Note: This is a document for design document 2, 3, and 4.

Design Patterns (Design Document 2)

Strategy design pattern: The strategy design pattern was implemented in 3 use case classes (AccountManager, PostManager, and CommentManager).

- In all 3 use case classes, the use cases were the context in the strategy design pattern.
- In AccountManager, IAccountSorter is the interface of the strategy while AccountSorter is a concrete implementation of IAccountSorter.
- In PostManager, IPostSorter is the interface of the strategy while PostTimeSorter is a concrete implementation of IPostSorter.
- In CommentManager, ICommentManager is the interface of the strategy while CommentTimeSorter is a concrete implementation of ICommentManager.

As the classes suggest, the strategy classes are meant for sorting accounts, posts or comments. Using the strategy design pattern is suitable here as it prevents the use cases from having logic related to storing entities. That is, single responsibility principle is enforced. Additionally, it makes the program more open-closed because the use cases only depend on the interface of the strategy (by using dependency inversion). So, adding in a new implementation for the sorting strategies (in the future) will not require significant refactoring (if any).

Dependency injection: Dependency injection is used in various parts of the program. Here are some (non-exhaustive) examples of dependency injection.

- The dependencies of use cases (IReader, IWriter, and IAccountSorter) are “injected” via its constructor. For instance, in AccountManager, its constructor is AccountManager(IReader, reader, IWriter writer, IAccountSorter accountSorter).
- The dependencies of controllers (use cases) are “injected” via its constructor. For instance, in AccountController, its constructor is AccountController(ManagerData managerData).

With the use of dependency injection in use cases, it was used along with dependency inversion. This makes the code more open-closed since if there were to be new implementations of an IReader, IWriter or IAccountSorter, this can be passed to the use cases (without the use cases “knowing” about this change). Since the use cases only depended on the interface, no code within use cases would have to change as a result of a new implementation of its dependencies. In the controllers, the same idea of being more open-closed also applies for the same reason but to a lesser degree because the concrete implementation isn’t being hidden behind an abstraction like an interface or abstract class.

Design patterns unused / removed

Facade design pattern: In phase 1, we used a facade design pattern as various requests in a menu page of a CLI lead to different controllers being called. By using a facade, we are enforcing single-responsibility (since the facade is logic-free and each controller only contains logic to handle 1 type of request). And, it made the app more open-closed as new requests can be handled by creating a new controller and adding the controller to the facade.

However, in phase 2, we made the decision to not use the facade design pattern. This is mainly because having routes for URLs serves as the facade as we can redirect a URL to a specific handler which then calls an appropriate method in a controller. Essentially, due to accessibility of URLs on the web and handlers from the Undertow library, it eliminated the need of having facades in our architecture.

Simple factory pattern: At some point in phase 2, we used the simple factory pattern to construct handlers for the appropriate endpoint. However, as discussed in the second last TA meeting, Undertow offers the ability to use instance methods to replace handlers instead of requiring actual handler classes by using the java built-in method reference operator "::". To reduce the total number of files (to make searching for code easier and for the file structure to be more organized), we opt to get rid of handler classes and use instance methods instead. This eliminated the need of using a simple factory pattern in our architecture and allowed us to better categorize our handlers similarly to our controllers.

Design Patterns we didn't get time to add

Observer design pattern: The observer pattern is helpful to maintain consistency of states between 2 or more classes and can be used if there is some cause-effect pattern. In our case, we realize that the observer pattern would be a good design pattern to use for account deletions.

In terms of account deletions, the account deletion is an event that should be listened to by the comment and post managers so that all comments and posts created by the account to be deleted are also deleted. This is a clear anti-pattern for the observer design pattern to be used but due to time constraint, it was not implemented.

Code Improvements from Phase 1 (Design Document 3)

Reduction of controller classes: In our phase 1 submission, there were many files (20+) for controllers. The main choice for this file structure was the use of abstract classes and facade design pattern which made the code very open-closed but we sacrificed file organization.

As noted above, we no longer needed a facade design pattern in phase 2. As a result, we decided to condense the controller files by grouping related controllers that affect the same entities into the same files. This reduces the number of files we have and makes the code more organized and also makes it easier for us to find code.

Removal of data clump: In our phase 1 submission, all the controllers inherit from RequestController which stores many instance attributes. These attributes include gateway classes (which were removed by phase 2) and use case classes. This resulted in a data clump code smell as related classes / objects, particularly the 3 use case classes, are not grouped together in a class. To resolve this, we created a ManagerData class which is meant to bundle together the 3 use case classes and it is also responsible for storing the current user who is logged in for the session.

Using abstractions: In our phase 1 submission, there is a lot of usage of concrete data types on the left hand side of a variable declaration (e.g. ArrayList<Post> posts = new ArrayList<>();). These declarations occurred a lot in controllers, use cases and the data mapper. To make the code more open-closed (by allowing for the concrete data type to be more "flexible"), we now use abstract data types (e.g. List in place of ArrayList and Map in place of HashMap) on the left hand side of variable declaration to make the code we have more open-closed.

Removing unnecessary interfaces: One of the feedback from phase 1 was to remove unnecessary interfaces in our code (which caused the number of files to balloon). Taking this feedback into account, we removed the interfaces for the use cases classes. This made sense because controllers are 1 layer "above" use cases in clean architecture and do not need to interact with a use case through an interface.

Usage of view models: In phase 1, the controllers pass data to the presenters and the presenters are the ones to "clean" the data that is to be output to the users. To better enforce the single-responsibility principle, the ViewModel class in phase 2 properly "cleans" data. Then, the data is sent to the presenter using a Map data structure via the handler classes. This ensures that the only responsibility of the presenters is to pass the data to the Jinja templates and the response can be displayed to the user via the web.

7 Universal Designs (Design Document 4)

Equitable use

Equitable use asks that "… design [is] appealing to all users." This can be done in our application by allowing users to pick between light and dark mode. This allows users to select the color scheme that better suits their preference.

Equitable use also requires an application to "provide the same means of use for all users". This can be done by ensuring the way we write our markup is suitable for users using a screen reader (e.g. by using correct tags or semantic ordering of elements instead of stylistic ordering of elements).

Flexibility in use

Flexibility in use principle states that it is important for an application to "provide choice in methods of use". This can be done by adding recording buttons that can convert speech to text when creating a post. This ensures that there is some flexibility for users who may struggle to type with a keyboard, allowing them to use the app regardless.

Simple and intuitive use

One of the most important points in this principle is to "provide effective prompting and feedback during and after task completion". This was implemented in our phase 1 CLI app. However, due to time constraints, we did not have sufficient time to provide users with feedback after success or failure of performing a task in our web application. Given more time, we would like to give feedback whenever users perform any actions that may result in success or failure (e.g. login, sign up, etc).

Perceptible information

One of the subpoints in perceptible information is to "provide adequate contrast between essential information and its surroundings." This can be done by using WAVE (https://wave.webaim.org) or similar accessibility websites where we can check if there are any colors that do not contrast sufficiently with its surroundings for any users. If there are any contrast issues, they can then be fixed by choosing better color schemes.

Additionally, the perceptible information principle also emphasizes the "use [of] different modes … for redundant presentation of essential information". A lot of our buttons involve words. While words communicate the use of buttons effectively, users might be more used to seeing icons. These icons may be added to our website either with the use of images or SVGs.

Tolerance for error

The tolerance for error principle states that applications should "provide warnings of hazards and errors". We feel like warnings of hazards are closely associated with irreversible actions on the app (e.g. deleting an account, deleting a post, promoting a user, banning a user, etc.). Whenever a user requests to perform such requests, our application should have pop ups that ask the user to confirm that this is the user’s intended request and not just a misclick.

Low physical effort
The low physical effort principle states that we should try to “minimize repetitive actions”. In our case, scrolling through a long list of posts (within a profile or feed) to search for a specific post can be a very repetitive task. So, this can be avoided by creating some filtering mechanism (e.g. filtering for author, data range, post title etc.) which reduces the amount of scrolling needed by users.

Size and space for approach and use
The size and space for approach and use requires applications to “provide a clear line of sight to important elements for any seated or standing user.” This principle was carefully executed in this application when designing what the frontend of the app would look like and using different HTML elements (e.g. heading versus paragraph tags) to differentiate important information from regular information.