CSCI 4448: Project Part 3

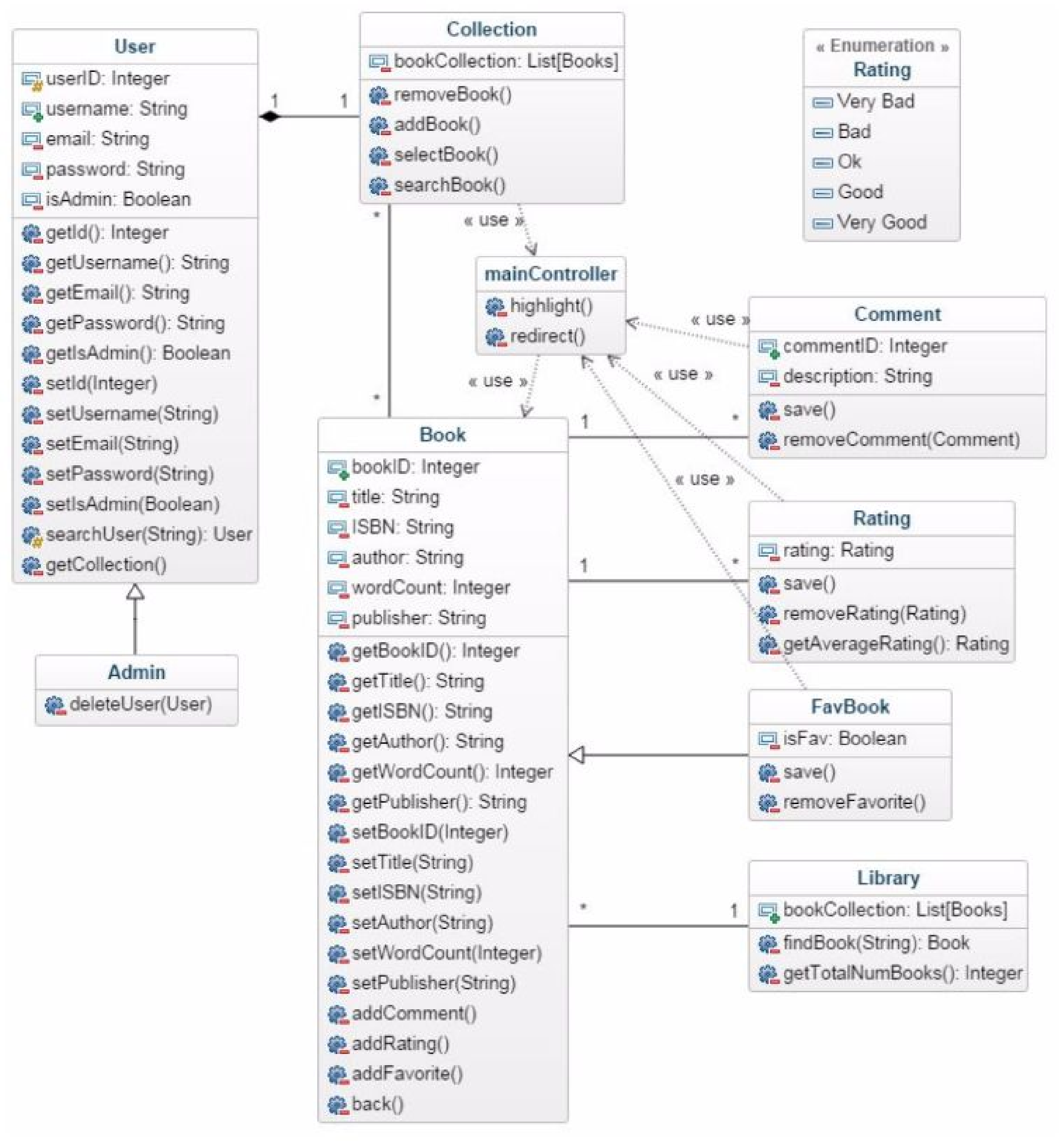
Team

- Matt Oakley
- Daniel Thompson
- David Ingraham
- Cory Cranford

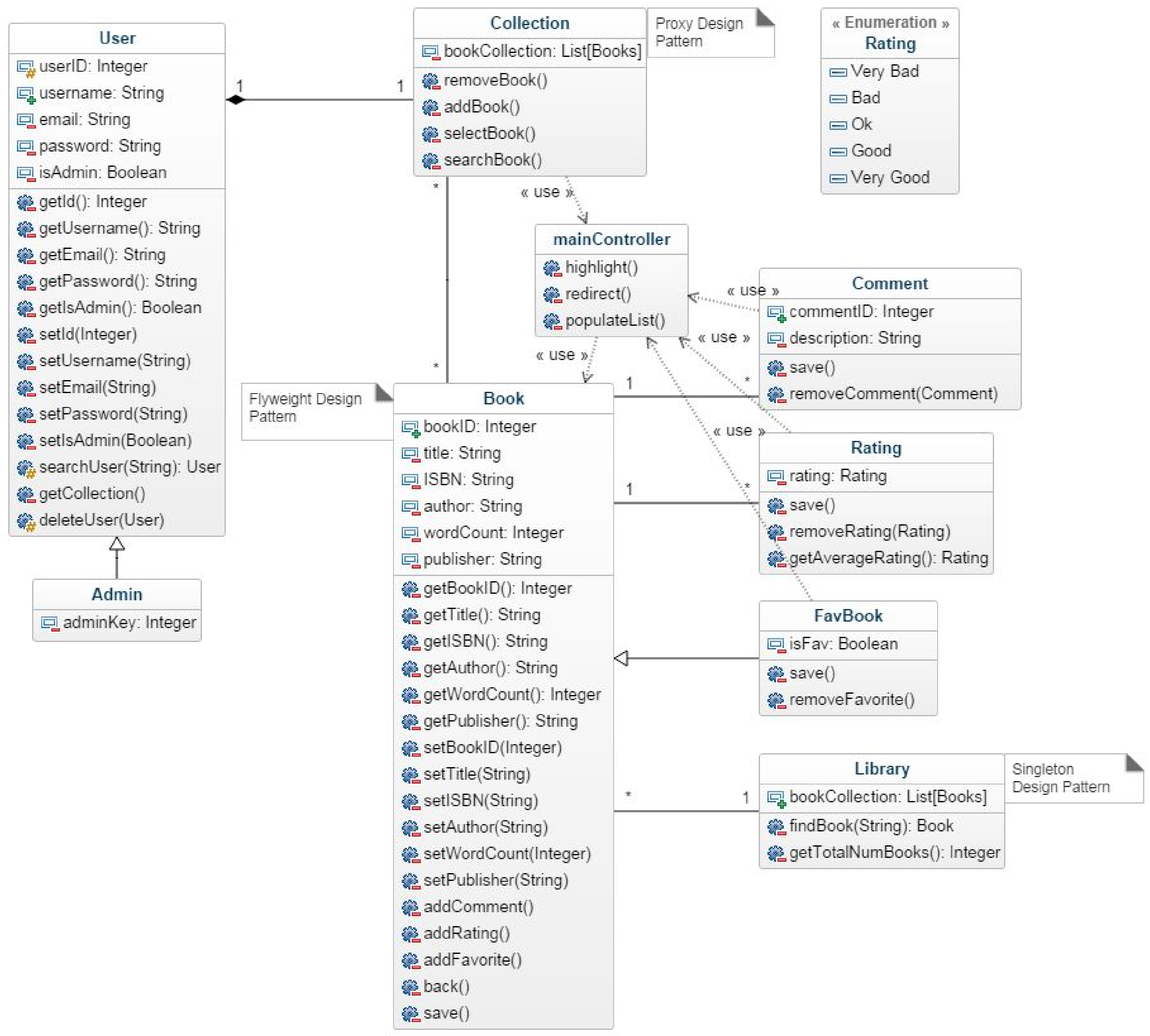
Title: Virtual Library

Part 2 Class Diagram (Original):

Class Diagram:



Updated Class Diagram:



Changes:

1) Removed deleteUser() method from Admin class as per Project Part 2 grading feedback
2) Added adminKey attribute to Admin class in order to specify different admins
3) Added save() method to Book class
4) Added populateList() method to mainController class
5) Added “Proxy Design Pattern” comment to Collection class
6) Added “Singleton Design Pattern” comment to Library class
7) Added “Flyweight Design Pattern” comment to Book class

Design Pattern Reasoning:
Proxy (Remote): In thinking about how we could add more functionality to our project, we entertained the idea of locally caching the user's collection of books if they were to lose internet connection. As our project currently stands, the user's collection is associated with a MySQL database which would be unable to complete transactions if the user loses connectivity. Therefore, we could implement the Remote Proxy design pattern in order to locally save the user's collection on their device. This would allow the user to continue to use the application and view their library if they were to lose connection.

Singleton: While all users have their own "Collection" as denoted by the Collection class, there is also a library class which contains *all* of the books in our database (while we haven't learned the technology yet, we believe this is going to be constructed via Hibernate). Since there should only be one single, overarching library we've decided to utilize the Singleton design pattern for this class.

Flyweight: We anticipate that our overarching Library is going to contain a very large number of Book objects. Since these Book objects are all fairly similar and only differ in details such as title, author, number of pages, etc. we thought we would implement the Flyweight design pattern in order to prevent the high storage cost of storing each individual Book object.

Strategy*: If our Library management system was to be extended to a general Media management system which could store things such as books, MP3 tracks, videogames, etc. then we could implement the Strategy design pattern. For instance, the standard algorithm to search/view/exit would be the same between these different media. However, if we added functionality to interact with these media then the algorithm would be slightly different for each different piece of media. This interaction would occur at the same place in the algorithm but would differ depending on the media. For example, books would have a readBook() method, MP3s would have a playTrack() method, videogames would have a playGame() method, etc.

*Denotes an idea, not a design pattern we are going to implement.

Summary:
For our refactoring we began by implementing the changes that were specified in the feedback from our Part 2 submission. This included both feedback from the teacher as well as feedback within the group. For example, some group members had methods in their sequence diagrams that didn't make it into the class diagram. We did a better job at synchronizing all the diagrams this time around. After this, we brainstormed about what kind of design patterns our project followed. We ended up agreeing on a couple possible design patterns we could add to augment our project (Flyweight, Singleton, Proxy) and one that is out of scope due to the limited amount of time that we have for the project (Strategy). We'd like to keep the idea of incorporating the strategy design pattern for future considerations if the project were to grow.

We also looked at what else we could “clean-up” as per the information we received in class from the refactoring lectures. In doing so we found that our book class was verging on becoming a “Blob anti-pattern”. On closer inspection most of the massive amount of methods are just getters, and setters (which should not be an issue), and all of the methods follow the rules of encapsulation; it just happens to be that the book class have the largest amount of methods so it looked out of place by comparison to the other classes. This is probably the main issue we need to keep in mind for our app. Additionally, once we really start getting into coding the project we should be very careful of the “Spaghetti” anti-pattern since a lot of the coding will be completed individually. We are well aware that without everyone on the same page in regards to the design, the code could turn into a jumbled mess. Being mindful of this will be very beneficial to the other group members and anyone wanting to view/understand the system.