Design Document:

Scenario walk through:

At any time during the program, you have the option to go back by clicking the back button.

- Run the program from main
- Sign up by entering in a new username and password (Give the username tony)
    - If you make a mistake, reset
    - Press sign up
- Click sign in and SignIn with your new username and password
    - Press login
- After you sign in, you have many options to choose from
- First, create a post by clicking make a post
    - Fill in specifications about your post.
    - Remember the category of the post you enter so that you can search for it later
    - Click share
    - Click return back to options from the pop up window
    - Click search and input the category of the product that you just created
    - Press search
    - You can see a scroll bar with your item inside
    - Click back
    - Click back again from search
- Logout
- Now sign up as a new user with a different username (usernames must be unique)(give the username evan)
- Sign in with the newly created account
    - Login
- Click on find sellers (this the is follow feature)
    - Enter the username you created the first time (tony) to allow evan to follow tony
    - You will get the message that the user was followed successfully
    - If you put another user that is not in the database, or if you try the same user again, you will be unsuccessful.
        - You (evan) have now followed the user tony!
    - Click back
    - Click browse
        - You can see the list of posts of all the users that you follow (you can currently see 1 post since the user you follow has created 1 post)
        - If there were more posts created by users that you follow, clicking next will allow you to look through those posts.
        - Click add to cart to add the item that you see to your cart
        - Click back after you get the message that the item was added to your cart
        - Click back
    - Click on cart
        - You can see that a product has been added to your cart

        - You can click on buy cart
        - Click back to go to main
        - Click on cart again and you will see that the cart is now empty
    - Click search
        - Search for the category that tony's post was in
        - Click search
        - Enter the index of the item (it is the number before the ")" character) in the text box and click buy something
        - If the product is still in stock (i.e. you made the quantity large enough), then you can buy the product
        - Click back.
    - Even if you exit the program and rerun it, you can still sign in, and the previous posts will still exist.

Updated Specifications for Phase 2:

The goal of creating the app Bazaar this semester was to have an app where users can connect with other people, buy and sell products, and overall have an efficient shopping experience online. The structure of our 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 products of interest using a tag system. Users are also able to follow other users, and can browse a feed. This feed contains the posts that have been made by the users that a user follows. Users are able to buy the products inside the posts, make comments, add likes, etc. for the post. Currently, other users cannot leave comments or likes, but that is a feature that we will hopefully implement in the future. Users can decide to buy products advertised in their feed by adding products to their shopping cart. They can also choose to search up products and get posts from users that they don't follow.

New Design Decisions:

- **Saving entities only once in an .ser file**: In phase 2, we made sure to save product, post, and user entities in only one of the databases to avoid having multiple databases storing duplicate entities. For example, if we wanted to save a product, we save it in the IdToProduct.ser file only, and then make sure to save the String ID of the product everywhere else instead of the product entity again. This way, if we wanted to make a change to the product and save it back to the database, we only needed to update the product entity in only one of the .ser files. This ensures that we never accidently leave an outdated entity in another file.

- **GUI implementation:** In phase 2, we switched from a command line interface to a GUI using JAVA swing. We were able to keep much of the code in our controllers, use cases, presenters, and entities largely unchanged. Hence, not much change was needed in the back end which gave us a good indication that our code followed clean architecture.

- **Browse, search, buy, follow other users feature**: We worked on simplifying the creation of the feed to only occur when the user asks for it. We also allow users to search for products inside the database(we continued developing this feature from phase 1) . Users are now also able to follow other users, and generate a feed based on the posts made by the users they follow. The users are also able to add products either from their feed or search into their cart, and purchase their cart.
    - **Following another user**: Users now have a list attribute containing the usernames of other users that they follow. 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

    - **Browse**: 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 is done in search.

    - **Buy:** when the user empties/"buys" their cart, the quantity of the product decreases for other users. We decided that products that are out of stock will not be taken out of the database. This is because products that are out of stock are still often displayed to users on websites such as Amazon, where creators of those products can edit them to increase their quantity without having to create a new post each time. Editing is a feature that will be implemented in the future.

- **Phase 2 testing**: We have added tests for the new features. Many of our tests carried through from phase 1 to phase 2.

- **Factory design pattern, Command design pattern, and scratching out Memento**: Factory was used to allow for better design of the GUI. Command design pattern was used to handle commands from the buttons in the GUI (explanation included below in the doc). The removal of Memento is explained in more detail in the major decisions section of this doc, but we decided to not use memento because the user could now easily input their post creation choices in a textbox and move through various user features using buttons from the GUI, rendering Memento redundant.

- **User requires password to sign in**: In phase 2, we require the user to input a password that is saved along with their username in a .ser file to allow for more secure access to their profile.

- **Fixed code smells:** Mainly for many backend classes that had very long if-else statements in the old command line implementation. We fixed those long methods when we implemented the GUI, since we had to separate the methods inside the controllers to smaller methods that could be called by the GUI.

The content of the database .ser files:

- IdToProduct.ser
  - A hashmap
    - Keys: ids of products
    - Values: Product object associated with the ID
- product.ser
  - A hashmap
    - Keys: categories of products
    - Values: list of product ID's in that category
- user.ser
  - A hashmap
    - Keys: usernames of users
    - Values: user entity associated with the username
- username_password.ser
  - A hashmap
    - Keys: username of users
    - Values: password of users

Clean Architecture:

Example Scenario Walkthrough (Making a post):

- When the user puts in information about the post, the post GUI takes in information from the user. Then createPostController is called from one of the classes in the GUI layer. We use the createPost method inside createPostController. Inside of the createPost method, we create an instance of createProductController. The createProduct method inside createProductController is used to create the product entity. (We will not go into details about how the product is created here because it is its own scenario). Then, we make an instance of the saveUserGateway as a saveUserGatewayInterface and pass it into the PostManager use case.(Notice use of dependency Injection here). Then to create the new post we use the createPostWithRateComment method to create the post entity. Then we use the savePost method in PostManager use case to save the new post. Notice that we have used dependency injection so that savePost method uses the SaveUserGateway interface injected to save the post.

Enterprise Business Rules (Entities):

- User: how we store a user and all their information associated with their profile (username, password, posts, etc). There are many functions outside the user that interact with it. User is heavily linked with Product and Post since they are the three pillars of program. When the product is created,

- Product: how we stored the actual products that were to be sold by our users (name, size, price, etc.). Product is closely linked with post but the distinction is necessary because the product interacts with Post and outside users can only interact with the post.

- Post: the container for our products that other users can see and interact with (like, comment, rate, buy, etc.). Post is the bridge between a user and a product they wish to buy, post displays relevant information like price and size while also allowing users to do things like add ratings and comments, which are outside of the scope of Product and its functions.

Application Business Rules (Use Cases):

When creating our Use Case classes, we kept clean architecture in mind as we implemented them, they only call on methods from our entity classes, or use dependency injection design pattern to not violate clean architecture. An example from our code of a Use Case class is the BrowseUseCase Class in the options.browse package, this code takes in a hashmap of username to user entities, and a list of usernames representing following, and it generates a list of posts by accessing methods in the user class since it is a use case, instead of this operation being performed in the BrowseController, which cant access user methods without violating clean architecture.

Interface Adapters (Gateways, Controllers and Presenters):

- Gateways read from the .ser files (database) and pass the information to use cases by calling the use case methods. If use case classes required to call methods from interface adapters, dependency injection was used to call methods from the interface of the gateways. Controllers take in user input (through click of buttons, input into textboxes, etc.) and pass that information to use cases to manipulate.
- Presenters implement interfaces (to allow for future extension to other languages)
    - Presenters contain methods that return strings to be used by GUI classes
- Controllers example: The CreatePostController is called from the CreatePostCommand to take in information that the user has inputted about the post that they want to create, and relay that information to the use case classes to create the post.
- We ensured in our code all of the gateways, controllers are the only ones that manipulate use cases, and if necessary, ensured that use cases only read the interfaces of classes in that layer.

Frameworks and Drivers (GUI and Database(.ser file))

- Java swing was used in the implementation of our GUI, it took in actions from the user (pressing a certain button or filling in a text field). We also have .ser files which are our de facto databases, they are used to store information about products, users, and posts. Both of these are in the outer layer of clean architecture.
- The database (.ser file)
    - The classes that implement gateway interfaces are the only ones that access the .ser files, these are only called by use case classes. The higher level use case classes are in the application business rules layer but are allowed to access gateways by using gateway interfaces that the gateways implement. Since this adheres to dependency inversion, clean architecture is not violated.
    - Example: When we want to save a product to product.ser and IdToProduct.ser, we have SaveProductGateway implement SaveProductGatewayInterface, the ProductUseCase then uses the SaveProductGatewayInterface as an attribute to help it access the .ser files to save product information while adhering to clean architecture.
- The GUI
    - The GUI classes interact with the classes in the layer above such as controllers and presenters. The GUI classes use presenters to help present to users English text through buttons and labels to help guide users on how to use the GUI. The GUI also calls on controllers to help facilitate the flow of information from the user's inputs to the rest of the program to be worked on and saved.
    - An example of this includes the GUI for making a post, when a user enters information into various text fields, there are many instructions written in English which is a result of the GUI interacting with the EnglishPostPresenter. A user is also prevented from leaving any text field blank or putting in nonsensical information (putting an letter as a quantity or price), this is checked by passing information to the PostInformationController which checks for all of this.

SOLID Principles

Single Responsibility: Every class in a computer program should have responsibility over a single part of that program's functionality.

- Every class in our program has only one purpose, this is demonstrated by the sheer number of classes. The implementation of over 50+ classes was by design because our program has many different functionalities (create and store user, create product, create post, browse, search, etc), and so we had to make many different classes to implement the wide variety of functionalities.
- Example: When accessing or saving user information, one of our design decisions was to use 4 different gateways which are GetIDandPasswordsGateway, GetUserGateway, SaveUserGateway, SetIDandPasswordsGateway, which are used to get user id and passwords from the user, get the user itself, save the user and their updated information to the .ser file, and set a new user in our system. At first, we wanted one gateway to

handle all the interactions with the user and the .ser files but switched to 4 after realizing that we were not adhering to single responsibility. If we used a single gateway, it would be used by many classes for very different purposes each time.
- Single responsibility principle is also evident in the fact that our classes are small and usually contain one method with a single purpose in the program.

Open Closed: Classes should be open for extension but closed for modification.
A few examples of how we adhere to the open closed principle throughout our code:
- By implementing the factory design pattern when creating our GUI frames in swing. This allows for easy addition to our user interface, without changing what the original user interface does.
- We use the command design pattern to help implement the ActionListener when pressing buttons on the GUI. This reduces the use of if-else statement blocks that check which button the user presses and reduces the need to add another else-if if we decide to add new buttons in the future.
- We also adhere to the open closed principle through implementing interfaces for the presenters. This allows the extension to multiple languages that our code is able to present to the user.
- We also adhere to this principle by using builder design patterns to help us to create posts. Using builders allows us to easily add new features to posts in the future without breaking the entire program when we make a change and forcing us to refactor everything. The reason builder allows for the program being open for extension is that in the future, different kinds of post builder can be introduced to make the posts in different ways, add different features to the post without changing the already existing code.

Liskov Substitution: Objects of a superclass shall be replaceable with objects of its subclass without breaking the application.
- Our program does not make use of subclasses and superclasses relations, so naturally our program adhered to this principle.

Interface Segregation: Clients should not be forced to implement interfaces or methods they do not use.
- Throughout our program we use many different interfaces and for all of them, every method is used.
- Many interfaces used in our program, such as GetProductGatewayInterface, SaveUserGatewayInterface, SignInGatewayInterface and many others only include one method each. They are kept this way because they are meant to enforce a very specific role which is important to the overall program. All the interfaces are vital in enforcing the necessary roles for maintaining the program and so all the methods are used. Some methods in interfaces include saveUser, which is vital towards saving a user to a .ser file, getUser which allows us to find a User who is in the .ser and many other interfaces and methods whose usage is crucial for allowing our program to function.
  Another example where interface segregation is used is the GUIFactoryInterface and the classes that implement it. All the GUIMakers implement this interface and therefore

implement the “createGUI” method which is in the interface. Note that the interface does not have other methods that are more specific since not all the GUIMakers might need to implement this method.

Dependency Inversion: High level modules should not depend on low-level modules. Both should depend on abstraction.

- Throughout our code, we use gateways many times to access .ser files. These gateways need to save entity objects such as User, Product, and Post to the .ser files, so what we did was we used abstraction to prevent dependency. We created interfaces which the gateways implemented and created interface instances which the higher level modules such as use cases could then use to create and store entities inside the .ser file without having to directly depend on the implementations of each gateway. What we did was dependency injection, by working through interfaces (abstraction) we did not have to directly import from gateways.
- Example: In ProductUseCase, there is an instance of SaveProductGatewayInterface which SaveProductGateway uses, this allowed us to use a method saveNewProductToSer in the use case to call on the interface instance to save to the .ser file.

Design Patterns:

Factory design pattern:

We explain below in the “Major Design Decisions” section in depth as to why we implemented this design pattern. In summary, we use this design pattern to create different types of GUI windows in our program.

This is an explanation with a specific example of how it is implemented in our code:

First, we have the GUIFactoryInterface inside the gui package that all the GUIMaker classes implement. For example, the WelcomePageGUIMaker implements the GUIFactoryInterface and implements its method called “createGUI”. We also have the “GUIFactory” inside the gui package that returns a GUIFactoryInterface type based on a string that is passed in. For example, we pass in the string “WELCOME” to create the Welcome Page of our program. Therefore, to create a Welcome Page in our program, we first instantiate a GUIFactory and we use the “getFrame” method (pass in the “WELCOME” string) to get the GUIFactoryInterface, which here would be the “WelcomePageGUIMaker”. Then, we call the “createGUI” method on the GUIFactoryInterface, which runs the “createGUI” method inside the “WelcomePageGUIMaker”. (since the “WelcomePageGUIMaker implements the GUIFactoryInterface)

All the GUIMaker classes use the same design as described above.

Command Design pattern:

We explain below in the “Major Design Decisions” section in depth as to why we implemented this design pattern. In summary, we use this design pattern to handle the action of each button.

This is an example of how it is implemented in our code:

First, we have the “ButtonCommandInterace” in the gui package that has the “apply” method that all command classes implement. The “ResetCommand” in the signin package inside of the login package implements this interface and implements the method. Then, the SignUpGUIMaker, inside the sign_up package that is in the login package, adds its “reset” button to a “commandmap” that is a hashmap with all the buttons mapping to commands. The reset button maps to the reset command. Then, when the button is clicked, the program gets the button from the hashmap and returns the command that is of type “ButtonActionInterface” and the “apply” method is called on the command. That way, no matter what command the button maps to, the “apply” method would work, since all command classes implement the method.

Builder Design Pattern:

- For post creation, the design pattern for creating the builder design pattern was user to build the complex object. This design pattern also allows us to be able to add different kinds of posts to the program, or add posts that have different features. This allows for our program to be more open to extension.
- Also, because of the use of abstraction in the builder design pattern, there can be different types of posts created in the future.
- To implement the builder design pattern, we have a PostDirector which decides which builder is called and calls the abstract builder methods.
    - The implemented builder class is responsible for creating the post and setting the attributes of the post according to the user’s input that was received from the controller.

Dependency Injection Pattern:

- Use cases such as ProductUseCase instantiated, and passed in instances of the Gateway interface and called their methods to save products to the database. Using this design pattern, use cases that called methods in the gateways did not depend on concrete implemented gateway classes, but rather only depended on the gateway interfaces and did not know anything about the implementation of those gateways. Hence, the use case classes did not depend on the gateway implemented classes, but only depended on the interfaces.
- Moreover, having presenter interfaces allowed us to call the methods for these abstractions instead of relying on concrete classes, which enhanced our use of the dependency injection design pattern.

Major Design decisions:

- **Factory Method design pattern (why we implemented it):**

In the creation of different “windows” for the GUI, we use the factory method design pattern. The main reason why we used this design pattern in this situation is to adhere to the SOLID principles, more specifically the single responsibility principle and the Open/Close principle. We now have the GUIMaker classes each creating their GUI “window” instead of the GUI classes creating the GUI window and doing other jobs such as handling all the clicks to buttons (The command Design Pattern coupled with this also helps here). We can now also create multiple types of “windows”, for example the “WelcomePageGUIMaker” or the “OptionsGUIMaker” by directly calling the GUIFactory. Also, the code is closed for modification since any modification of each GUIMaker would be done inside each of the GUIMaker classes. Finally, the code is open for extension since we could easily add a new GUI “window” that would simply implement the GUIFactoryInterface.

- **Command Design pattern (Why we implemented it):**

To handle the action of each button, the command design pattern is used. It is used since multiple buttons being clicked on different GUI windows amounts to the same code execution. This design pattern allows for reuse of code since it decouples the object that invokes the operation(in our case being the different GUIMakers) from the one that knows how to perform it. (in our case, the individual commands). Additionally, having multiple if-else statements for each button in the GUIMaker classes did not follow the Open/Close Principle since our code would not be closed to modifications and therefore using this design pattern helped us solve this issue. (for example when we want to add a new button or a new command). Lastly, these design patterns helped us to get rid of very long methods in our code (get rid of code smells) when we decoupled the commands to their own classes.

Therefore, using this design pattern ensures that our code adheres to the Single Responsibility principle since each command execution is in it’s own class. (For example, “BackWelcomePageCommand” in the login package only has code that applies to going back to the Welcome Page).

Then, this design pattern ensures that the classes that call the Command classes are closed for modification since any modification needed would be done in the individual command classes. Also, the code is open for extension since any new command that needs to be included would just be added as a class that extends the “ButtonCommandInterface” in the gui package and implements the “apply” method. We should also point out that the Interface Segregation Principle is followed since all command classes implement the “ButtonCommandInterface” and implement the “apply” method. (no unnecessary method that the command classes need to implement)

- **Builder design pattern for Post:**
    - For the creation of post, the builder design pattern was used to create a complex post object through a director and post builder. This way, there can be different types of post builders developed in the future to implement the creation of different types of posts.
    - This allows the post creation to be open for extension and closed for modification.
- **Getting rid of the memento design pattern:** Since we are not using a command line anymore, the user can easily edit their product information, without having to type an undo command. Hence, we did not need the memento design pattern anymore.
- **Using .ser file for our database rather than other databases**:
    - This database allows for good information hiding and saves the data after the user signs out of the program, or closes the program.
    - The information saved is not understandable by human hackers, and so our user information is protected.
- **Show posts in browse, and products in search**: we decided that when a user searches, they are likely looking for a certain product, under a category rather than a post. Hence, when the user searches for a certain category of products, they will access a list of any product under that category. (although, they would not be able to see comments or likes, but since users cannot add comments at this stage, it is not a major issue)
    - If the user decides to browse, they want to look through posts. Hence, posts are shown for them.
- **Create the feed for the user when they ask for it, instead of saving the feed as an instance attribute**. Saving the feed as an instance attribute would have forced us to continuously update the feed of the user even when the user is not signed into the program. This would cause massive issues with the amount of space that would be taken by our database. Now, the feed is constructed with a for loop through all the users that this user follows, and constructing the feed only whenever the user asks for it.
- **What GUI to implement:**
    - Using Java spring was the best option for the GUI because it allowed us to create a user interface with buttons, scroll bars, text boxes, etc. that is extensible to make better in the future. Also, it allows us to easily use our knowledge of coding in Java.

<u>Use of Github Features:</u>

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 to easily divide up tasks. We also made use of the “view files” in pull requests so that we

could have another member of our group look over the new code to try to spot any errors in logic before committing to the main branch. 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.

- The pull requests were reviewed by at least one other student, if there were conflicts, in the pull request, they were resolved using the web editor, and then merged to main when there were no more conflicts
- Issues were created and assigned to student to ensure appropriate task division

Code Style and Documentation:

Generally in our code, there are no warnings in IntelliJ, as we tried to fix these issues as they arose.The only issues we currently have in our code are related to casting the Hashmap we generate with our serialized file writer, and issues with raw usage of swing containers. We used Javadocs throughout our code, documenting methods, classes and attributes where necessary. We tried to keep the descriptions simple and short without sacrificing any necessary information. This also made reviewing our pull requests much easier when a new method was implemented, as we knew exactly what it was supposed to represent from the Javadoc.

Code Organization:

Our packaging strategy is packaging by feature. Each feature is related to a specific feature we want the user to be able to do. At the top level, we have 4 packages: login, options, user, and product. Login contains the packages for each of the features related when a user logs in: the welcome page, signing up, and signing in, and all of the related classes are inside each. The options package contains the packages related to the features a user can access after signing in, such as browsing and cart. The product Package contains all of the classes relating to the creation, gateways, use cases about the product entity class, and the User package is organized in the same way. These were not included in the login or options package, as they do not have a corresponding gui that they implement, like the post entity.

Refactoring:

We used a lot of refactoring when we switched over from using a command line interface to the swing GUI, since we needed to change how many of the classes were called, and how they interacted with each other. A big use of refactoring was when we resolved the problem of having to catch two exceptions, IOException and InvalidClassException, all throughout our code from when we implemented serialization in Phase 1. These were refactored in pull request 221, where the exceptions were changed to be handled in the class DictionaryReadWriter, where they were first thrown, instead of having to use several try/catch statements throughout our code to try and catch them. Refactoring using Intellij was also helpful when changing our packaging structure, as it made it simple to rename and move files. We also refactored a lot of serialization code, to reduce duplicated code in many of the classes.

Testing:

One problem that arose when implementing the GUI using swing was that the classes that implement our GUI are hard to test directly. Instead of testing the GUI classes directly, we made sure that the GUIs had very low actual functionality (other than creating buttons, etc), and instead called methods from other classes that have actually been tested using Junit.

Additional Functionality :

- Interface for presenters to extend to different languages - improve accessibility
- GUI and working with buttons
- Searching for posts
- Following other users
- Browsing through feed
- Creating a post
- Password

Future Goals and Questions:

- Connect to a server so that multiple users can have access to our serialized database at the same time so that multiple users can sign in simultaneously and make posts that can be saved and be accessed by other users live.
    - Allow for chatting between users
    - Connect with services such as banks for exchanging money
    - Update the feed live using the observer design pattern so that when the user is still logged in, and another user that they follow create a post, their feed will be updated.
- Improve the GUI and add visual features
    - Allow users to add profile pictures, pictures for their posts, etc.
- Save a user's cart instead of deleting it when they log out
- Fix the bug that allows users to add and buy products to their cart even if the quantity is 0 while browsing
- Allow users to disable their account and delete their posts.

Progress Report:

- **Diego:**
    - Worked on cart functionality, ensuring items were able to be added to the cart, bought, as well as ensuring users would be able to return to home if the cart was empty or after the cart was bought.
    - Worked on Gui for cart functions
    - Integrated cart and cart gui to the rest of the program

- Updated code from phase 1 to make it more compatible with gui and phase 2 modifications
- Helped ensure tests were acting on phase 2 files instead of phase 1 files
- Implemented buy functions that were compatible with the cart's input and output of data
- Testing for Post and files directly associated with Post class
- Added JavaDocs to many methods that were fully functional but lacking JavaDocs
- Fixed bugs in several packages

- **Albert:**
  - Worked on GUI for the welcome page, the sign in page, the sign up page, the options page with Swing
  - Helped to come up with and implement design decisions of using design patterns such as Factory and Command to adhere to SOLID principles such as open closed principle and single responsibility principle with making GUI and pressing buttons in action listener
  - Worked on many presenter classes and presenter interfaces to ensure that English and other languages can be compatible with the program
  - Worked on post gui and the subsequent create and share post functions
  - Tested classes in the product package
  - Fixed bugs throughout the code, especially in the various GUI action listener methods such as fixing the ability to create posts or products with nonsensical inputs (letter as a price) or when not all the fields are filled in.
  - Design Document: SOLID principles

- **Umayrah**
  - Implemented the CommandDesignPattern for the Buttons in the GUI
  - Implemented the Factory Method design pattern for the creation of the different "pages" for the GUI
  - Implemented presenters multiple packages such as login and Post package
  - Implemented the GUI for creating a post feature (creating post and sharing)
  - Helped with fixing the packaging system by feature
  - Fixed multiple bugs throughout the code (example: multiple null pointer exceptions in the GUI)
  - Design document: Factory Method Design Pattern and Command Design Pattern

- **Sam**
  - Implemented GUI for Browsing and Following
  - Fixed problem with generating a UserFeed, and how it serializes and deserializes data too much, and generates it on the spot,instead of storing it with the user.
  - Helped with implementation of the Cart feature,

- Fixed updating the quantity throughout the data. This meant refactoring Posts to store the ID of the product they represent, instead of an actual product attribute
- Refactoring serialization to have less duplication
- Created tests for cart,buy and read_write packages
- Fixed the packaging system to be more intuitive, and be less messy
- Fixed bugs with following multiple users, cart not updating.
- Added multiple presenters, and updated existing presenters.
- Fixed numerous intellij warnings
- Design document: UseCases, open/closed principle, testing, github features, refactoring, Code style/documentation, Code organization, two paragraphs in accessibility doc.
- Fixed Problems with the way Exceptions where being handled by our serialization functions

- **Jacqueline**
  - Worked on SearchGUI, SearchPresenter, SearchPresenterInterface
  - Fixed import statements and errors in files when transferring files over to a new main folder
  - Adjusted naming, organization of test files/packages to reflect new changes
  - Adjusted existing tests, added new tests for new classes
  - Edited and proofread design document to make writing more clear
  - Worked on the accessibility questions
  - Fixed many bugs: ex: duplicate products appearing in search, null pointer exception being thrown when browsing/searching
  - Helped resolve merge conflicts

- **Yasmin:**
  - Worked on the search option: GUI (scroll and buttons), controller, use case, etc.
  - Helped with and worked on follow users option: GUI (scroll and buttons), controller, use case, etc.
  - Buy feature for search: allow users to add index of items of interest to their cart
  - Login feature: helped connected database so that user information is saved
  - Worked on builder design pattern for post
  - Design document: Specification, instruction of use, and and future goals

Each group member should include a link to a significant pull request (or two if you can't pick just one) that they made throughout the term. Include a sentence or two explaining why you think this demonstrates a significant contribution to the team.

Pull Request:

- **Albert**

1. https://github.com/CSC207-UofT/course-project-csc207gang-amazondevelopers/pull/162/files

   This pull request includes the implementation of the welcomePageGui, SignInGui, SignUpGui and also the OptionsGui using Java Swing. This pull request is significant because these were the very first gui's that we implemented in our project and was the stepping stone for us in developing the gui for our entire program as they acted as a framework for how later gui's would be modeled after. Many group members used this pull request as inspiration to implement gui's on parts that they were working on.

- **Sam**
  1. https://github.com/CSC207-UofT/course-project-csc207gang-amazondevelopers/pull/184 This pull request is significant, as it was the first implementation of the Browse GUI to the code, which allows users to cycle through posts in the feed, it was not the final implementation I implemented but it started the foundation to the browse option of our GUI
  2. https://github.com/CSC207-UofT/course-project-csc207gang-amazondevelopers/pull/168 This pull request was from before our GUI was implemented, but it was the pull request where I fixed the implementation of BrowseUseCase to generate feed as soon as its called, instead of serializing each users feed attribute when a post is created, also has numerous small fixes to our code that needed to be implemented.

- **Jacqueline**
  1. https://github.com/CSC207-UofT/course-project-csc207gang-amazondevelopers/pull/69
     While this class was not needed in phase 2, this modified version of the command line used for testing classes allowed me to write up a majority of the tests for the classes in phase 1 that handled some of the most important features of our program, such as searching, logging in, buying, etc since many of them took in user input from the command line. As a result, for phase 1 we had a reliable way to test our features automatically instead of manually running our program and inputting different arguments. Many of these tests also served as a baseline for tests in phase 2, as most of them only needed slight adjustments when we went from the command line to the GUI.

- **Diego**
  1. https://github.com/CSC207-UofT/course-project-csc207gang-amazondevelopers/pull/181

This pull request is a significant contribution because it implements the entire cart function as well as the majority of the GUI associated with cart. Cart was repurposed for the GUI, the Cart window was implemented, and it was tied together seamlessly to the rest of the program.

- **Umayrah:**
    1. https://github.com/CSC207-UofT/course-project-csc207gang-amazondevelopers/pull/238
       This pull request was put in right after I had finished implementing the Command Design Pattern and the Factory Method Design Pattern for all packages except the post package. I believe this is a significant contribution because I have implemented design patterns that were crucial to our code following the SOLID principles (as we talk about in above in the progress report)
    2. https://github.com/CSC207-UofT/course-project-csc207gang-amazondevelopers/pull/172
       This pull request was put in after I implemented the GUI for creating a post. I believe this is significant in our program because one of the main features of our program is creating a post and all the other features such as search and browse are dependent on this feature working correctly.

- **Yasmin**
    1. Searchcontroller by Yasamin-Nouri Helyani · Pull Request #192 · CSC207-UofT/course-project-csc207gang-amazondevelopers (github.com)
       This is the pull request where I implemented the search option, including the GUI (buttons, scroll bar, etc.), connection to gateways, and controllers. This is an important feature of our program where the user can search through products that they search for from posted products by other users (or themself). This feature also allows users to add products to their cart.

    2. can now connect to the gateway from the GUI for login by Yasamin-Nourijelyani · Pull Request #164 · CSC207-UofT/course-project-csc207gang-amazondevelopers (github.com)
       This pull request was when we were first starting with the login GUI and wanted to connect it to the backend. I think it is significant because this code helped us connect the controllers, and gateways to connect to the GUI frontend. This also allowed us to read from our database and get information from the user and pass it to the back end.

Accessibility report:

1. For each Principle of Universal Design, write 2-5 sentences or point form notes explaining which features your program adhere to that principle. If you do not have any such features you can either:

   (a) Describe features that you could implement in the future that would adhere to principle or

   (b) Explain why the principle does not apply to a program like yours.
   https://universaldesign.ie/What-is-Universal-Design/The-7-Principles/#p1

   Principle 1: Equitable Use

   - Every user upon signing up creates a password for their account that gets encrypted.
   - The current user interface, however, is still very simple and not very appealing. To fix this we could expand our GUI to include images, different language options, fonts, and colors.

   Principle 2: Flexibility in Use

   - The program does not allow for flexible use at the moment as the program can only accommodate for those that are able to read. There is no audio option for the visually impaired or those that would just rather listen than read. To resolve this, we could implement a voice-over system.
   - Currently, you can only access the program on a computer. In the future we could implement our program to run on various other devices, especially on phones.

   Principle 3: Simple and Intuitive Use

   - Program uses simple, short prompts for the user, buttons labeled with short messages
   - Print out an error message every time a user is blocked from doing something that explains why a user cannot do something/what went wrong. For example, when the user leaves a textbox empty while signing in/signing up we tell the user that they need to fill in all fields.

   Principle 4: Perceptible Information

- Print out messages in green if the user a successfully completed a task (such as successfully creating a post), and in red when something has gone wrong
- Placed buttons and
- Unfortunately, information in the GUI lacks contrast - grey background for all menus with white text boxes and buttons, and most of the information is communicated through text. To fix this, we would need to adjust the size of each menu to be larger, adjust the size and font of the text for readability, add images, and much more.

Principle 5: Tolerance for Error

- In our product creation menu, we used a few drop down menus instead of textboxes to try and minimize sources for spelling errors. We could improve this further by allowing users to edit posts they make after they create them, in case they make a mistake in inputting the text information for the post.
- Allow users to go back and forth between options so they can return to wherever they were if they accidently click the wrong option
- Need to reformat placement of buttons so that buttons such as “clear search bar” and “back” are placed further away from buttons such as “search”, allow users to delete items from their cart, save drafts of posts they can return to later, provide warning message that allow the user to double check whether they want to share a post or buy their cart.

Principle 6: Low Physical Effort

- Allows users to remain in a neutral position as it is a casual social media site, but there is some repetitiveness as the program requires typing most of the time. However, there isn't much that can be changed to reduce this repetition.
- We have an empty cart option that empties out the users entire cart, so user does not have to manually click buy product each time

Principle 7: Size and Space for Approach and Use

- This principle does not apply to our program, as it is an online social media site.

2. Write a paragraph about who you would market your program towards, if you were to sell or license your program to customers. This could be a specific

category such as "students" or more vague, such as "people who like games". Try to give a bit more detail along with the category.

- We would definitely market our program to a younger audience, and especially those who are interested in selling used/homemade items. We modeled our program after the likes of Instagram, which is very popular among a younger audience, by taking inspiration from its use of followers, hashtags, and a feed to create an app similar to Ebay but with Instagrams much more relevant features.
- We could also look at expanding to sell corporate licenses to large companies that would like to use the platform to market their products.

3. Write a paragraph about whether or not your program is less likely to be used by certain demographics. For example, a program that converts txt files to files that can be printed by a braille printer are less likely to be used by people who do not read braille
    - Our program is less likely to be used by people who do not like online shopping, as the core of our program is based around buying and selling products through the program.