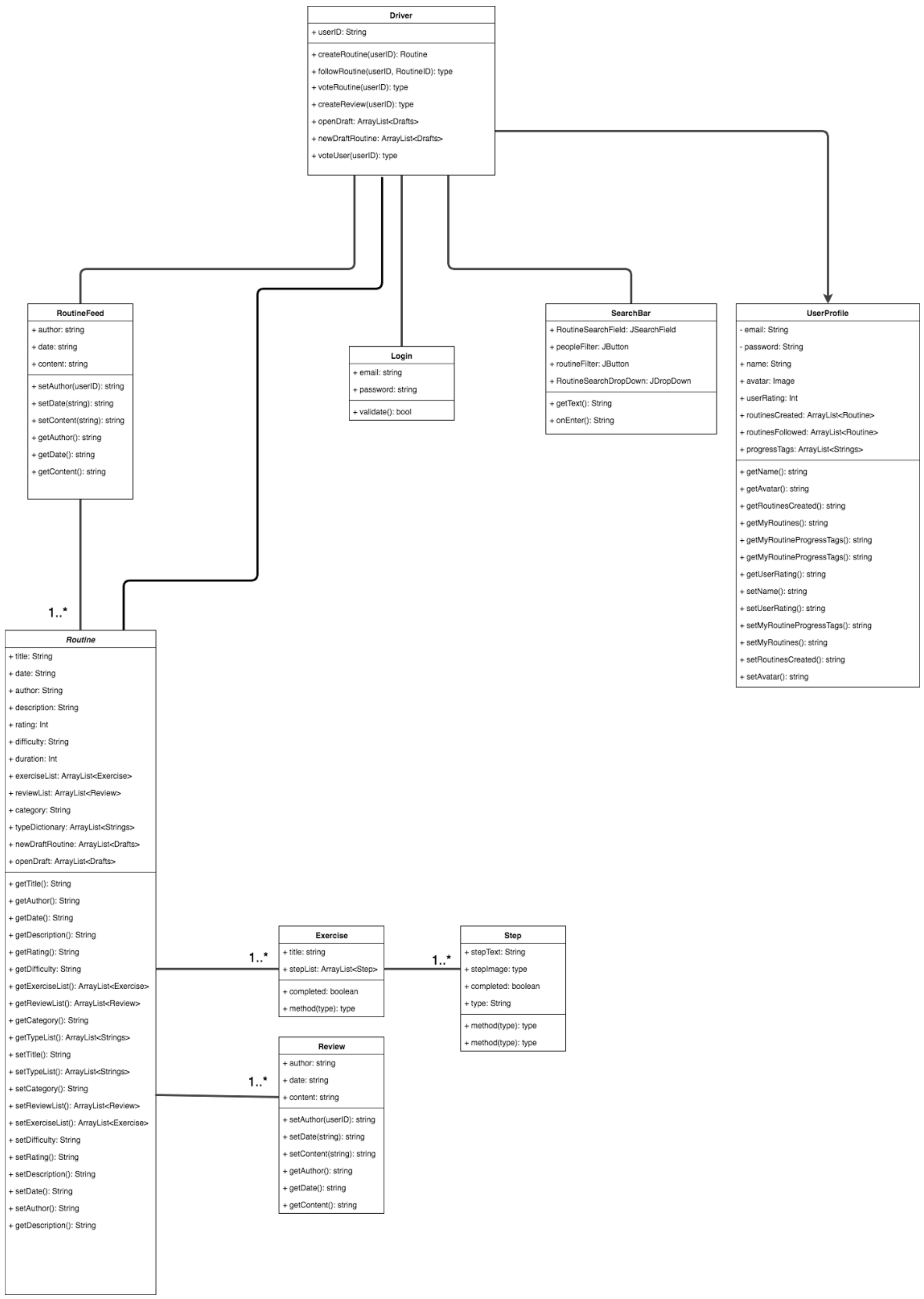
1. **Completed features**

UR-02	A registered user must be logged in before accessing the home page.	Authentication	All Users	Critical
UR-03	Upon successful login a user must be directed to his or her personalized RoutineFeed.	Navigation	All Users	Critical
UR-04	A viewer must be able to filter their RoutineFeed based on pre-defined categories.	Sorting Feature	Viewers	High
UR-05	A viewer must be able to search for other users.	Sorting Feature	Viewers	Medium
UR-06	A viewer must be able to search for routines based on keywords.	Sorting Feature	Viewers	High
UR-07	A creator must be able to draft their routine.	Creation	Creators	High
UR-08	A creator must be able to submit their routine to the RoutineFeed.	Submission/Upload	Creators	High
UR-09	A viewer must be able to vote on a routine after selecting the routine.	Feedback	Viewers	Medium

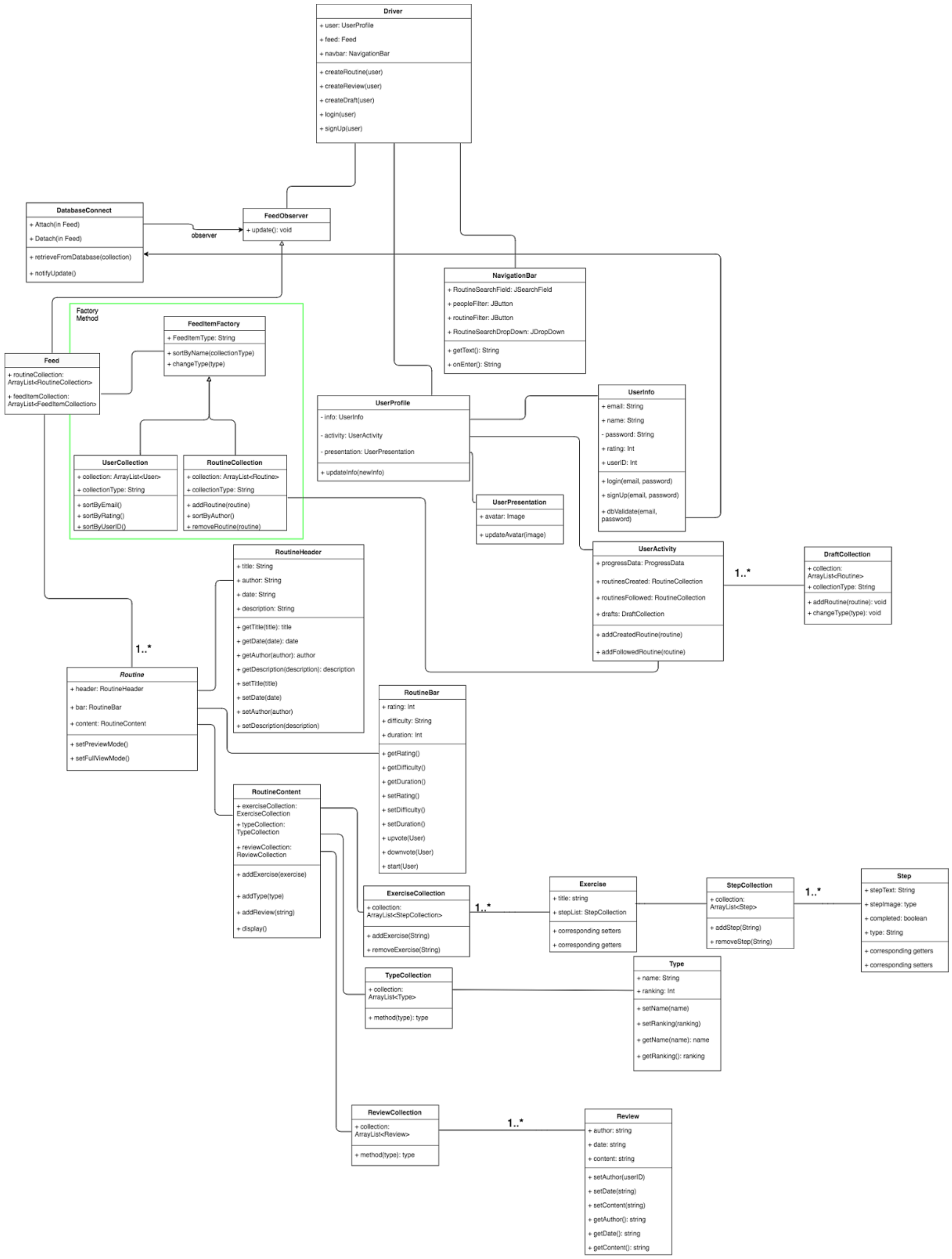
2. **What wasn't implemented**

UR-01	A non-registered user must create an account before logging into the service.	Authentication	All Users	Critical
UR-10	A viewer must be able to follow a routine after selecting the routine.	Feedback	Viewers	Medium
UR-11	A viewer must be able to provide a review (post a comment) for a routine.	Feedback	Viewers	Low

3. Class Diagram (Part 2)


Driver
+ userID: String
+ createRoutine(userID): Routine
+ followRoutine(userID, RoutineID): type
+ voteRoutine(userID): type
+ createReview(userID): type
+ openDraft: ArrayList<Drafts>
+ newDraftRoutine: ArrayList<Drafts>
+ voteUser(userID): type
RoutineFeed
+ author: string
+ date: string
+ content: string
+ setAuthor(userID): string
+ setDate(string): string
+ setContent(string): string
+ getAuthor(): string
+ getDate(): string
+ getContent(): string
Login
+ email: string
+ password: string
+ validate(): bool
SearchBar
+ RoutineSearchField: JSearchField
+ peopleFilter: JButton
+ routineFilter: JButton
+ RoutineSearchDropDown: JDropDown
+ getText(): String
+ onEnter(): String
UserProfile
- email: String
- password: String
+ name: String
+ avatar: Image
+ userRating: Int
+ routinesCreated: ArrayList<Routine>
+ routinesFollowed: ArrayList<Routine>
+ progressTags: ArrayList<Strings>
+ getName(): string
+ getAvatar(): string
+ getRoutinesCreated(): string
+ getMyRoutines(): string
+ getMyRoutineProgressTags(): string
+ getMyRoutineProgressTags(): string
+ getUserRating(): string
+ setName(): string
+ setUserRating(): string
+ setMyRoutineProgressTags(): string
+ setMyRoutines(): string
+ setRoutinesCreated(): string
+ setAvatar(): string
1..*
Routine
+ title: String
+ date: String
+ author: String
+ description: String
+ rating: Int
+ difficulty: String
+ duration: Int
+ exerciseList: ArrayList<Exercise>
+ reviewList: ArrayList<Review>
+ category: String
+ typeDictionary: ArrayList<Strings>
+ newDraftRoutine: ArrayList<Drafts>
+ openDraft: ArrayList<Drafts>
+ getTitle(): String
+ getAuthor(): String
+ getDate(): String
+ getDescription(): String
+ getRating(): String
+ getDifficulty: String
+ getExerciseList(): ArrayList<Exercise>
+ getReviewList(): ArrayList<Review>
+ getCategory(): String
+ getTypeList(): ArrayList<Strings>
+ setTitle(): String
+ setTypeList(): ArrayList<Strings>
+ setCategory(): String
+ setReviewList(): ArrayList<Review>
+ setExerciseList(): ArrayList<Exercise>
+ setDifficulty: String
+ setRating(): String
+ setDescription(): String
+ setDate(): String
+ setAuthor(): String
+ getDescription(): String
1..*
Exercise
+ title: string
+ stepList: ArrayList<Step>
+ completed: boolean
+ method(type): type
1..*
Step
+ stepText: String
+ stepImage: type
+ completed: boolean
+ type: String
+ method(type): type
+ method(type): type
1..*
Review
+ author: string
+ date: string
+ content: string
+ setAuthor(userID): string
+ setDate(string): string
+ setContent(string): string
+ getAuthor(): string
+ getDate(): string
+ getContent(): string

Class Diagram (Final)



Our group changed a good amount between our original class diagram and the final state of our class diagram. We quickly realized that we needed to add extra classes between each of our routines, exercises, and steps. The classes that we had planned contained a huge amount of information with several getter and setter methods. This was fixed by breaking related attributes within larger classes into their own smaller classes. For instance, the routine class was broken into a routine header with metadata about the routine, a content class with all information on what makes up the routine, and a bar class to hold actions. This made all attributes and methods of RoutineMe much more manageable.

Doing class design before coding this project helped our group understand how the app would function. We had the initial idea for RoutineMe but we needed to look at how we would lay out classes to understand the sequence of events that were needed to do one thing. For example, being able to add a step to a routine seems simple enough, however several classes and checks were required to make that happen. The class diagram allowed our team to understand the need for the collection classes that would provide us with multiple exercises and each exercise would be made up of several steps. Once we began writing the code, we found it easy to implement these steps and have this piece of functionality working properly.

4. Design Patterns

Our group had planned on using some of the design patterns that were covered in class. The refactored class diagram designed by our group in part 3 of this project outlined two main design patterns to be used in RoutineMe. When a user would log into the app, they would immediately be directed to their routinefeed. This is the location where a user would want to see the most recent, popular, and relevant routines that would apply to them. To keep this feed at its most up to date state, our group wanted to implement an observer. This would be a feed observer class which would connect directly to the database which holds all of the information for each routine on RoutineMe. Whenever a new routine is created and added to the database, the observer would notify the routine feed and immediately update it with the newest instance of the routine. This design pattern would have allowed RoutineMe to dynamically update with the most recent user submitted content. We were unfortunately unable to implement this class as our use case requirements did not require a database and we were unable to implement the hibernate framework.

The other design pattern that our group had planned to use in RoutineMe was the factory pattern. This pattern would be implemented with new routines in the feed which would easily add new users and routines into the routine feed. This addition would be very useful for a popular app with many users. As more routines are created and submitted to the routine feed, the factory class would handle the heavy work of placing routines into the feed. As RoutineMe grows with more users, it would handle adding these users into our user collection. At the current scale of RoutineMe this feature is not needed. However, if more daily users were on the app, we would want to implement this design pattern.

5. Lessons Learned

One of the major lessons that our group has learned from this process was the importance of planning realistically. This project began with our group thinking of several features that RoutineMe could have. We were very excited about the idea of programming this app that we didn't consider what we would be able to implement within a semester. After the steps of planning, designing, and refactoring, we realized that we had a lot of plans for RoutineMe and had to prioritize to get the main functionality working perfectly.

Our group discovered the importance of all of the steps leading up to programming in object-oriented development. This project required a lot of classes with specific methods in order to achieve the functionality that our group desired. The process of planning out use cases and requirements taught us what steps we wanted our users to go through to achieve a small task with RoutineMe. We had the opportunity to discuss what we wanted each piece of the app to look like and which buttons would perform which actions.

The design phase allowed our group plan out the back-end of the app and how the classes would interact behind the scene. This was the most challenging phase for our group because RoutineMe would hold a lot of content which required several getter and setter methods along with our other methods. Mapping out these classes seemed like an impossible task but in the end, our group was able to plan what would be needed by all of the classes of our project. Refactoring was another useful step because it allowed us to take our original plan for RoutineMe and clean it up using some of the design patterns we had learned about. Overall, it was important learning about the design process and has taught all of us lessons about time management and the reality of being able to produce a product with a short timeline.