the original interface, but the only kind of thing you can use it for is a Movie.

  This interface should have one additional method in it: the recommend that takes an extra parameter for genres. To implement this interface will require you to provide 2 methods, since the 1 method is automatically inherited from IRecommender.
- Within the package recommendation.interfaces, define a generic class *PopularRecommender<T>* which will work on all possible types of items. This class should implement IRecommender and should provide the one method, based on what you did last phase.

  Hint: The class header will look

  public class PopularRecommender<T> implements IRecommender<T>
- Within the package recommendation.interfaces, define a generic class *PersonalizedRecommender<T>* which will work similar to PopularRecommender<T> except based on the most popular items.
- Within the package recommendation.movies, define a class *PopularMovieRecommender* which extends *PopularRecommender<T>* and implements *IMovieRecommender.* This class is where you should put the recommend method based on genres that you wrote in part 2. The constructor for a *PopularMovieRecommender* should take as input a Movie[] and a Rating[] as before, but it should use super() to call the constructor for the PopularRecommender

  Hint: To avoid code duplication, you should call the recommend method that is part of the *PopularRecommender<T>* method using the *super* keyword. (Remember that you can call a method in the base class by writing super.methodName(input)).

  If you do this, then you won't need to store the Rating[] within the PopularMovieRecommender at all. You will only need to store the Movie[], and that is just for the purpose of checking the genres.

  However, if you call super.recommend(userId, n), the resulting array will include Movies that do not have a matching genre. For this reason, you will need to call it with a number bigger than n. In fact, there is no number that *guarantees* that you can get n movies with a matching genre. So you will need to think through how best to write this logic.

  One idea is to call it with a number bigger than n (for example 10 * n) and then count how many of those movies have a matching genre. If there still aren't n movies with that genre, then call it with 100 * n movies. Note that it is possible that no matter how large *n* is, you will never be able to get *n* recommendations with that genre (if, for example there are not *n* movies with that genre). So you need to make sure your code works properly here.

  Another idea is to write a *protected* method in the *PopularRecommender* class that simply returns *all* of the items matching (in sorted order based on the rating of the similar

user). Then you can call this method and do the extra filtering based on genre in your PopularMovieRecommender class.
Hint: The header for this class will look like
public class PopularMovieRecommender extends PopularRecommender<Movie> implements MovieRecommender

- Within the package recommendation.interfaces, define a class *PersonalizedMovieRecommender* which does the same idea as *PopularMovieRecommender* except it should work based on the personalized algorithm from phase 2. PersonalizedMovieRecommender should extend PersonalizedRecommender<Movie> and implement IMovieRecommender

  You should also delete your old recommenders from phase 1 and 2.

Part 3: Providing Book Recommendations: Book class

For books, we will be using a data set called GoodBooks. There are two files to work with: A sample file which contains a subset of the actual files as well as a more detailed one. You should primarily work with the sample files (posted on Lea), but are encouraged to try getting your recommender to work with the full set as well.

The sample files (identical to what is posted on Lea) can be found at https://github.com/zygmuntz/goodbooks-10k/tree/master/samples

(The bigger files can be found at https://github.com/zygmuntz/goodbooks-10k If you choose to use the bigger files, it is important *not* to add them to your gitlab repository. Otherwise, they are so large that every time anyone updates them, a git pull will take a long time!)

Add the two sample files to your repository under datafiles/books/. You'll notice the format is very similar to what we saw with movies, but the number of columns is a bit different.

Now we will add support to support book recommendations as well. Because of the infrastructure we set up, there actually wont' be as much code here as we might expect!

1. First define a class Book in a package recommendation.book. A Book, like a Movie, implements Item. However, books have slightly different things. In addition to an id and name, they have a String[] authors and a String isbn.
2. Add getters for these two fields.
3. Add a constructor that takes as input a String that looks like a single line of the file books.csv and splits it accordingly to put all the right fields in the right places.
4. Add a **BookTest** class (in the same package as your other tests) to verify that everything works correctly with this class

Part 4: File Input and Output: Books

Define a class GoodReads*FileReader* inside a package recommendation.fileio . This class should have two statics method in it:

- loadBooks() : This method should take as input a String representing a path to a file and should return a Book[] of all the Books contained in that file. You should use the Book constructor written in the previous part.
- loadRatings() : This method should take as input a String representing a path to a file and should return a Rating[] of all the Ratings contained in the file. The

format of the rating file for Books is nearly identical to the format of the rating file for Movies, but there is no longer a timestamp in it. Because of this, you may have to make a couple minor tweaks to your Rating constructor so that it can work on each kind of file.

Part 5: Updating your main application

Add logic to your main application so that it can work for Book recommendations as well. Before you ask the user to enter the two files, ask whether they want a Book or a Movie recommendation. You can then use this choice to determine which file reader you will use as well as what questions to ask (since book recommenders don't have genres).

Submission:

To submit your team's code, open a merge request of your repository's staging branch against master. When creating the merge request:

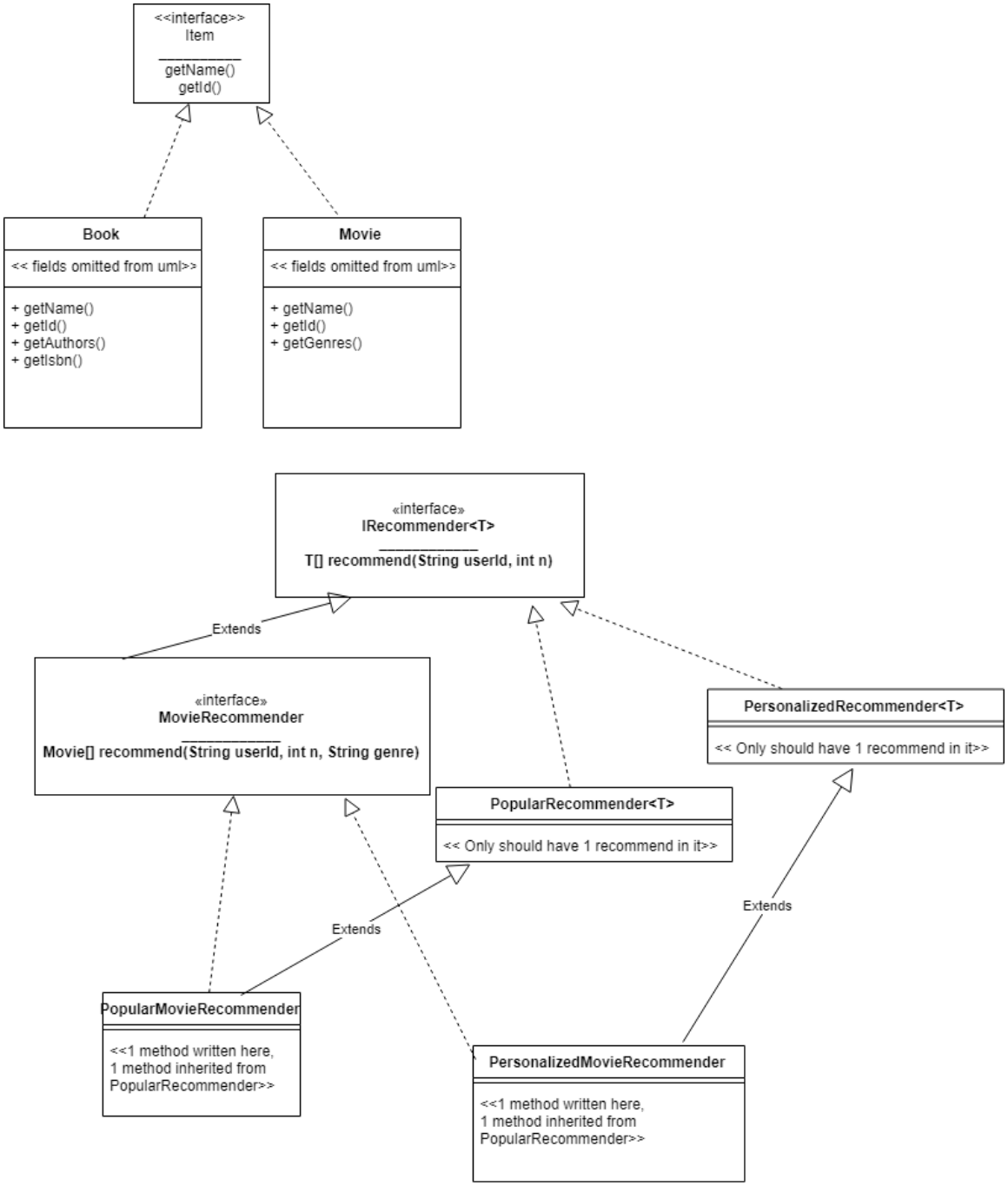
Set the title of the merge request to Complete

Set @dpomerantz as the Assignee

@-mention your teammates in the description
Add a label called grade (you should be able to create a new label if it doesn't exist))

Each student must individually submit their peer evaluations via Lea. Note that the peer evaluations will not be the only thing considered in terms of your team contribution. They will mainly be used to help detect as early as possible if there is any problem with the team dynamic.

In the evaluation, you should highlight anything---good or bad---that is exceptional. For example, if there was a specific thing that a teammate did that impressed you, you can point it out there. You should also rate yourself as part of this process.


<<interface>>
Item
getName()
getId()
Book
<< fields omitted from uml>>
+ getName()
+ getId()
+ getAuthors()
+ getIsbn()
Movie
<< fields omitted from uml>>
+ getName()
+ getId()
+ getGenres()
«interface»
IRecommender<T>
T[] recommend(String userId, int n)
Extends
«interface»
MovieRecommender
Movie[] recommend(String userId, int n, String genre)
PersonalizedRecommender<T>
<< Only should have 1 recommend in it>>
PopularRecommender<T>
<< Only should have 1 recommend in it>>
Extends
Extends
PopularMovieRecommender
<<1 method written here,
1 method inherited from
PopularRecommender>>
PersonalizedMovieRecommender
<<1 method written here,
1 method inherited from
PopularRecommender>>