Tyler Scott
Chris Rhoda
Jackson Markowski
Sean Moss

CSCI 4448 - Object Oriented Analysis and Design
Professor Boese

06 - Media Management System: Final Report

1. List the features that were implemented (table with ID and title).

Implemented Features

ID	Title
UC-01	Create account
UC-02	Login to account
UC-03	Logout of account
UC-04	Rent Media
UC-05	Return Rented Media Before Deadline
UC-06	Purchase Media
UC-07	View Personal Media Inventory
UC-08	Search System Inventory
UC-09	Add Media to System Inventory
UC-11	Edit Media in System Inventory
UC-13	Review/Rate Media
UC-14	Sell Back Purchases
UC-15	Add Funds to Credit Balance

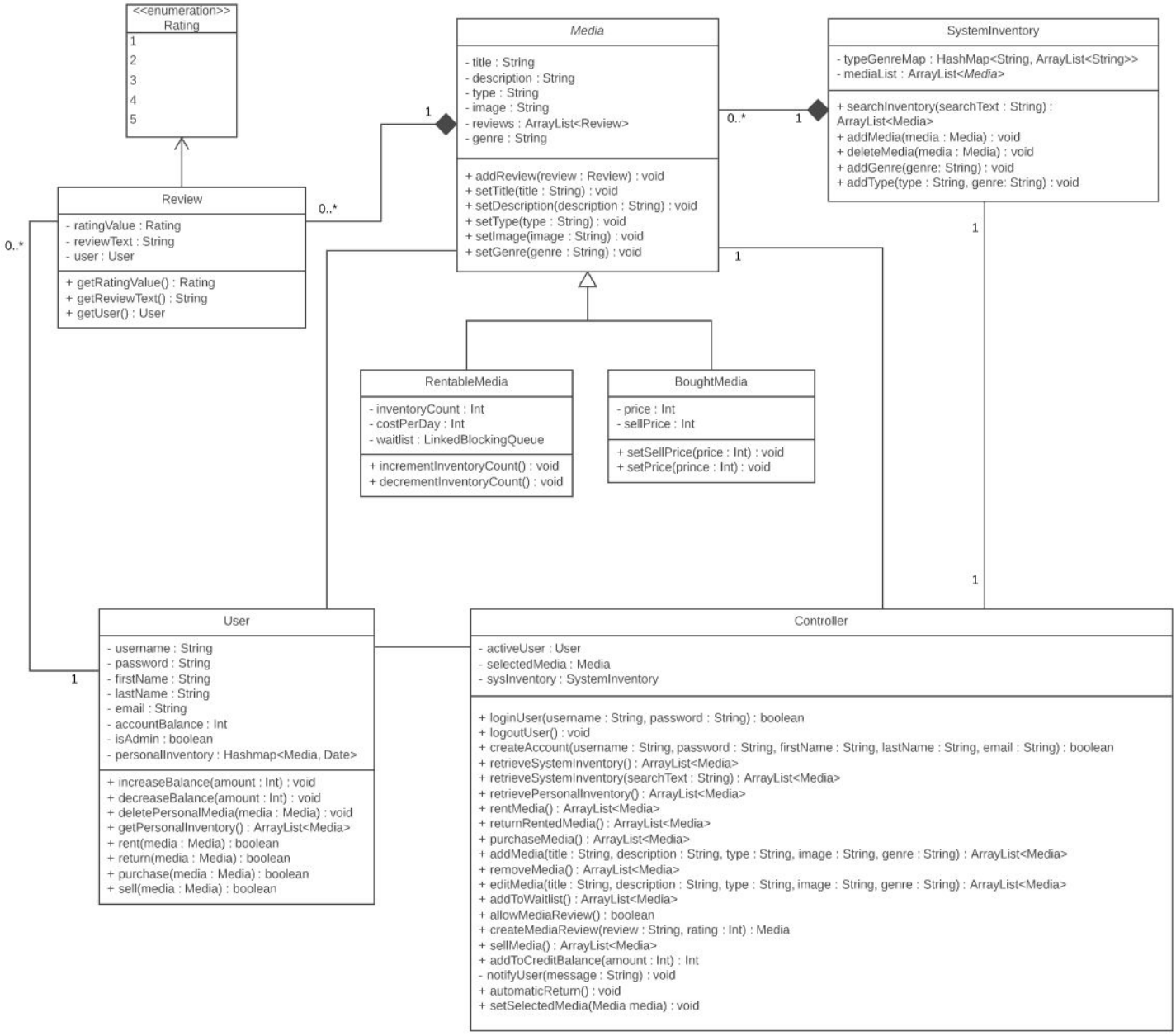
2. List the features were not implemented from Part 2 (table with ID and title).

Features Not Implemented

ID	**Title**
UC-10	Remove Media from System Inventory
UC-12	Add to Waitlist for Unavailable Rentable Media

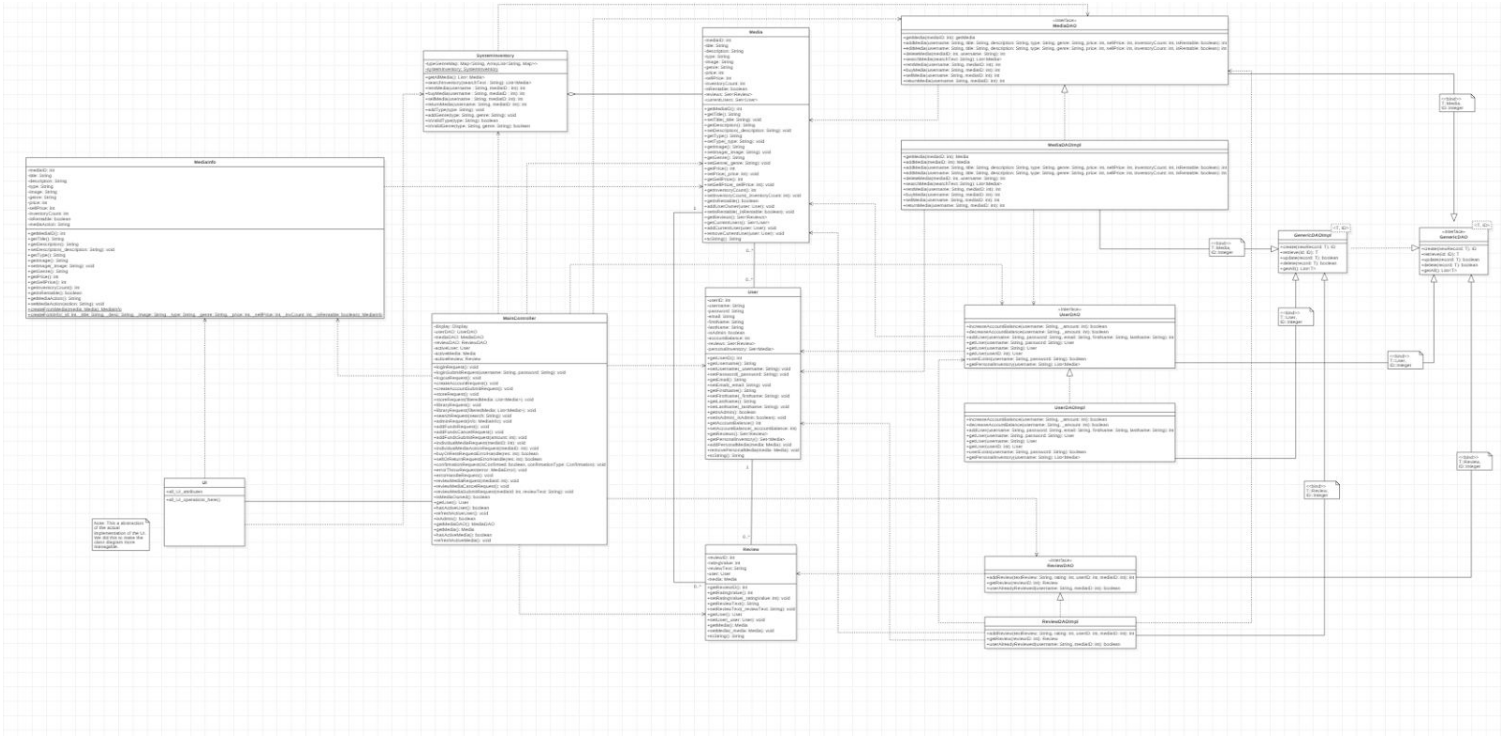
3. Show your Part 2 class diagram and your final class diagram. What changed? Why? If it did not change much, then discuss how doing the design up front helped in the development.

Part 2 Class Diagram:

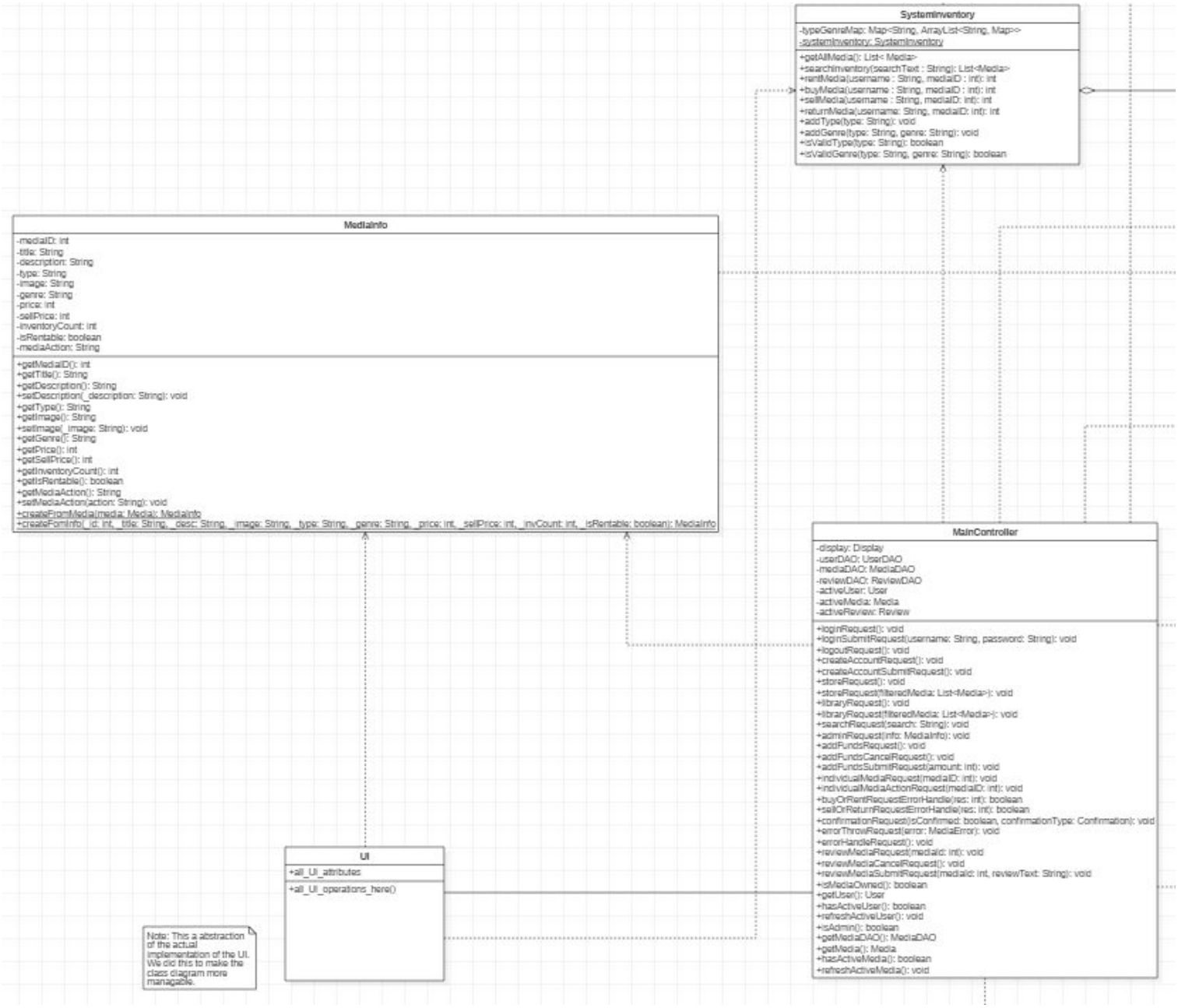


Final Class Diagram:

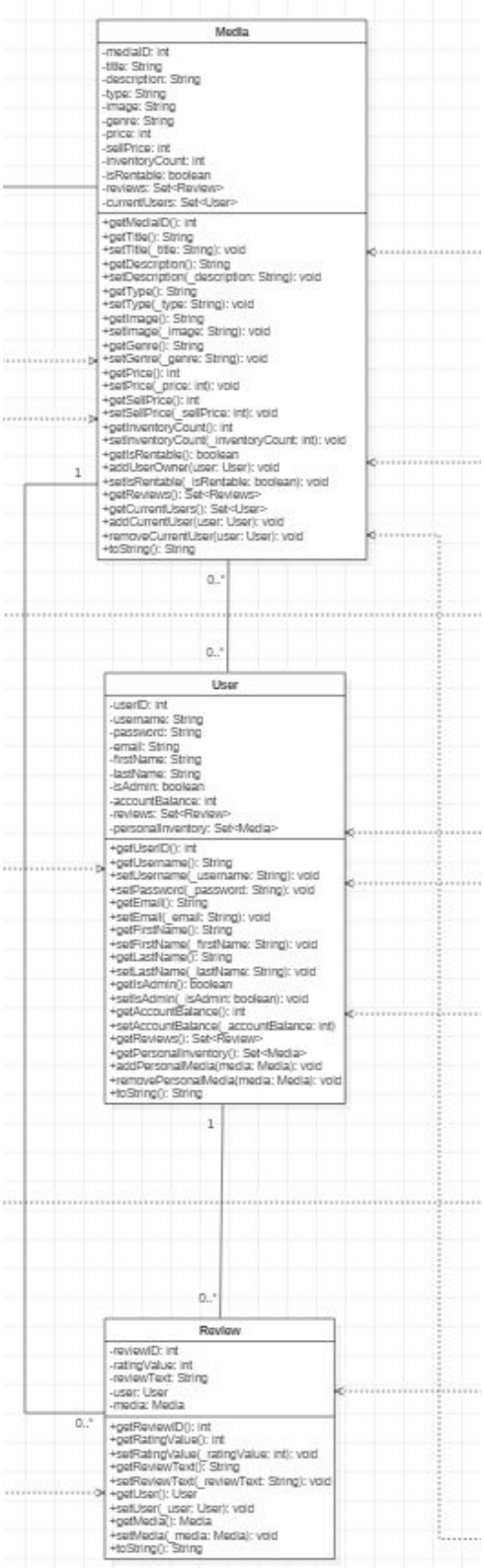
This is the full class diagram. The class diagram image is very small, so we included images of subsections below.



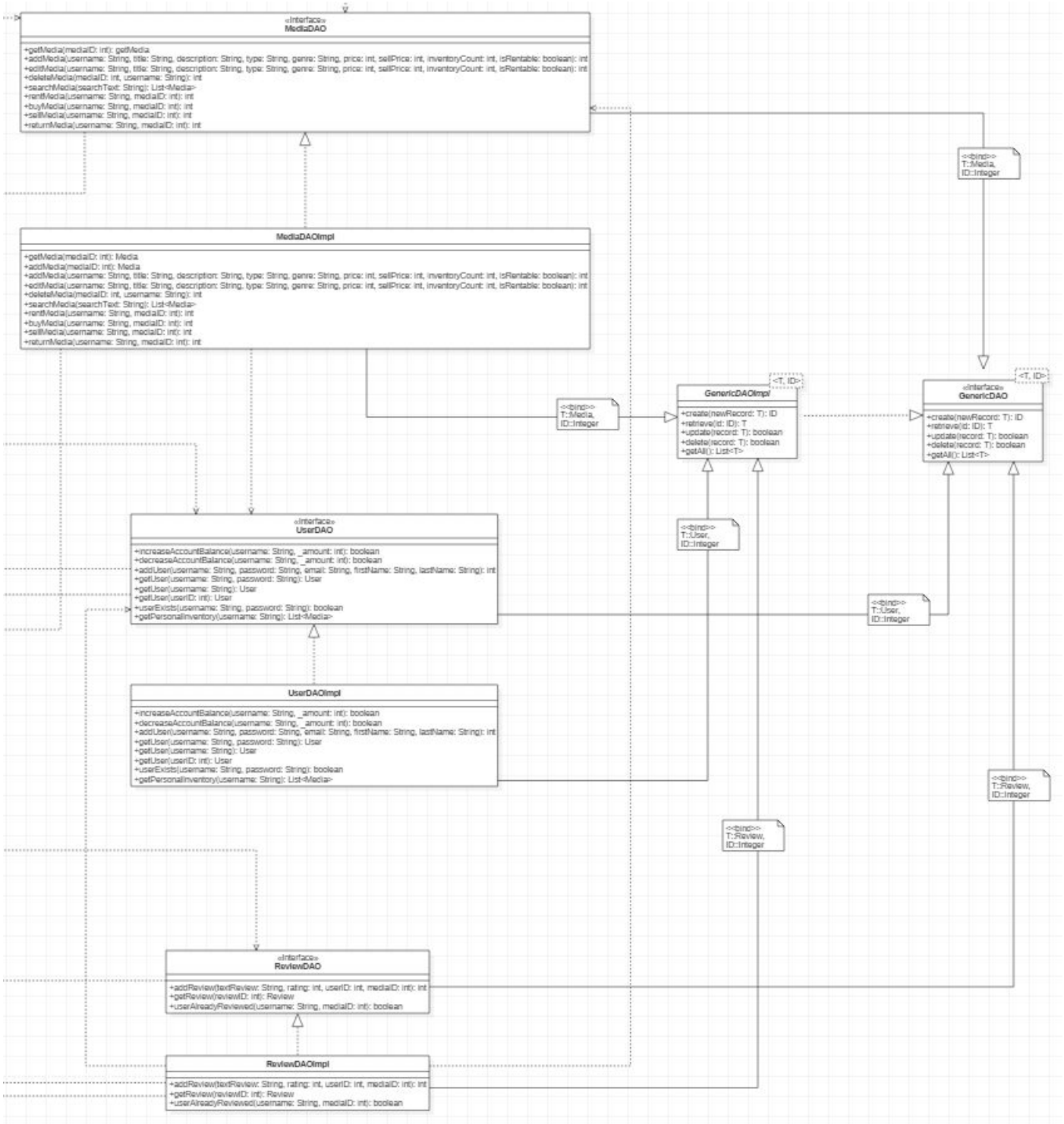
This subsection shows the relationships between MediaInfo, SystemInventory, MainController, and the UI. The full UI has been simplified to be represented by a single class, to make the class diagram more manageable.



This section shows the relationships between the Media, User, and Review classes. These are the @Entity classes for Hibernate. They map to tables in our MySQL database.



These classes were necessary for the DAO pattern that we implemented. This diagram shows the relationships necessary for Media, Review, and User DAOs to perform their functions properly. The Generic DAO uses generic typing to allow reuse of CRUD operations.



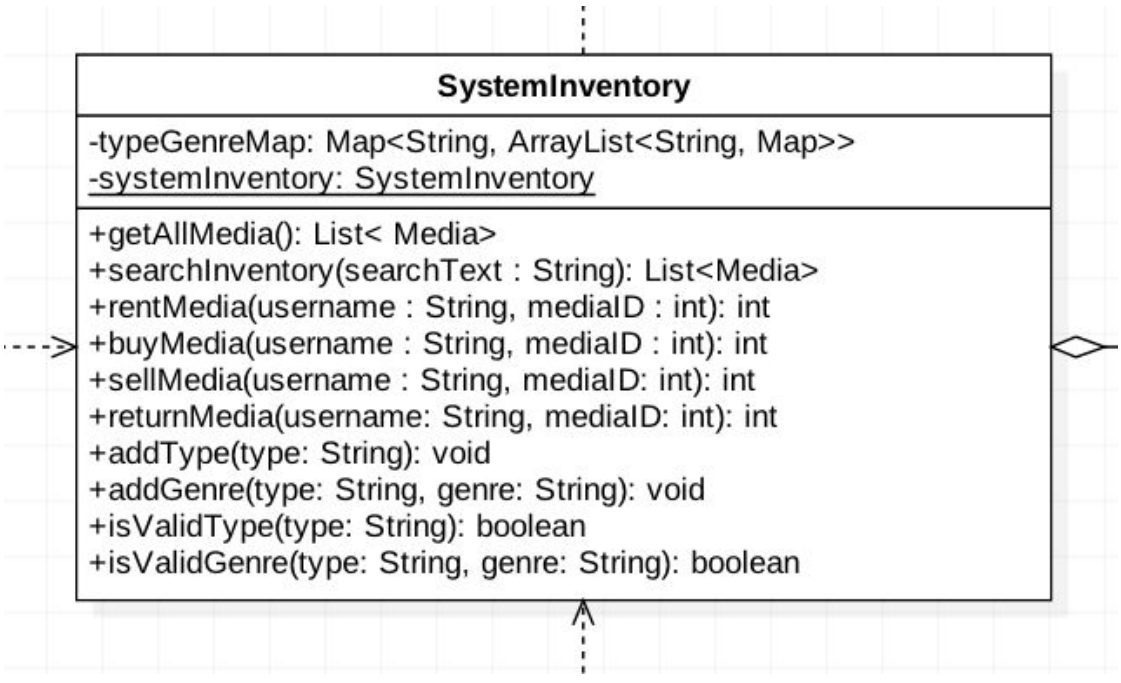
Overall, the class diagram from part 2 changed dramatically from the final class diagram. The main changes occurred in the “model” section of the system. At the time of creating the class diagram we were unfamiliar with ORM and specifically, Hibernate, so the persistence classes were completely redesigned. In particular, we used a Data Access Object (DAO) design pattern which extracted the create, read, update, and delete (CRUD) database operations from the business logic and allowed for sharing of the CRUD implementation throughout all classes in the model. On top of those changes, we simplified the design of the Media classes. Initially, we needed to differentiate between Rentable and Buyable media, but with Hibernate and persistent storage, we stored that information in the database as a column, which allowed us to delete the hierarchy of media-related classes. The Controller/MainController stayed similar to its initial functionality. The controller served as a bridge between the model and the user interface, and that is consistent with the final class diagram. One thing we would change if we could go back, would be to split the controller into a package of smaller controllers, where each specific controller would be used to delegate user requests for similar types of functionality. For example, there would be an authentication controller for logging in and out, as well as a transaction controller for dealing with renting and buying media.

4. Did you make use of any design patterns in the implementation of your final prototype? If so, how? Show the classes from your class diagram that implement each design pattern (each design pattern as a separate image in the .PDF). If not, where could you make use of design patterns in your system? Show a class diagram of how you could implement each design pattern and compare how it would change from your current class diagram.

Our team made use of several design patterns in the model and user-interface packages of the project. Some of the design patterns may seem like overkill, but the goal for our team wasn't to design the system as simply as we could, but to learn how design patterns work in large scalable systems.

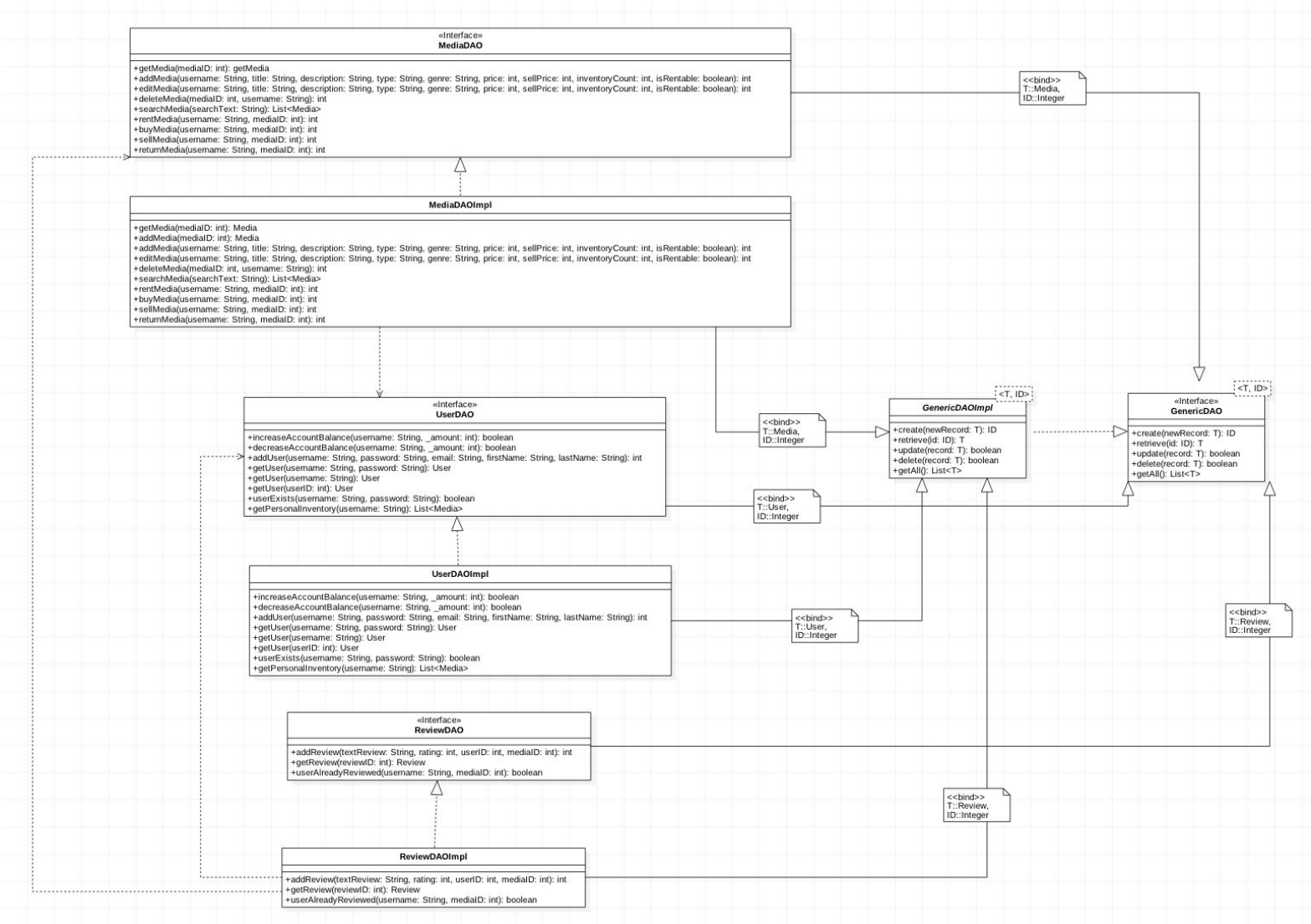
In the model portion of the codebase, we made use of two design patterns, Singleton and Data Access Object (DAO).

The singleton pattern is used to ensure that only one system inventory exists while the application is running. While many different pieces of code access the system inventory, only one instance of that inventory should exist. The implementation of the system inventory was relatively simple, but the methods are key to the entire system working together. Most requests made by the user accessed this singleton whenever any type of media was retrieved. This is the main reason we thought a singleton design pattern would fit in the code.

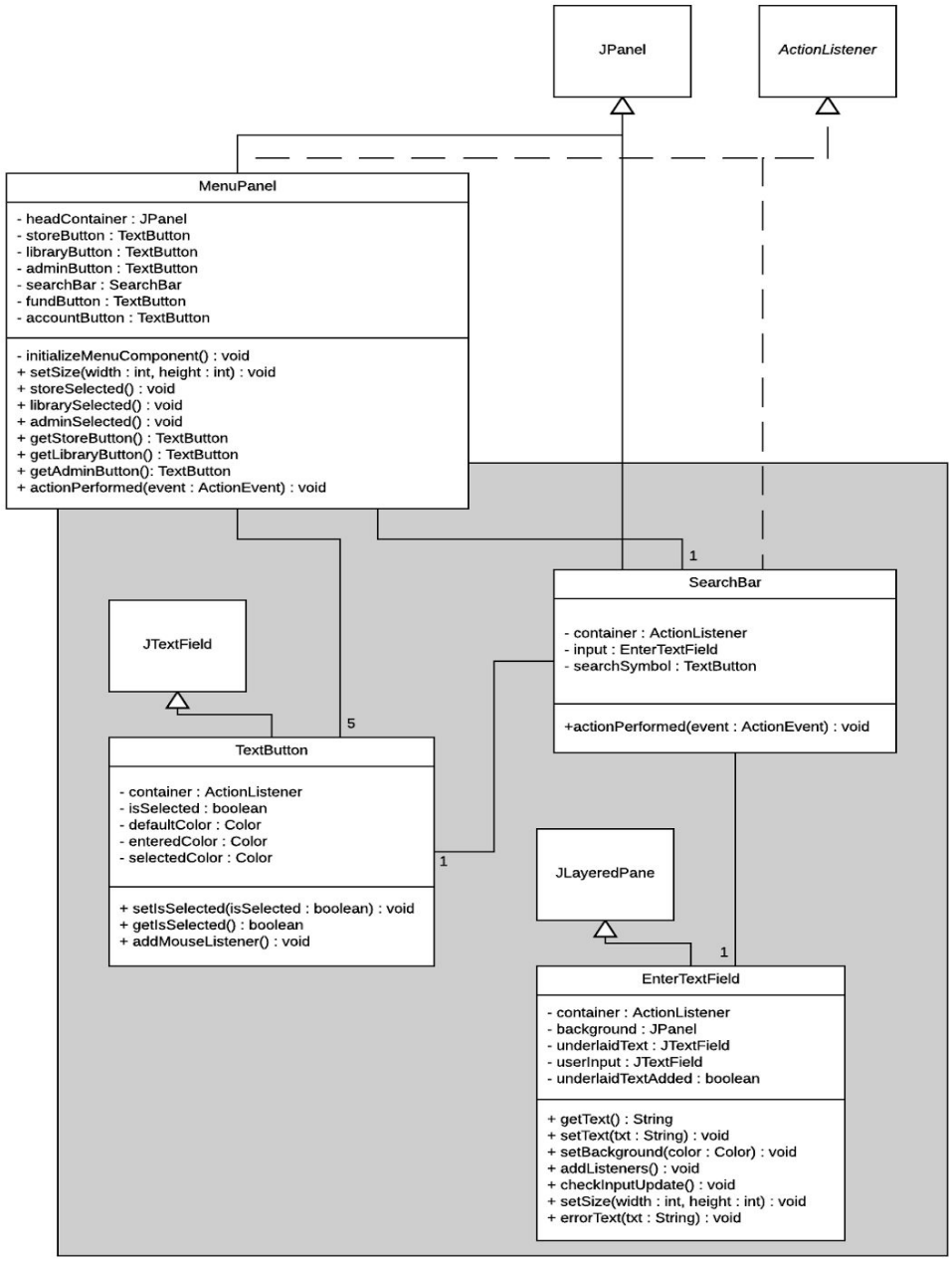


The second design pattern we implemented in the Media Management System was the data access object (DAO). This design pattern is not a Gang of Four pattern, but was created when object relational mapping frameworks, such as Hibernate, became popular. A data access object is a stateless class that handles accessing persistent storage through CRUD operations. Initially, we implemented a data access object for each table in the database, but this produced huge amounts of duplicated code for CRUD operations. The DAO design pattern is an architectural pattern that is used to strip out the CRUD operations from each table's individual access objects, allowing them all to share the same operations through generic typing. By implementing the "Generic DAO" interface, the specific DAOs in our system such as "Media DAO", "User DAO", and "Review DAO", could share the CRUD operations by binding the correct types to the generic types in the Generic DAO. This design pattern is crucial for systems that use large relational databases because implementing CRUD operations for each specific table would lend to thousands of lines of duplication.

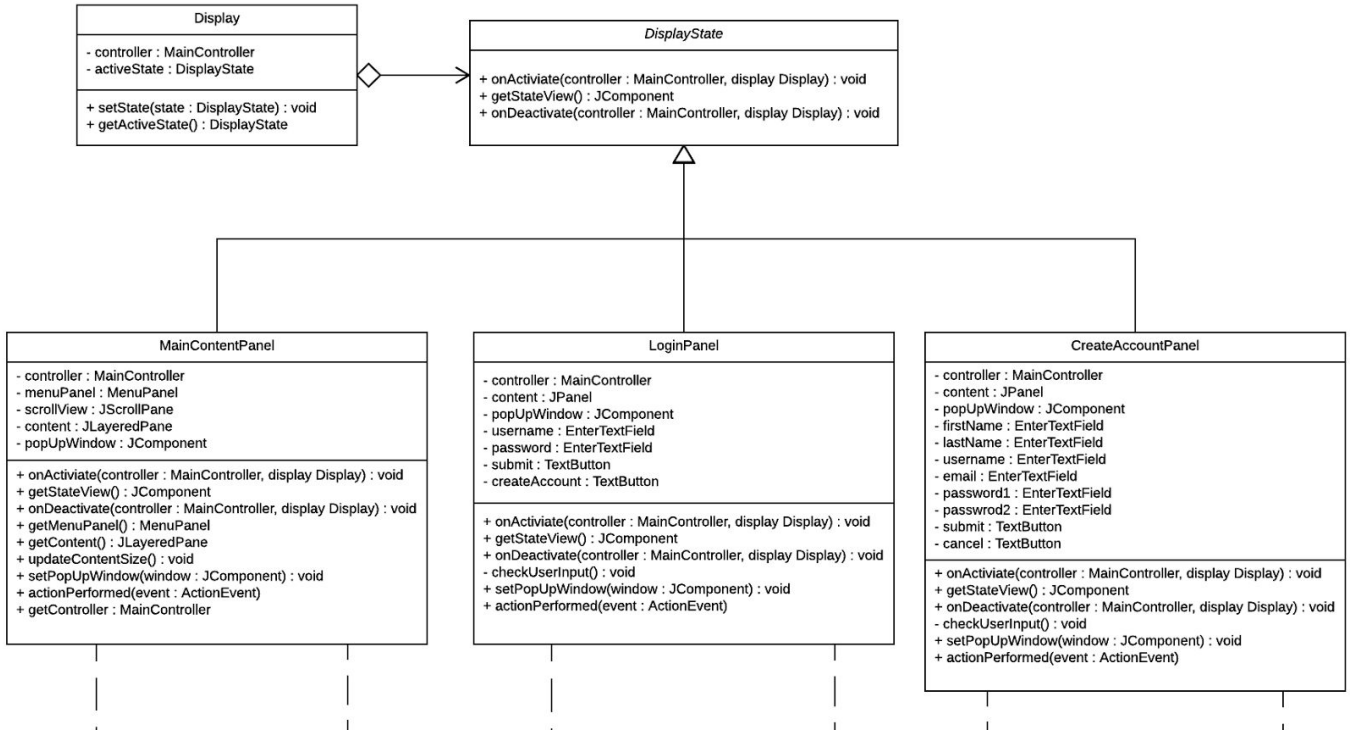
In the Media Management System, we used the DAO design pattern for our database. As I mentioned above, this was most likely overkill due to how small our project is, but we learned a lot about scalability of database applications through implementing this design pattern.


«Interface»
MediaDAO
+getMedia(mediaID: int): getMedia
+addMedia(username: String, title: String, description: String, type: String, genre: String, price: int, sellPrice: int, inventoryCount: int, isRentable: boolean): int
+editMedia(username: String, title: String, description: String, type: String, genre: String, price: int, sellPrice: int, inventoryCount: int, isRentable: boolean): int
+deleteMedia(mediaID: int, username: String): int
+searchMedia(searchText: String): List<Media>
+rentMedia(username: String, mediaID: int): int
+buyMedia(username: String, mediaID: int): int
+sellMedia(username: String, mediaID: int): int
+returnMedia(username: String, mediaID: int): int
<<bind>>
T::Media,
ID::Integer
MediaDAOImpl
+getMedia(mediaID: int): Media
+addMedia(mediaID: int): Media
+addMedia(username: String, title: String, description: String, type: String, genre: String, price: int, sellPrice: int, inventoryCount: int, isRentable: boolean): int
+editMedia(username: String, title: String, description: String, type: String, genre: String, price: int, sellPrice: int, inventoryCount: int, isRentable: boolean): int
+deleteMedia(mediaID: int, username: String): int
+searchMedia(searchText: String): List<Media>
+rentMedia(username: String, mediaID: int): int
+buyMedia(username: String, mediaID: int): int
+sellMedia(username: String, mediaID: int): int
+returnMedia(username: String, mediaID: int): int
«Interface»
UserDAO
+increaseAccountBalance(username: String, _amount: int): boolean
+decreaseAccountBalance(username: String, _amount: int): boolean
+addUser(username: String, password: String, email: String, firstName: String, lastName: String): int
+getUser(username: String, password: String): User
+getUser(username: String): User
+getUser(userID: int): User
+userExists(username: String, password: String): boolean
+getPersonalInventory(username: String): List<Media>
<<bind>>
T::Media,
ID::Integer
GenericDAOImpl
<T, ID>
+create(newRecord: T): ID
+retrieve(id: ID): T
+update(record: T): boolean
+delete(record: T): boolean
+getAll(): List<T>
«Interface»
GenericDAO
<T, ID>
+create(newRecord: T): ID
+retrieve(id: ID): T
+update(record: T): boolean
+delete(record: T): boolean
+getAll(): List<T>
<<bind>>
T::User,
ID::Integer
UserDAOImpl
+increaseAccountBalance(username: String, _amount: int): boolean
+decreaseAccountBalance(username: String, _amount: int): boolean
+addUser(username: String, password: String, email: String, firstName: String, lastName: String): int
+getUser(username: String, password: String): User
+getUser(username: String): User
+getUser(userID: int): User
+userExists(username: String, password: String): boolean
+getPersonalInventory(username: String): List<Media>
<<bind>>
T::User,
ID::Integer
<<bind>>
T::Review,
ID::Integer
«Interface»
ReviewDAO
+addReview(textReview: String, rating: int, userID: int, mediaID: int): int
+getReview(reviewID: int): Review
+userAlreadyReviewed(username: String, mediaID: int): boolean
<<bind>>
T::Review,
ID::Integer
ReviewDAOImpl
+addReview(textReview: String, rating: int, userID: int, mediaID: int): int
+getReview(reviewID: int): Review
+userAlreadyReviewed(username: String, mediaID: int): boolean

Within the UI portion of the system, facade was one of the patterns used. The image below showcases the basic implementation of this pattern within the MenuPanel class. The MenuPanel acted as the menu at the top of the screen. A good amount of classes needed to implement a menu and update it in order for there to be a functioning menu displayed. Since there are many components that make up the menu it made sense to wrap them all in a single class. This allowed for other classes to only have to call a method on MenuPanel in order to cause changes to all the menu components, eliminating the need for other classes to worry about all the components that make up the menu. The UI uses facade in a few other classes, not depicted below as they all follow a very similar format.



The final major design pattern the user interface utilizes is state. The main UI display of the system constantly switches between actions such as logging in, creating an account, media store view, personal library view, and many others. State allows for the display to have a simple way of transitioning between these various actions. Each state(logging in, main store view, etc.) implements the derived methods from DisplayState in order to allow the display to call these methods and cause the states to perform their individual functions. Specifically, the DisplayState interface exposed two functions: onActivate and onDeactivate, which are called during creation and removal to allow the state to change details and load information about the system. Each state was also provided a content panel and popup window controller to abstract out the creation and sizing the windows, allowing the state classes to focus on content and not overall design. (It should be noted that store, library, and other views extend MainContentPanel in order to have state functionality)



5. What have you learned about the process of analysis and design now that you have stepped through the process to create, design and implement a system?

After stepping through the process of creating, designing, and implementing a system, we learned several lessons about the process of design and analysis of object-oriented software. First off, the importance of preparation through detailed use cases, class diagrams, and sequence diagrams is key. We found that the structure of the UML diagrams helped the team when we began implementing the system, but that they also need to be malleable and able to change as the requirements of the project evolve. Specifically, it was extremely important to thoroughly think through the class diagram before coding because the vague, overlooked details in the class diagram led to unorganized code. As the system grew larger and larger, it was important to be careful where certain new features were implemented. We found that some of the last features implemented in our system produced some of the messiest code because the class diagram wasn't detailed enough. If this was a system in industry, there would need to be a lot of refactoring to get the code more clean and organized. This brings up the next point of scalability. Many times during the implementation phase, it is relatively easy to implement a new feature, but it may not scale. A big lesson learned during this semester long project was to think about and plan for scalability. True, the database may not have thousands or millions of records, but the code should not change regardless of the size of the database. The Data Access Object (DAO) design pattern may have been unnecessary for a system as small as this one, but the practice in coding these constructs are invaluable for a system that scales.