SOFTENG 306: SOFTWARE ENGINEERING DESIGN 2

Sneaker Hub

Quality Report

TEAM 21

Jaskaran Sandhu, Harry Qu, Ibrahim Anees

CONTRIBUTIONS

Developer 1: Jaskaran Sandhu

- Inconsistencies with regards to SOLID principles
- Inconsistencies with the design document – New additions

Developer 2: Harry Qu

- SOLID principles
- Inconsistencies with the design document – Changes in project schedule

Developer 3: Ibrahim Anees

- Good coding practices
- Inconsistencies with the design document – Changes in UI

CONTENTS

1. INTRODUCTION

Sneaker Hub is an android app designed to help users view, save, and compare their favourite sneakers. Our application provides users with four categories of brands (Air Jordan, Nike, Adidas, and Vans) to choose from. Other features include a 'Top Rated' section, a favourites section, a rating system, and an advanced filter search.

This quality report aims to outline the final realisation of SOLID principles, inconsistencies within these principles, inconsistencies with the original design document, and good coding practices followed by the Sneaker Hub development team.

2. SOLID PRINCIPLES

The final architecture of Sneaker Hub is mostly consistent with what we have planned in the Design Doc. The use of MVVM design pattern and CLEAN architecture made sure our code followed the SOLID principles and minimised multiple design smells to improve the maintainability, reusability, and coherency of the project.

2a Single responsibility principle

Because of separation of layers, the classes within each layer have only one reason to change. For example, the Repository Layer is responsible for all the CRUD operations with the database, and hence the ProductRepository and CategoryRepository classes are exclusively responsible for the Product and Category CRUD operations respectively. Single Responsibility Principle is also demonstrated by each method within each class, for example getProductById method in the ProductRepository class, as its name suggests, gets a product from the database by its Id value. Doing so minimises design smells such as rigidity and fragility. As any changes to the database schema now would not affect any other classes in other layers, only changes in the Repository layer would be needed. This ensures reusability, because different parts that need to interact with the database can simply use methods from the corresponding repository class without worrying about internal details and repetitively writing the same code to do the same operation in different parts of the project.

2b Open close principle

Open Close Principle is accounted for within the design through providing interfaces for classes whose implementation is likely subject to change. For example, the ProductRepository implements the IProductRepository which declares all the methods needed for CRUD operations on the products stored in the database. All implementing classes of this interface (ProductRepository in this case) hence are required to implement all these methods. If the logic of interacting with the database should be changed because the database has been changed, only a new class implementing the IProductRepository interface needs to be created. Any other parts of the project interact with the interface IProductRepository and does not care about any implementation details of the implementation class. This enhances the project further in maintainability and reusability by minimising rigidity, rigidity and useless repetition design smells.

2c Liskov substitution principle

The Liskov Substitution Principle is implemented in the model layer by creating abstract super class to hold any common fields and methods among the children classes. For example, the Entity abstract class is the parent class of all model classes (Fig. 1), and it holds the Id and Name fields and their getters which are common for all the model classes. The Category subclass holds the common fields and methods for only the category classes and has no overlap with the Product Model class. Anywhere a certain concrete model class is needed, that model class can be instantiated without violating any of the policies and methods of the super class.

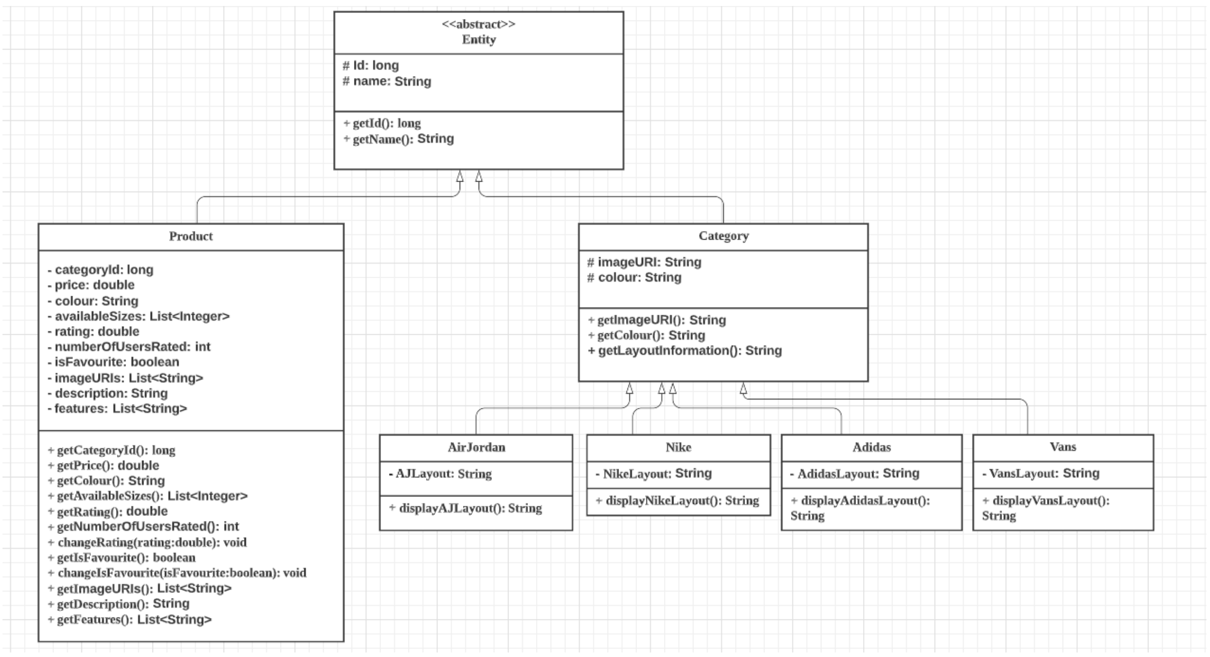


Fig. 1. Entity abstract class

2d Interface segregation principle

Interface Segregation Principle:

Interface Segregation Principle states that classes should not depend on other classes, whose methods it doesn’t require. In our architecture, the Use Case layer is added to ensure that the ViewModel layer classes do not directly depend on the Repository layer classes. Doing so, the ViewModel layer classes are not exposed to all methods (including methods they do not use) in the Repository layer classes while still able to access any needed operations within the Repository layer. Each class in the Use case layer fulfils a specific function, and it interacts with the Repository layer for its purpose suggested by its name, hence it is much easier to maintain compared to the more complex ViewModel layer classes which have multiple methods.

Classes within each layer interact with classes in other layers through their interfaces, unless it is certain that the class needed is fixed and very unlikely to change. For example, the SearchProductWithFilter use case interacts with the interface IProductRepository to fetch all products and filter them. This ensures Dependency Injection and if there is a new implementation class for IProductRepository that we wish to use, we simply have to register the new class as the class we actually want using whenever we use IProductRepository operations. Furthermore, we have 4 different categories (list view) with slightly different layout from each other, but they all utilise one single adapter, implementing dependency injection.

3. Inconsistencies with regards to SOLID principles

Each activity class and the corresponding ViewModel class in the architecture do not communicate with one another using an interface (Fig. 2). This violates the Open close and Dependency Inversion Principles, as both the activity and ViewModel classes are concrete classes. The reason why it isn’t viable to use an interface is because the ViewModel classes make use of the LiveData class. As a result, the activity class has to then use the “observe()” event listener object to access every public method from the ViewModel. This tight coupling between the ViewModel and activity classes exists by design and is essential for the ViewModel class to provide data persistence and state management across the entire lifecycle of the activity.

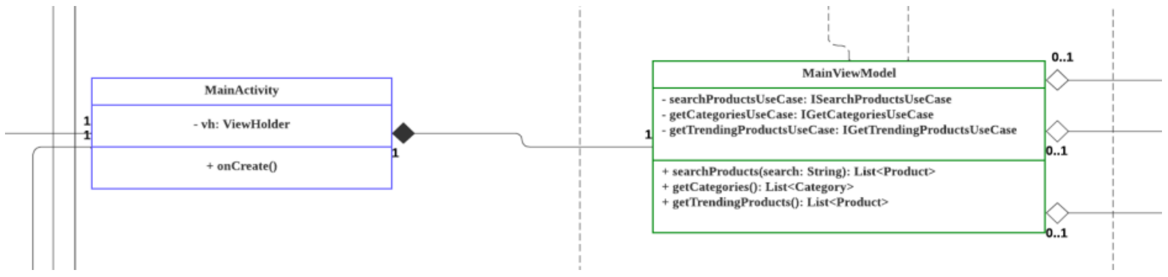

MainActivity
- vh: ViewHolder
+ onCreate()
1
1
1
1
MainViewModel
- searchProductsUseCase: ISearchProductsUseCase
- getCategoriesUseCase: IGetCategoriesUseCase
- getTrendingProductsUseCase: IGetTrendingProductsUseCase
+ searchProducts(search: String): List<Product>
+ getCategories(): List<Category>
+ getTrendingProducts(): List<Product>
0..1
0..1
0..1

Fig. 2. MainActivity and MainViewModel classes

There is a ViewModel class for each activity class within the architecture, as by design each activity class will require an implementation of its own ViewModel class. This means that the ViewModel classes are not modular, reusable components and the same ViewModel cannot be used for multiple activity classes. This further negates the need for having interfaces, as only the corresponding activity class will be communicating with the ViewModel.

The purpose of an interface is to provide abstraction and encapsulation, through hidings it’s inner workings/implementation details from the other classes making use of it. So that changes to the implementation don’t affect how other classes make use of it. However, in the activity and ViewModel scenario, the implementation of the ViewModel class depends specifically on the components of the activity class. Therefore, the addition or removal of components in the activity class require the implementation of the ViewModel to change accordingly. Which also defeats the purpose of having an interface, since the method signatures of the ViewModel are likely to change.

4. INCONSISTENCIES WITH THE DESIGN DOCUMENT

There were some inconsistencies with our initial design document. These included new additions to our final application, a change in the project schedule, and changes in the UI of the application.

4a New additions

It is virtually impossible to correctly account for all parts of the architecture with 100% accuracy before development has begun. Hence, there were a few changes to the classes within our working architecture, in comparison to the architecture shown within the class diagram.

"SelectProductsByColour" use case was not required in the implementation of the View Layer (activity, ViewModel classes) hence it became stale code and was removed. Instead, the "SearchProductByName" use case was created, as being able to retrieve the different colours of same sneaker was required by the View layer implementation.

We initially assumed that all the list views within the app would only require one adaptor. In actual, a second adaptor ended up being needed to implement the list view in the Favourites activity. This is because the unlike the Category List activity, which only needed to fetch all the sneakers for each brand and populate the list view. The Favourite activity, list view also needed to maintain and update the state (whether the isFavourite field is set to true or false) of each sneaker that was favourited by the user. This meant the implementation of the adaptor needed to be different from the adaptor used for the Category List View and hence a second adaptor was created.

Our initial class diagram also did not consider how the animations would be implemented and whether certain animations could require a class of their own. That is why our actual architecture required the creation of another class to implement the logic of the splash activity our application has on start up.

4b Changes in project schedule

Initially in our project schedule, we have planned to have all members work on the Frontend work of Sneaker Hub and then progress to the Backend work. Then after reconsideration, we separated the roles of each developer in one specialised area and hence able to progress on both the frontend and backend in parallel in the early stage of development. This helped us to discover any possible problems in both frontend and backend, and also have enough time to come up with solutions for these problems. For example, we have discovered the problem that database operations cannot be executed on the main UI thread, we then spent a lot of time researching and learning about how to resolve that problem. If we had continued with our initial project schedule, we may not have been able to finish everything we planned in the given time.

We also decided to populate the database with all the records we have, so that we can constantly test each part during development to ensure it works how we expected. However, this is not adaptable to any changes. If for example we wanted to add an additional field to the

Products Collection we have stored, we have to change all the code we had for populating the database to include this additional field. Hence instead, we only worked with a few sample records stored in the database at the early development stage for testing purposes, and only add all the products to the database once we are certain that our design is finalised and there will not be any additional fields.

4c Changes in UI

A few aspects of the final application's UI were changed from the mock-up design outlined in the design document. On the list view of each category, a new logo was added (Fig. 3) as per the lecturer's suggestion. This was done in order to fill the empty space.

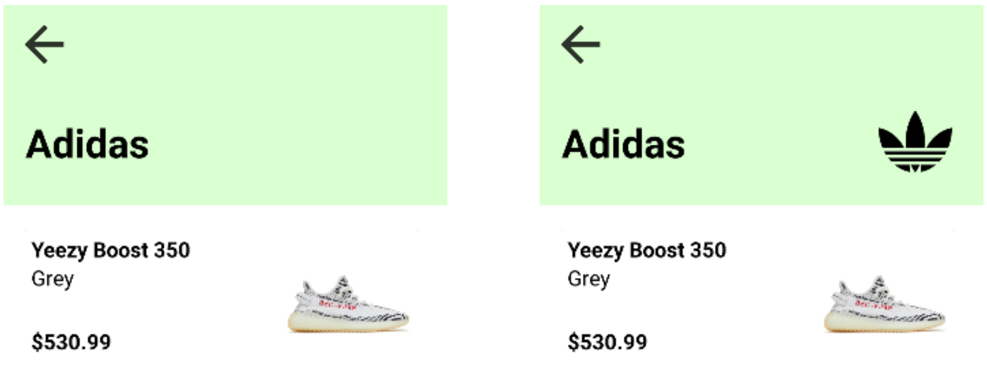


Fig. 3. Original design with no logo (left) vs revised design with logo (right)

On the main screen, the featured view was relabelled as 'Top rated' and made horizontal instead of vertical. This view was also placed above the category view, which was now vertical instead of horizontal (Fig. 4). This change was done to meet the requirement of having a horizontal featured view.

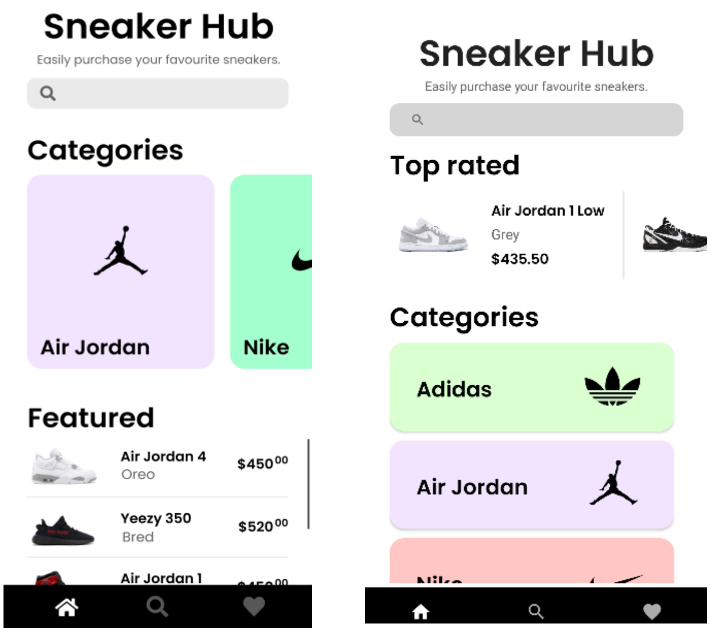


Fig. 4. Original main activity layout (left) vs revised main activity layout (right)

5. Good coding practices

Writing clean code is important as it enables developers to explain their code easily and clearly to other developers [1]. This is extremely important in our context of Sneaker Hub as there are three developers working on the application. It is inevitable that some parts of code will need to be understood and used by a developer who has not written it, so following good coding practices was vital for this project. Good coding practices that are used include naming conventions, clear and concise commenting, proper use of packages, and following good workflow and collaboration principles.

5a Consistent naming conventions

Throughout Sneaker Hub's project structure, consistent and understandable naming is used for each aspect of the application. This naming convention is used for package names, class names, XML names, image names, and variable names.

All our package names follow Java package naming conventions [2]. This is why all package names start with lowercase. Our package names also clearly and concisely describe each package's role. For example, the "repository" package contains all repository classes.

All class names follow Java conventions of Pascal Case [3]. Looking at our 'details' classes as an example (Fig. 5), it is evident that class names concisely and consistently describe the class. All interfaces also follow convention by starting with the capital letter "I".

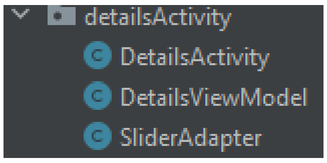

detailsActivity
DetailsActivity
DetailsViewModel
SliderAdapter

Fig. 5. Details classes

The image names for all our products follow a concise naming convention (Fig. 6). Each product starts with the letter "s", which stands for sneaker, followed by the product ID number. Preceding the product ID number (and separated by an underscore), is the view number of the image (ranging from 1 to 3). Thus, for example, an image named s65_2 would indicate an image of the second view of sneaker with product ID 65. This consistent naming convention enables easy storage and retrieval of image URIs from the database.

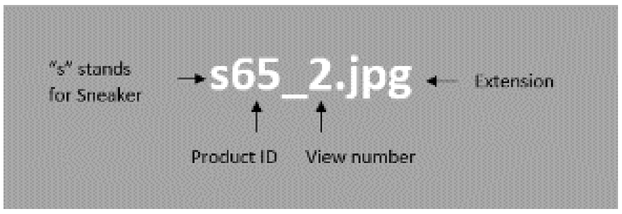
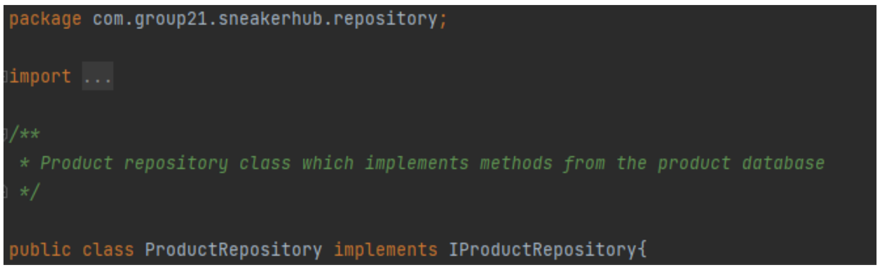


Fig. 6. Naming convention of product images

All variable names are sensible, concise, and indicative of functionality. For example, each Product object consists of a boolean variable "isFavourite." Conventions such as camel case are followed in accordance with Java variable naming conventions [4].

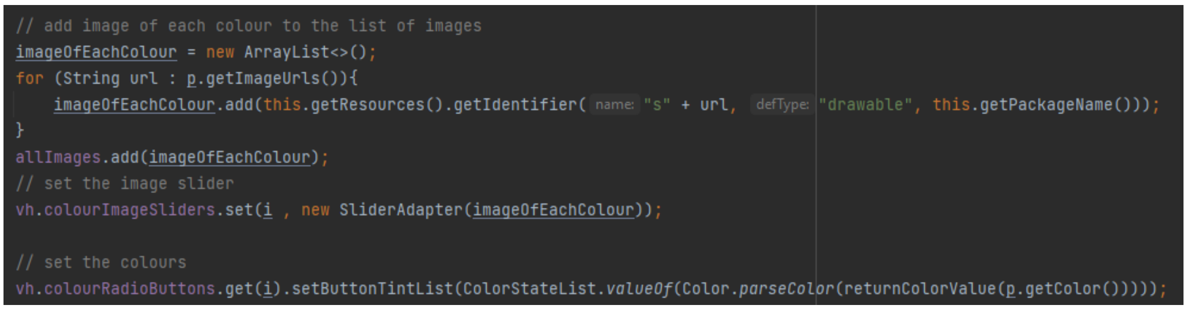
5b Commenting



```
package com.group21.sneakerhub.repository;

import ...

/**
 * Product repository class which implements methods from the product database
 */

public class ProductRepository implements IProductRepository{
```

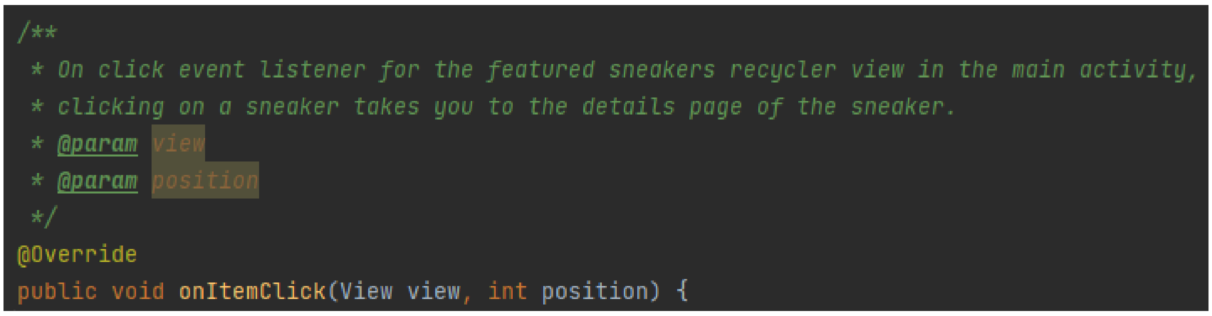
Fig. 7. Comment describing class

Each class has a comment on top describing the purpose and functionality of the class (Fig. 7). This is done so that developers are easily able to understand the purpose of a class without having to read through the code. Sensible comments are made throughout the code (Fig. 8) for each class explaining functionality, so that whoever reads the code is able to understand the functionality of each section of code. Comments are also only written when required, so that the classes do not become over-cluttered with comments. Important methods also follow Java commenting convention [5] of describing the method functionality, parameters, and return type in a comment body on top of the method (Fig. 9).



```
// add image of each colour to the list of images
imageOfEachColour = new ArrayList<>();
for (String url : p.getImageUrls()){
    imageOfEachColour.add(this.getResources().getIdentifier( name: "s" + url,  defType: "drawable", this.getPackageName()));
}
allImages.add(imageOfEachColour);
// set the image slider
vh.colourImageSliders.set(i , new SliderAdapter(imageOfEachColour));

// set the colours
vh.colourRadioButtons.get(i).setButtonTintList(ColorStateList.valueOf(Color.parseColor(returnColorValue(p.getColor()))));
```

Fig. 8. Comments throughout code

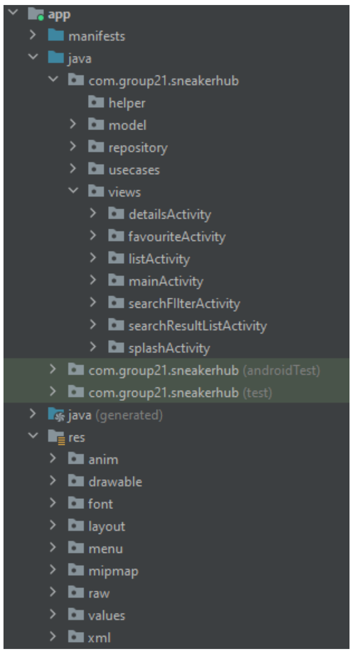


```
/**
 * On click event listener for the featured sneakers recycler view in the main activity,
 * clicking on a sneaker takes you to the details page of the sneaker.
 * @param view
 * @param position
 */
@Override
public void onItemClick(View view, int position) {
```

Fig. 9. Comment describing method signature

5c Use of packages

Packages are used to organise Sneaker Hub’s project structure (Fig. 10). The main package includes all sub-packages that relate to classes. These sub-packages include “model,” “repository,” “usecases,” and “views.” The “usecases” and “views” packages are further divided into several packages for each use case and each view. These inner packages contain the classes relating to each use case and each view. As per Android Studio conventions, the "resources" package holds several packages such as "drawable," which consists of all images used throughout the application. Likewise, the "layout" package consists of all layout XML files, "font" consists of all fonts used, "anim" consists of animation XML files, and "values" consists of XML files that hold specified values (e.g., colors.xml holds the colour value for each colour used in the colour palette). By organising Sneaker Hub's app into packages, we have been able to maintain an easy-to-understand package structure.



```
app
  manifests
  java
    com.group21.sneakerhub
      helper
      model
      repository
      usecases
      views
        detailsActivity
        favouriteActivity
        listActivity
        mainActivity
        searchFIlterActivity
        searchResultListActivity
        splashActivity
    com.group21.sneakerhub (androidTest)
    com.group21.sneakerhub (test)
  java (generated)
  res
    anim
    drawable
    font
    layout
    menu
    mipmap
    raw
    values
    xml
```

Fig. 10. Package structure

5d Workflow and collaboration

While using GitHub for version control, all developers worked on different features. Each feature was worked on from a different branch, which was branched off from the "development" branch. Once the "development" branch was in a stable condition, it was then merged to the "main" branch. This process ensured that no app-breaking code was in the main branch. For each merge, a pull request was created by one developer and required the approval of the other two developers to ensure that changes were reviewed for errors before merging.

Messenger and Discord ware used for team communication and collaboration, and tickets/tasks were delegated to each developer on Trello, this made it easy to keep track of the project's progress.

6. CONCLUSION

To conclude, the final version of Sneaker Hub is designed to better encapsulate SOLID principles including single responsibility principle, open close principle, and interface segregation principle. There are a few inconsistencies with regards to the SOLID principles, and to the design document. Finally, good coding practices such as naming conventions, commenting, use of packages, and proper workflow have been used throughout the process. Overall, developing Sneaker Hub has given the three developers invaluable experience of working with Android Studio and working within a team.

7. References

[1] M. Grey. “Why Should I Write Clean Code?” Echo Bind. https://echobind.com/post/why-should-i-write-clean-code#:~:text=Writing%20clean%20code%20is%20important,in%20the%20software%20development%20world./ (accessed Nov. 25, 2022).

[2] “Naming a Package.” Oracle. https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html/ (accessed Nov. 26, 2022).

[3] C. McKenzie. “Pascal case vs. camel case: What’s the difference?” The Server Side. https://www.theserverside.com/answer/Pascal-case-vs-camel-case-Whats-the-difference/ (accessed Nov. 26, 2022).

[4] “Variables.” Oracle. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html/ (accessed Nov. 27, 2022).

[5] “Commenting in Java.” Rice University. https://www.clear.rice.edu/comp310/JavaResources/comments.html#:~:text=By%20convention%2C%20in%20Java%2C%20documentation,begin%20with%20a%20%22*%22./ (accessed Nov. 28, 2022).