Design Document 2:

Patterns Implemented
Four design patterns make up the essence and structure of our program: the Facade, Dependency Injection and Builder, and strategy pattern.

The Facade Pattern:

We had six UseCase classes (each of the ReviewManagers, UniversityManager, UserManager, and ProfileManager). To run the work of Phase 2 and ensure that all data is saved, we implemented a MainManager that stores an instance of each of these managers as attributes and delegate relevant data manipulating calls to them (for instance, username and password will be saved in UserManager, and CourseReviews will be saved in CourseReviewManager). This facade benefits our Gateway Read-Writer as it reads the entire MainManager to serialize and deserialize instead of having to do so on each Manager.

The Builder Pattern:

The aforementioned ReviewManagers themselves function as builders. Our Reviewable and Review entities do not store anything other than their own information. Thus, for the Reviewable Profile page to be created, the ReviewManagers handle the jobs of:
* Creating Reviewable Profiles
* Creating Reviews
Adding Reviews to associated Reviewable Profile to be displayed to the viewer on a Profile page (stored within the ReviewManager).

The Dependency Injection:

The dependency injection was a key factor in our ability to form the Reviewable pages. Following Clean Architecture and with the usage of user inputs, we formed our entities' constructions from the ReviewManagers with user inputs as arguments for the command, passed in these objects as arguments for any following commands and saved the User running the program by calling the UserManager to save their information and pass it as arguments for other Managers to use. Another example of dependency injection is in our implementation of AppController, which serves as the text UI of the program. In order to perform functionalities such as track user history and edit profile, the AppController must keep track of the current user that is logged into the system. This was achieved by passing a User object returned from LogInController into AppController. Notice that currentUser is treated like a variable, and the controller class never accesses any of its attributes or methods, thus preventing us from violating clean architecture. Finally, we have different Comparators for sorting and each sorting method in subclasses of ReviewManager just takes in a Comparator object to do the sorting. This pattern allows us to avoid ”Hard Dependency” and allow testing easier.

The Strategy Pattern:

We used the Strategy Pattern to make it easier to implement different versions of our sorting algorithm. We have different strategy classes such as SortByOveralRatingComparator and SortByVotesComparator which all implement the Interface Comparator<Review>. Under each class, they sort reviews in their own ways. For example, SortByOveralRatingComparator can sort reviews based on their overall rating, and SortByVotesComparator can sort reviews based on their votes. In the ReviewManager class, there is an abstract sort method that takes in a Comparator<Review> as an argument. Then, we implement sort methods for ReviewManager's subclasses, and we can call their own sorting methods by using different strategies for sorting. This allows us to choose which sorting algorithm to use easily.

Patterns that we are not using.
Observer pattern:
We tried to implement an observer pattern before by making CourseReviewManager, ProfessorReviewManager, and DormReviewManager the Observable and ProfileManger the observer. In that case, when one of the ReviewManger mentioned above changes, such as a new review added by a user, it can update the ProfileManger to find the user's profile and attach the review to it. However, we found out that this will violate clean architecture in our program, because in order to update ProfileManger in Observable class, we need a name of the user who leaves that review from the controller layer. This means that multiple Manager classes which are use case classes will rely on controllers to provide extra information. This clearly violates clean architecture.