Real Estate Manager Design Document

Specification (Anudev)

Our program is designed to simulate a real estate platform that allows users to create buyer and seller accounts. Sellers can list properties on the platform by providing information about the listing such as the address, the number of floors, the number of bedrooms and bathrooms, the price of the listing, and any additional information. Buyers, on the other hand, can use a series of commands provided in a text-based menu to view properties by specific features, such as their price, the number of floors, the number of bedrooms and bathrooms, the location, and so forth. The program allows for three different kinds of properties to be listed—houses, townhouses, and apartments. Buyers can place offers on properties by messaging sellers and explaining their interest. Likewise, sellers can accept or refuse offers by replying to messages. The messaging system in the platform allows users to view their inbox and outbox, as well as their full chat history with another user. If a seller accepts a buyer's offer or decides to take down their listing, the program provides the option to delete any of their listings. Alternatively, buyers have the option to favourite the listings that they are particularly interested in and can view and edit their favourites at any time.

The program also allows for the creation of admins (as a secret option) and admins can ban or unban buyers and sellers as well as delete their accounts. Admins also have the ability to view the chat history between two users in order to ensure that the conversations taking place are ethical and do not violate the terms and conditions of the program. Moreover, admins can also create new admin accounts if a greater staff presence is needed to oversee the use of the program. Lastly, like all users, admins also have the option to view their login histories.

For phase 1 of the project, we implemented creating buyer and seller accounts, adding and deleting listings of the 3 types given above, searching by the features listed above, messaging users, and being able to view your inbox and outbox. For phase 2, we implemented the ability to favourite listings, create properties without a unit number, view your chat history (with timestamps) with another user, allow admins to view the chat history between any 2 users, and be able to sort properties by price in both ascending and descending order.

Design Decisions for Clean Architecture (Felix)

Our program follows Clean Architecture because each layer of Clean Architecture in our program is only dependent on the layer inside it. In order to incorporate a more intuitive file structure into our program, we created directories for each of the layers of clean architecture. This both makes our program easier to read and understand whilst also allowing us to visualize dependencies and ensure that the rules of clean architecture are being followed.

In phase 2, in order to follow the Dependency Inversion rule of Clean Architecture, we changed our use case classes so that they no longer instantiated CSV readers or writers within them. This was done in two steps. The first was to create a DataInterface interface class with a read() and write() method. This would then be added as a parameter to the constructor of the necessary use case classes enabling them to call the read() and write() methods of the classes implementing the DataInterface. The second step was to create the various CSVController classes that implement the read() and write() functions for the different entity classes. As described above, these would then be instantiated and passed into the correct use cases. This was done in two commits with the hashes d006986b12a7e8af1c2fcced1ed16081e0c3b4bf and 8e211d077a5c09a256d7d6389d024ef15b6e76a3. The first sets up the requisite classes and methods while the second implements them.

Design Decisions for SOLID Principles (Rana)

Our program follows SOLID principles because, for one, each class only has a single responsibility. To ensure that we followed this principle we created a separate class called GenerateUniqueID that generates a unique integer for each listing and message that can be used as an identifier. While we were originally planning on including such a method inside the CreateListing and SendMessages use cases this would have given these classes multiple responsibilities. Furthermore, we also followed the Dependency Inversion principle by ensuring that the use case classes do not explicitly create CSV readers and writers and created an interface to ensure there is a layer of abstraction between the controllers and use case classes.

During phase 2, in order to better uphold the SOLID principles, we moved all the calls to the data reading and writing methods from the main class into a separate class called StoreData. We then also moved the code required to run the program into a method within a separate class called ProgramController which is then called by the main method instead. This allows the newly created classes and the main class to now each have a single responsibility rather than the main class having multiple responsibilities. This was done in the commit with the hash ee44c325de9cd94ee7c0c3eef8c4bce7d871a9de. Additionally, to follow the Dependency Inversion principle, we changed the CSV controller classes to extend the abstract class CSVController which in turn implements the interface DataInterface. This allows the methods within the use case classes to simply create an instance of the interface in order to call the read and write methods within the controller classes without violating the Dependency Inversion principle. This was done in the commit with the hash 8e211d077a5c09a256d7d6389d024ef15b6e76a3.

Design Patterns (Andrew)

We implemented the Dependency Injection design pattern to avoid circular dependencies in the gateway classes because originally, creating instances of the gateway classes relied on the use case classes and creating instances of certain use case classes depended on gateway classes. To fix these dependencies, the read method of the gateway classes was modified so that it returned the contents of the CSV file that it read rather than attempting to also store that data in the corresponding containers. The task of storing the data in the containers was done instead by the read() method of the use case classes. The commit hashes for these fixes were d006986b12a7e8af1c2fcced1ed16081e0c3b4bf and 8e211d077a5c09a256d7d6389d024ef15b6e76a3.

We also created a ProgramController to reduce the size of our main method which originally had too many dependencies. Whilst this wasn't exactly any particular design pattern since none of the patterns exactly fit what we wanted to do, the idea for this change was inspired by the Builder design pattern since the ProgramController created instances of the necessary classes that were needed to run the program such as StoreData, LoggedInManager, and LoggedOutManager and populated the StoreData class with the data from all the CSV files when starting the program and wrote all the data in the containers to the CSV files before terminating the program. The commit hash for this change was ee44c325de9cd94ee7c0c3eef8c4bce7d871a9de.

Major Refactoring in Phase 2 (Eren)

The biggest and the longest refactoring we did in phase 2 was the changes we did to our CSV file processing system(s). Originally, the use cases were directly responsible for modifying CSV files, but we quickly realized that was not the proper way as it did not adhere to the principles of clean architecture. Afterwards, the first thing we did was create separate classes (one for each CSV file) that did the readings and writings. Additionally, we made it so that we read and process all the CSV files before the program starts, and write into them when the program ends. Then, we realized that we had multiple (3) ways of processing CSV files in the program. Realizing that this should be consistent, we modified the classes to all use the process described above. After this, to make the Main method shorter, we made a class that calls all of the reading and writing methods of these CSV processing classes. Finally, we decided that in order to be able to do dependency inversion without circular dependencies we would have to separate the read and write methods; and then implemented the dependency inversion design at the last step of this refactor.

A slightly bigger refactor was the shortening of the main method, with the use of the ProgramController class and instantiating the use case classes when we decided that they were ready for use, and abstracting that process into the constructors of the controllers.

There were some relatively minor refactorings as well, such as refactoring how messages are stored so that we can easily make the feature of viewing chat histories and we can display more detailed information about messages (namely, the time they were sent).

How the Program Follows the 7 Principles of Universal Design (Rohit)

Principle 1 - Equitable Use

The input and menus work with simple keyboard input and console output on display. This is the same for all users. Braille keyboards and windows output to speech can be used for blind users. Hence, we provide the same means of use for all users: identical whenever possible.

Principle 2 - Flexibility in Use

We give users the ability to list and buy properties classifying them as buyers and sellers. Hence, we provide choices in methods of use.

Principle 3 - Simple and Intuitive Use

Our UI is very clean and easy to understand. We provide users with accurate error and exception messages. Consider a login to a banned user. In this case, they accurately receive the message “This user is currently banned, please try again later.” Hence, we provide effective feedback during and after task completion.

Principle 4 - Perceptible Information

Our UI is made of clean menus that differentiate each option clearly. For example, once you log in as a user there is a clear menu for options: Press 1 to see your login history, Press 2 to create a user, Press 3 to ban or unban a user, etc. Hence, we differentiate elements in ways that can be described.

Principle 5 - Tolerance for Error

We have made error and exception classes to catch most of these cases. A good example would be to consider if a user tries to create an account with a username that already exists, we catch this error and throw the following message "That username is taken, please enter another one". Hence, we provide warnings of hazards and errors.

Principle 6 - Low Physical Effort

The program works on keyboard input and we implemented menus that take inputs from the keypad, hence minimizing the user's effort.

Principle 7 - Size and Space for Approach and Use

The input and menus work with simple keyboard input and console output on display. Hence, we make reach to all components comfortable for any seated or standing user.

Everyone's Contribution to Phase 2

Anudev (GitHub - anudevgill):

- Inverted CSV dependencies by moving the location of the interface and creating read and write methods in the use case classes that use the interface's methods
- Populated these methods with the code originally in StoreData() and simplified that class to call the read and write methods in the use case classes
- Added CSV support for the new message chatting feature so that timestamps are also stored in CSV files and read from the files when populating the containers
- Created the presentation and demo
- Created the specification section of the design document
- Helped debug errors

Rana (GitHub - rjak03):

- Implemented the favoriting and unfavoriting listing feature functionality within the LoggedInManager class and made necessary changes in related use case classes
- Added support for the deletion of users with previously sent messages
- Wrote the "Design decisions for SOLID principles" section of the design document and created the uml diagram

Eren (GitHub - FakeDeepLearner):

- Implemented the initial phase of the csv controller classes for all files.
- Implemented the FavoriteAndUnfavoriteListing class
- Added the feature of viewing message chat histories to LoggedInManager
- Wrote the "Major Refactoring in Phase 2" part of the design document.
- Took part in debugging small errors.
- Added prompts to the UserInterface
- Took part in refactoring the csv reading and writing process.

Rohit (GitHub - rshetty22166145):

- Added the Sort Listing by Price Feature making changes in LoggedinManager and ListProperties functions. Made necessary Changes to test this feature in ListingProperties Tests.
- Wrote “How our program follows the 7 principles of Universal Design” of the design document

Felix (GitHub - ff-zhang)

- Implemented the classes and functions required for the CSV dependency inversion
- Refactored the Main class such that instances of the use cases are instantiated within controllers themselves instead of the Main.main() method
- Increased the coverage of our test suite to include all the use case classes
- Wrote the “Design Decisions for Clean Architecture” section of the design document

Andrew (GitHub - aychun):

- Created MessageChat class to implement chat feature
- Added Comparable, DateTime variables, and other relevant methods to the Listing class
- Added unit tests, fixed bugs in the sendMessages class, and fixed/refactored the IntelliJ warnings for the final submission.
- Wrote the “Design Pattern” section of the design document