Specifications:

Our goal in this project is to create a marketing app whose main purpose is for users to sell products and other users to buy from them. The structure of the app resembles that of the Instagram app where the user is able to browse, look through other user's posts, and also search for specific categories related to their items of interest. The users are also able to follow other users, and have a feed to browse through. This feed is the posts that have been made by the users that this user follows. Users are able to buy the products inside the posts, make comments, add likes, etc. for the post. Other users can see the posts in their feed and decide to buy the product advertised inside the post. The users can add products to their shopping cart, and decide to continue browsing/searching for products, or can choose to buy their cart. Every user can add money to the app, and buy products through in app purchasing. The buying feature is yet to be implemented, since currently, the users cannot pay inside the app.

To explain our program in further detail, our program has 3 serialized files. First is the user.ser file which saves a hashmap of the user usernames (keys) to user (values) inside that hashmap. Whenever a user signs up, if the username is unique, their new user profile will be saved in this database. When the user chooses to sign in, the user will be accessed using the username, and this user will be given multiple options inside the app. The user can:

- 1.Search and buy
    - Here, the users can search for specific categories of items that interest them. They will receive a list of items in that category, and will be able to choose the index of the item that they would like to purchase (or they can go back to all options again)
    - If the user chooses to buy an item at a certain index of the list, they can input that index, and will add that item to their cart (a cart is a user attribute) When the user adds an item to their cart, they can choose to continue browsing, or to buy their cart
    - If the user chooses to buy the cart:
        - The following feature will be implemented in phase 2: The user will be able to purchase using the money that they have in the app to make in-app purchases, and the money is transferred to the user that has created the post. Then, this user can choose to continue browsing.
        - The users will have attributes such as address, etc. which shipping will be made possible
    - 2.Make a post
        - In phase 1, we implemented undo features. This allows users to go back to the previous command, and re enter their choice in creating a post. So if the user makes a mistake during the process of creating a post, they can go back to the previous step, instead of having to move forward in the program and create a post with incorrect information.
        - Making a post includes a process of choices for which the user can decide what features the product and the post has, including information about the product, and information about the post such as whether or not it will have a number of likes or comments.

- 3.Follow another user
    - This user has a list attribute of usernames of other users that this user follows. The user can follow other users, which will add them to their followers list, which will allow this user's feed to be updated whenever the users that they follow create a post (this feature will be implemented in further detail in phase 2)
- 4.Browse and buy
    - Users who choose this option will receive a list of all the posts that were posted by the users that they follow. They can look through this feed, choose the index of the item that they would like to purchase, and purchase in a similar process that was done in search. This way, users who are looking for trends or who would like to see what other users that they follow have posted, can choose this option.
- 5.Settings
    - This option will be implemented in more detail in phase 2. It is the option to change this users profile, or logout.
- 6.logout
    - Brings the use back to the welcome page where they can sign in or sign up.

Summary of the new features implemented in phase 1:

- Allow users to sign in and sign up, with their information saved in a serialized file.
- Allowing the users to search for products according to tags.
- Allow users to choose which products they wish to buy when they search, and be able to add the product to their cart
- Currently, if the user empties the cart, the user is not resaved into the user.ser file, which is a bug that will be saved in phase 2
- Creating posts
- Implementing the undo feature to avoid incorrect post creation.
- Saving the products and users and posts in serialized database.
- Allowing the user to go back to the welcome page/the page before.
- Give the user the option to browse, follow other users, create a post to add their product to their list of follower's feed, allow searching of products based on tags.

The features to be implemented in phase 2:

- Implement browse controller (already started in phase 1)
- Develop a web app
- Settings feature to allow users to navigate different settings such as modifying the user profile.
- Allow for browsing. This feature was partially implemented in phase1, more will be implemented later on.
    - Browse, and show the feed of the user as a post list.
    - Allow users to have followers, so that the feed of the followers can be updated automatically.
- Allow users to buy from the cart, empty the cart

        - Allow in app purchases by adding money to the app, and using that money in purchasing products in the app
- Implement enhanced security with passwords instead of just usernames
- Allow the user to see their own profile (their posts, their followers, etc.)
- Replace products with posts (currently, presenters show posts to the users, we wish to change that to presenting posts.)
- Fix code smells and long methods by implementing helper classes.

Packaging Strategy:
Our packaging strategy is packaging by component: This is what makes the program easier to navigate. Classes with related functionality were grouped together. Hence, classes that are similar in their function are easier to find.

SOLID:
Single Responsibility Principle
The single responsibility principle states that every class should be responsible for one part of the program's functionality. Throughout our code, we have demonstrated this with classes that do very specific tasks. An example of this includes the classes that create a user. All of these classes have very specific names which is also reflected in what they do, their tasks also being very specific. When you want to sign up (which means a new user is being created), from WelcomePageController, it calls the SignUpUseCase whose sole responsibility is to create a user and save it in a .ser file, this is reflected by one method in the class. Another example is when a user wants to follow another user, we call a completely separate use case class to implement this, userFollowingUseCase to do this. We use a similar logic when using different functions such as creating a product, there are different controllers and use cases however each class along the way has one task in the creation of said object. One drawback when trying to adhere to the single responsibility principle was that we made too many interfaces for the gateways which could have been merged together, this is something we will consider for phase 2.

Open Closed Principle
Open closed principle states that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification. Currently the user class can be a parent of user classes who can represent wholesale companies or individuals so we may plan to have different subclasses to represent different users. By making subclasses, we are opening the user to extension. The rest of the classes are strictly closed for modification.

Liskov Substitution Principle
The Liskov substitution principle states if we substitute a superclass object reference with an object of *any* of its subclasses, the program should not break. Currently, we have not made any classes with a subclass superclass relation so naturally our code adheres to this principle.

Interface Segregation Principle
Interface segregation principle states that clients should not be forced to depend upon interfaces that they do not use such as unused methods. The purpose is to reduce side effects and changes to code by splitting up the software into independent parts. As

mentioned with the single responsibility principle, all of our classes in our program have their own independent functionality. Because each class is unique and has a single responsibility, clients only need to call the classes which they need and ignore classes which are not needed, calling needed classes only does one thing so clients do not need to worry about unwanted side effects.

Dependency inversion Principle

The dependency inversion principle states that high level modules should not depend on low level modules; both should depend on abstractions, abstractions should not depend on details, details should depend upon abstractions. This is shown throughout our code with the use of interfaces which represent the abstractions that do not depend on details. Throughout the code are higher level modules such as various use cases such as ProductUseCase and lower level modules such as CreateProductGateway. Higher level modules such as various use cases and lower level modules such as gateways both depend on the aforementioned interfaces. This is done through dependency injection so that higher level use cases don't need to directly import from lower level gateways (violates clean architecture) and instead does this through interfaces.

Clean Architecture:

While implementing our project, we had a heavy focus on making sure to adhere to the fundamentals of clean architecture. Firstly, our four entity classes, which are "Post", "Product", "User" and "Undo" are only ever accessed directly by Use Case classes. If a new instance of an entity class needs to be created, or we need to access methods like a getter or a setter from our entity classes (or other methods like a toString method), we always go through a Use Case class.

Then, our controller classes are always the ones that directly take in information from the user and relay them to the Use Case classes, and present information back to the user. Therefore, our controller classes direct the flow of data in our program. We have one presenter class that presents information to the user when they want to search for a product to buy. This presenter class is called from another controller, which is lawful and adheres to clean architecture.

Therefore, we have made sure to follow the dependency rule of clean architecture, where dependency can only happen within the same layer or on an adjacent layer. (from outer layer to inner layer) Therefore, we have made sure that methods are only ever directly called when the classes are in the same layer of clean architecture, that is for example, a controller can call a method inside another controller directly or when we are calling methods from the outer layer to the inner layer. Also, we tried to minimize coupling when having dependency within the same layer by only having controllers calling other controllers and having this only happen when really necessary. (For example when we want to adhere to the S - Single Responsibility Principle in the SOLID principles).

Moreover, only use case classes are ever able to access our .ser files that live in the outer layer. In order to adhere to the dependency rules of clean architecture, instead of the use case directly calling the methods that access our .ser files, we have created gateway interfaces and gateway classes that implement those interfaces. Then, we use the Dependency Inversion Principle (DIP) to access the .ser files from the use case classes. That means that the use case classes have their constructor take in the gateway interface object as a parameter, and the main program creates the class that implements the gateway interface, call the use case constructor, passing it in. This allows the use case method to

access methods inside the classes that implement the gateway interfaces. Also note that the classes that implement the gateway interfaces are the only ones that access the .ser files (serialised files) and are only called by the use case classes.
As mentioned above, our program uses serialised files to save information, which is used throughout our program and this shows that our program works well with files (or an external database) and therefore adheres with Clean architecture.
All in all, our program adheres to Clean Architecture.

Design Patterns:

During phase 1, we implemented a feature when the user creates a product in our system, that allows them to undo the progress they input. It does this by implementing a version of the Memento design pattern in the class Undo. How we implemented it is as the user inputs information, it is stored in an instance of the Undo class, and if the user wishes to undo, it reverts to a previous instance of Undo class, so the user is able to input the information again, or undo again if they choose.

Use of Github Features:

We have used a lot of different Github features for our project to help facilitate our progress. We have used the Issues feature to make clear what we need to fix and implement, and who will be working on this to help us focus on coding rather than figuring out who needs to do what. We also made use of the pull request feature, where when merging branches into our main branch, we would make sure to have another member of our group look over the code, and see if it makes sense, and try and spot any errors that pop up. We also used feature driven branches, where the specific branches we made and edited, were related to one specific feature that we wanted to implement. This made it so it was easier to track who did what commits, and what functionality those commits changed in our program.

Code Style and Documentation:

Testing: We have implemented a variety of tests for most of the classes in our code that we feel require testing in the first place, without testing redundant methods. But there are still some more tests that we feel we should implement in our code.

What has worked well so far with your design:

- Adhering to clean architecture, by separating the layers into controllers and use cases, etc.
- Adhering to SOLID principles with well designed classes
- Serialization and saving to external database

Future questions:

- Currently, we want to allow users to add money to the app, added as an attribute, but we do not have a connection to other apps such as banks. How do we allow that connection?
- Is implementing messaging between users feasible with the remaining amount of time that we have (now until end of classes)

Summary of what has been worked on, and what to do next:

Yasmin:

- Phase 1 work:
    - Implemented serialization to allow for serializing products, users, and posts to database
    - Created gateways and controllers for the welcome page,
        - Implemented the options for logging in, signing up, and quitting the program from the welcome page
        - Developed controllers, use cases, and gateways to allow for signing into the program through the welcome page
    - Implemented user options use case and controller to provide the user with options upon signing into the program to
    - Implemented search controller and gateway
        - To allow users to search inside the product related serialized files and be presented with the products (in phase 2, we will implement this to present the posts instead of products)
    - Implemented presenter to present feed to the user
    - Implemented serialization for posts
    - Implemented serialization for products
        - Implemented the read writer for all serialized objects.
    - Implemented settings
        - Not used yet, although will be usable in phase 2
    - Implemented user serialization feature
        - Implemented class to be called by any other class that wants to save any changes to the user to the serialized file in the save changes gateway
        - Implement Allow users to add products to their cart
    - Writing the specifications of the final report
    - Packaging files for code organization, description of packaging strategies in the report.
    - Testing some gateways
- Phase 2 work
    - Implement web app
    - Implement browsing so that it is accessible
    - Implement settings
    - Implement in app purchasing by adding money to the app (add money method), and spending that money inside the app to purchase the cart

Umayrah:

- Phase 1 work:
    - Implemented the follow user feature
    - Refactored and signin and signup feature
    - Implemented and refactored buying feature (add to cart/empty cart, etc.)
    - Implemented create product feature
    - Implemented the browsing feature (not fully implemented - will be available for phase 2)
    - Implemented the settings feature (delete user) (not fully implemented - will be available for phase 2)

        - Writing about Clean Architecture in the final report
    - Phase 2 work:
        - Finish browsing feature implementation
        - Finish settings feature implementation
        - other

Sam

- Phase 1: work
    - Implemented Undo Class to allow users to Undo during the creation of products / any new features we add
    - Implemented Serialization for posts. add posts to a users feed/post list in the .ser file directly when they are created
    - Change classes around so that they don't violate clean architecture
    - Wrote tests for classes that I implemented
    - Writing about github features and design patterns in the final report

Phase 2: work

- Allow undo to delete serialized instanced variables
- Help with other things that require it

Albert

- Phase 1 work:
    - Writing about SOLID in the final report
    - Worked on product functions such as createProductController, createProductGateway, Product, ProductUseCase
    - Implemented post functions such as Post, PostManager
    - Helped with serialization of posts
    - Tested various gateways such as SignUpGatewayTest, SignInGatewayTest, SearchGatewayTest, SaveProductGatewayTest, SaveUserChangesGatewayTest
    - Fixed bugs throughout classes in login package
- Phase 2 work:
    - Continue writing tests for test driven development
    - Help implement browsing features
    - Help implement setting features
    - Look into potentially doing a GUI
    - Look into potentially doing a messaging system

Diego

- Phase 1 work:
    - Checked files for adherence to SOLID Principles
    - Bug fixes in tested files
    - Created Test classes for the follows classes:
        - SignInController
        - SignUpController
        - SignUpUseCase
        - WelcomePageController
        - BuyController
        - ListOfProductsPresenter

        - SearchController
        - UserOptionsController
        - UserOptionsUseCase
    - Ensured adherence to clean architecture principles
- Phase 2 Work:
    - Make tests even more thorough
    - Continue testing to ensure development is more efficient
    - Help developing tests for other classes
    - Help develop functional classes in order to make user experience smoother

Jacqueline

- Phase 1 work:
    - Created basic code to allow user to exit user options, exit search and buying
    - Created a modified version of the command line, SystemInOutTest to use for testing classes that require user input
    - Started work on testing classes that require user input
    - Made tests for classes in ProductFunctions, UserFunctions
    - Extra code for handling special cases for user input while creating a user/product
    - Fixed multiple errors in the code
- Phase 2 Work:
    - Continue modifying code to handle special cases in user input
    - Continue creating more tests
    - Search for more potential errors in the code
    - Help refine code to be more simple/concise