Project Part 4

1.

Choreganizr is an application to track and manage household chores. Ultimately, we did not implement all of the features we planned from part 2 of the project.

Implemented Features (from part 1):

- Users can sign up
- Users can log in
- Users can join group with other users to create a “house”
- Users can create new chores
- Users can assign chores to other members of house
- Users can mark chores as complete
- Users select from premade list of chores that apply to the house

Requirements (from part 2):

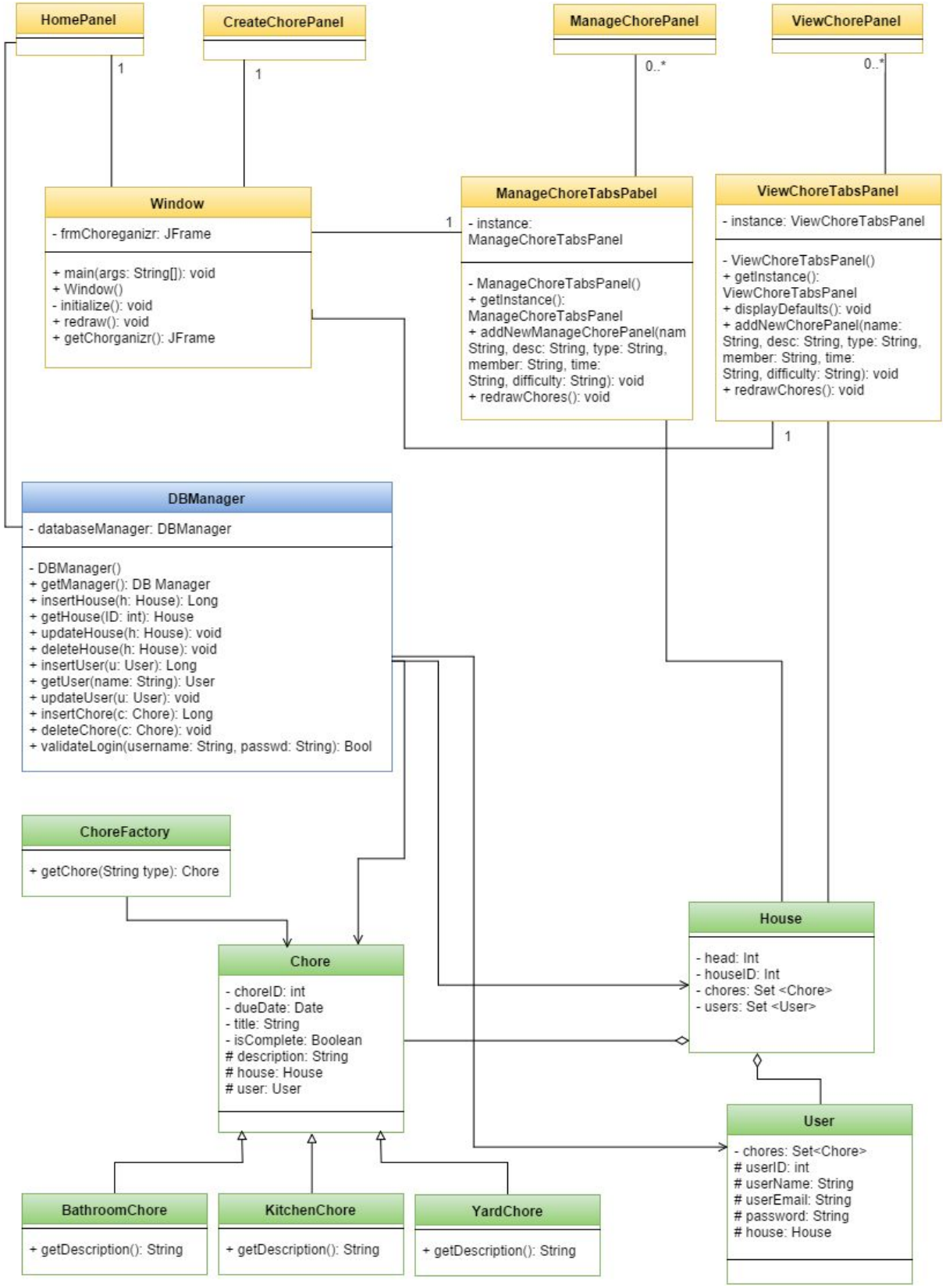
Business Requirements		
Id	**Requirement**	**Completed?**
BR-01	Passwords must be 8 alphanumeric characters.	NO

User Requirements		
Id	**Requirement**	**Completed?**
USR-01	Users can create account with chosen email, username, and password.	ALMOST
USR-02	Users can recover passwords.	NO
USR-03	Users can log in.	YES
USR-04	Users can create password-protected Households as the Head.	ALMOST
USR-05	Head users can add users as Housemates to Household.	ALMOST
USR-06	Housemates can view Chores and who they are assigned to.	YES
USR-07	Head users can select Chores from a premade list.	YES
USR-08	Housemates can create new Chores.	YES
USR-09	Head users can manually delegate Chores to Housemates.	YES
USR-10	Housemates can mark Chores they’ve been assigned as Completed.	YES

Functional Requirements		
Id	**Requirement**	**Completed?**
RF-01	User passwords are hashed before being stored in the database.	NO
RF-02	When a user attempts to create an account, the password is checked to ensure it meets minimum requirements.	NO
RF-03	A user should only see chores that apply to their household.	YES
RF-04	When a user marks a chore as completed or comments on a chore it should update on all other user's accounts.	YES
RF-05	New users should immediately be presented with an option to sign up.	NO

Non-Functional Requirements		
Id	**Requirement**	**Completed?**
NFR-01	After the user enters login information, the time before they are brought to the home page should be no more than .5 seconds	YES
NFR-02	Searching for a user should not take more than .5 seconds.	YES
NFR-03	The system should perform the same on mac, pc, and linux.	YES

Final class diagram:



We noticed several benefits to doing a complete design of the application before beginning to write any code. Creating a design helped us understand the scope of the what we were trying to accomplish. When first planning to complete a project, it's easy to overestimate how much a team can do in the span of one semester. Initially, we wanted to implement many more features and have a much more complicated GUI. Designing the project during part 2 made it clear that we needed to scale back some of the requirements in order to be able to deliver a working product. Designing helped us break down nebulous ideas into a concrete and realistic plan.

Another benefit to doing the design is that it helped us write reusable and understandable code. Although our initial design is quite different from the final product, approaching the project with object-oriented principles in mind and adhering to best practices allowed us to create a cleaner and better product than if we had started writing code immediately.

Since this is a team project, designing also helped us coordinate. Talking through and making diagrams for use cases and classes allowed us to understand each other's ideas about the application. We had to communicate and advocate for our own ideas in order to incorporate all the different views into one cohesive design. Without this step, we could've started coding before realizing that our visions for the product clashed.

Lastly, completing part 2 of the project helped us visualize and recognize the shortcomings of our design. The design was far from perfect (which is partly the reason we deviated from it in the implementation), but having it laid out in front of us was helpful for understanding what we needed to change to create a final, functional product.

2.

We used several design patterns in the final implementation of our prototype.

We created a class called DBManager to handle all interactions with the database. It was implemented using the Singleton design pattern. This was an appropriate time to use a singleton because several classes needed to be able to access the DBManager, and in order to avoid potential issues with concurrency (we didn't want multiple objects accessing the same database), we needed to have only one instance that would take care of all necessary operations.

To create Chore objects, we used a ChoreFactory class that implemented the Factory design pattern. We wanted to have several different types of chores, and each Chore's type needed to be determined at run-time from user input. The factory allows us to treat all Chores equally and have a variety of Chore subclasses. Additionally, the factory design pattern allows us to easily add new subclasses of Chores in the future. We adhere to the Open-Close Principle in this way.

For the whole application, we used the Model-View-Controller architectural style. We separated responsibility into sets of Model, View, and Controller classes. The View classes deal with the GUI (and most end with "Panel" for clarity). The View classes use the Java Swing framework to create the GUI. These View classes do not interact with themselves, which helps to ensure data integrity.

The Model class, DBManager, updates the database as the Controller objects change. It accomplishes this through the use of the Hibernate ORM. It also alerts the View and Controller classes when a change in the data model needs to be reflected in the application. This ensures that what the user sees actively reflects objects in the backend.

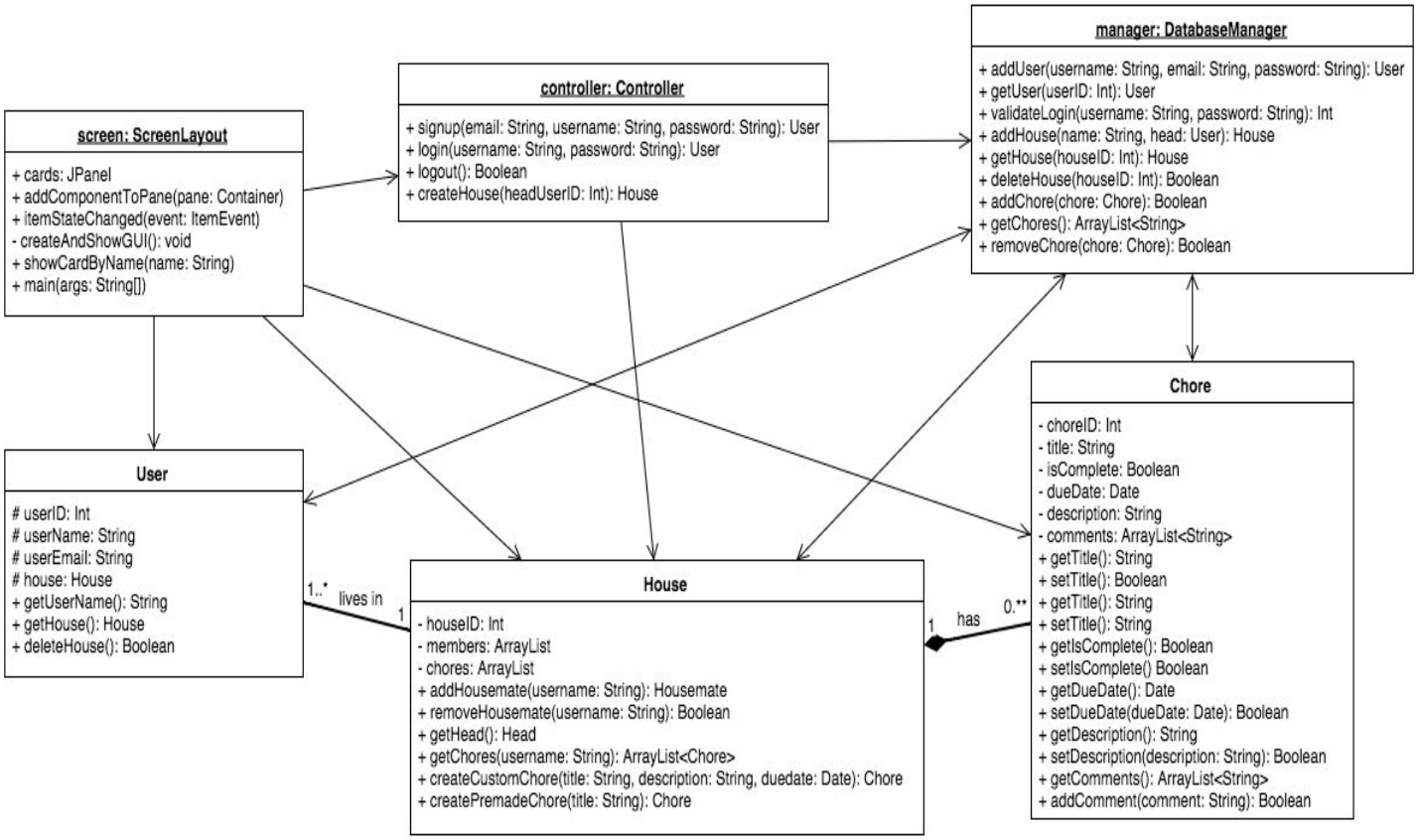
And lastly, we have a set of controller classes (House, Chore, User, etc.) that make decisions about the actual behavior of the application. Functions from these Classes are called when the user interacts with the View classes.

Our implementation of MVC allows us to easily delegate tasks to specific classes, maintain high cohesion, minimize coupling, and uphold data integrity. In the design process, this pattern also allowed us assign programming tasks to different team members while minimizing internal conflict.

To implement the MVC architectural pattern, we extensively used the Observer design pattern. Though we could have written our own classes, we opted to use the Observer interface and Observable class provided by Java. This helped ensure correctness and efficiency. We made the House class inherit from Observable, and added various panel classes as Observers. When a new Chore was added to the House, it notified all interested classes and updated the view with the changes.

3.

Class diagram from Part 2:



In general, our original design was not too different from the final implementation. Classes like User, House, and Chore stayed mostly the same. Some details changed (for example, House stores Chores in a HashSet instead of an ArrayList), but the overall structure was designed fairly well. One large difference in the controller classes is the addition of different types of chores (subclasses of Chore) and the addition of a factory to create those Chores. We didn't realize we would need the factory design pattern because we hadn't originally anticipated having different types of chores. We chose to implement several types of chores because it was a natural potential extension of the application's functionality.

The DBManager class remained essentially true to design. It retained the same functionality, though we implemented the ORM database using Hibernate in the final product. We did not anticipate this, but it has no effect on the class diagram.

We did, however, change the view classes and some of the relationships between classes in the final design. The view portion of MVC architecture was represented by only the ScreenLayout class in our original diagram. In our final product, 7 classes are responsible for the GUI. We needed to change the design so much because none of our team members were very familiar with Java Swing. In fact, only one member had previously done extensive work with GUI applications. We did a fair amount of research before part 2 of the project, but didn't

understand fully how the implementation would work. In the future, to avoid this issue, we would code a small sample application or consult someone that has worked with the toolkit, so that the design better reflects the intended final product.

The relationships in the class diagram also changed because we made a solid effort to separate concerns and follow the MVC architectural pattern. We could have made a better design plan for this aspect initially but didn't realize that we could have organized our class structure in a much more efficient and modular way.

4.

The biggest lesson we learned from the full process of designing and implementing a system is that there is a lot of value in thoroughly planning a system's functionality, even when the final product may not look much like the planned design. Our project changed significantly from initial design to final implementation, but it's clear that we couldn't have decided on desired features as we wrote the code. Thorough planning is important because it's also easy to falsely assume that all team members agree on how a certain system should look and work. We learned that these assumptions can create big impediments later on in the project, but that they can be relieved by documenting all decisions for team members to review. Even though we couldn't simply transcribe our design into functional code, the design analysis was valuable because it helped us think about the implementation details of our product.

We also learned how to take a large and complex project composed of many different functional parts and break it down into more manageable tasks. This allowed us to delegate independent tasks to different team members. In this way, one individual's work did not depend on another's progress, and we worked efficiently to create *Choreganizr*.