
# Quality Report

SOFTENG 306: Team 20

Amy Rimmer (arim402)

Christine Sun (csun325)

Samantha Mebius (smeb890)

# Table of Contents

# 1 Introduction

‘Sammy & Sun’ is a native Android application that showcases luxury designer bags and meets the requirements of the YouSee Soft competition. This report highlights the final realisation of SOLID principles and other good coding practices to ensure this application is of a high quality to be maintained and extended in the future. Also discussed are any inconsistencies between the final implementation and the initial design document planning.

# 2 SOLID Principles

## 2.1 Single Responsibility Principle

On a small scale, we are demonstrating the Single Responsibility Principle as the methods of our classes are only performing one function. For example, the updatePopularCount method in the details activity is only responsible for updating the count. Not only does this keep the method code tidy, but it also makes implementing these methods easier.

On a slightly larger scale, this principle is being implemented as the classes within each layer of our model either only represent one thing or are only responsible for a singular functionality. For example, the model classes only represent one ‘actor’ such as Tote or Category. As well as this, the repository classes have been separated into the logic responsible for a single collection in the database. Another example is the activity classes, each responsible for only one activity of the application.

This principle is also applied at an architectural level. Classes are grouped in packages of single responsibility such as activities, models or repositories. Any changes to these classes will all be done in the same package.

## 2.2 Open-Closed Principle

The Open-Closed Principle has been implemented by avoiding ‘hard coding’ functionalities in our project. For example, if more products or categories are added, the repositories, view models and adapters can handle these changes. Another example is if more photos are to be added to a product, the image slider and slider indicator can handle it without any modification to the code.

The use of interfaces allows us to maintain the Open-Closed Principle. Interfaces have been implemented to all entity, repository and view model classes. This ensures that classes are only dealing with the interfaces and not other classes directly. Future extensions of classes implementing the interfaces have no effect on associated objects using the interfaces.

## 2.3 Liskov Substitution Principle

Liskov’s Substitution Principle has been implemented in our design through the concrete Product superclass with Tote, Clutch and CrossBody as subclasses. These subclasses override the getCategoryID method with the same parameter and return to ensure the superclass can be called correctly.

Objects that interact with the IProduct interface can be instantiated by any of these subclasses without violating any of its methods. These classes will not notice the difference between the child class implementations of the interface. This also allows for more implementations of the product interface to be added in the future. For example, in the list recycler adapter where the super Product class is replaced with the specific subclasses for each category.

### 2.4 Interface Segregation Principle

The use of interfaces in our project ensures that Interface Segregation Principle has been met. For example, all the view model and repository classes have their own interface. Implementation of interfaces completely supports all methods of the interface and interaction between classes is done through these interfaces. This allowed us to work on layers separately in parallel as the classes did not directly interact with each other. Interfaces allow packages to be more independent from others with fewer dependencies. They are also easy to read by other developers to see what methods the class implements.

### 2.5 Dependency Inversion Principle

Our project implements a version of the Model-View-Viewmodel (MVVM) architecture where there is an encapsulation of lower layers from higher layers and each layer is only dependent on adjacent layers. Interfaces have been used as abstractions to allow for flexibility between different implementations and dependency injection. The higher classes depend on abstracted interfaces of the lower classes and not implementations themselves. This prevents high levels of dependency throughout the class structure. For example, the activity classes only depend on view model interfaces, view model classes only depend on repository interfaces and repository interfaces only use model interfaces.

## 3.0 Inconsistencies to Design Document

### 3.1 Class Diagram

#### Entity Layer

The product class is no longer abstract as there is no difference between the child entity classes.

#### View Model Layer

The initial plan to separate domain use case classes from the view model classes has not been implemented. Instead, we combined these layers into one view model layer in order to simplify the code structure and make it easier to implement. Due to the time pressure of this project, we realised these layers can be combined whilst still maintaining SOLID principles as much as possible.

### Adapters

In our plans, we had one item adapter that was going to be used by all recycler views. We have still met the requirements by having a single custom adapter to populate the listview activity. However, we have implemented additional adapters for the different layouts within our application. These include an adapter for category items and the horizontal panels on the main activity as well as an adapter for the search activity and the slider images on the details activity. Implementing these adapters was a ‘hacky’ solution with a lot of repetitive code amongst them. Given more time these adapters could inherit from a common abstract parent class adapter.

### View Holders

The view holders for our list adapter were implemented as planned. However, the view holders for popular, favourites and list are now a part of their own adapters as mentioned above.

## 3.2 Data Scheme

Due to the initial lack of understanding of how the data will be called by our application from the database, our data scheme plan has changed. Initial plans were to store the products separately under collections according to their category. However, instead we implemented a collection for all products as there is no difference in how the data is stored to simplify the retrieval. The favourites and popular collections are stored with documents referencing the IDs of the product instead of a singular array list as this is a more logical implementation.

As well as the products collection, we are also storing the products under their relevant collections. This solution is not ideal as duplicate data is being stored. However, this was initially done to be able to access the products quickly and easily before caching was implemented. With more time these category collections would be stored as references similar to the popular and favourite collections.

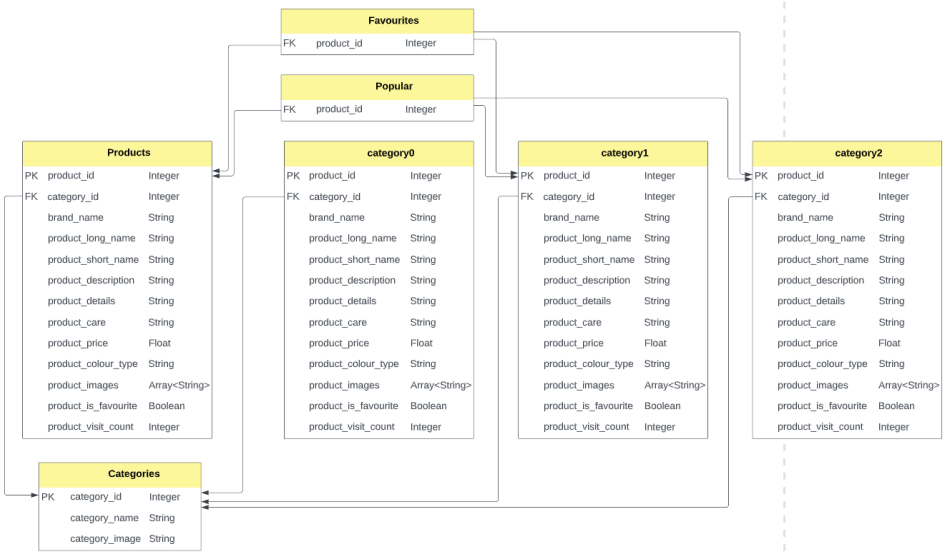


*Figure 1: Final Data Scheme Diagram*

## 3.3 UI Design

### Main Activity

The headings have been simplified to ‘Popular’ and ‘Favourites’ with smaller product cards underneath to allow more bags to be seen without needing to swipe. We have also removed the curved top corners of the main screen body as we realised that in our initial design this feature gave the impression that you can swipe down to reveal more content on the app bar layer but this is not the case.

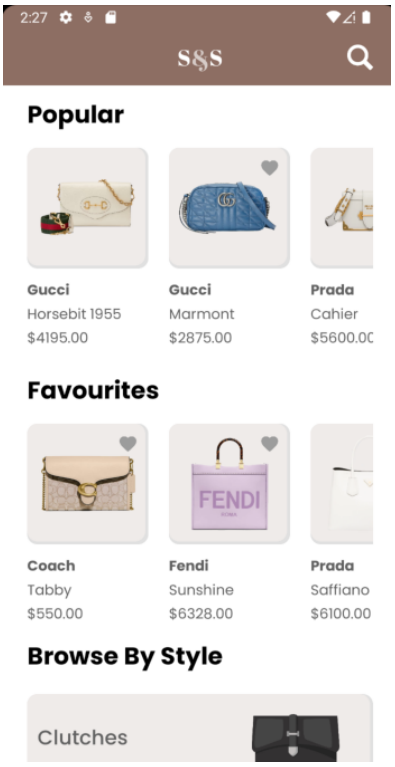


*Figure 2: Final Main Activity*

### App Bar

The button to go to the search activity has been moved to the app bar to allow it to be accessed from any screen and avoid taking up valuable space on the main activity. Also, the brand name in the app bar has been simplified to just ‘S&S’ instead of ‘Sammy&Sun’ to take up less space and give a more minimalistic, luxurious feel. Another change is that on all the app bars except on the main activity, the back arrow is now a home icon. During implementation, we realised that a back button is already implemented in android so this feature was not necessary.


*Figure 3: Final App Bar*

### List Activity

We initially had the item layout for the clutches list view as narrow cards in two columns. However, to maintain consistency amongst the list views this layout was changed. The

clutches are now displayed in cards the same width as the other two list views; only the photo is above the text. This design was intentional for the clutches category as the images are more horizontal, showcasing them in a better way.  This way we can meet the requirements by demonstrating different card layouts in each listview while also maintaining consistency throughout the application.

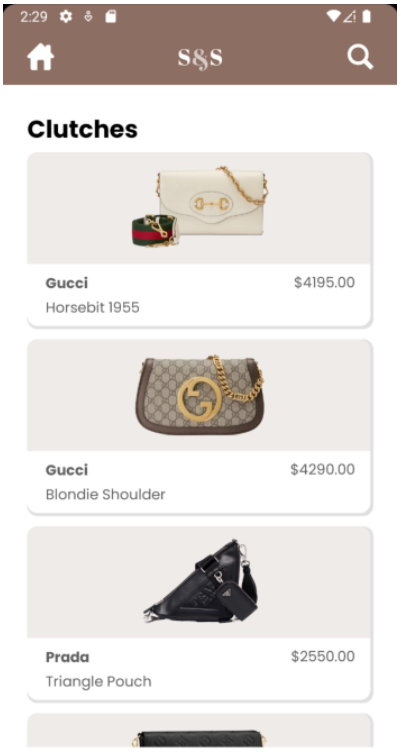


*Figure 4: Final Clutches Listview Activity*

### Search Activity

Our initial wireframe design had the search results appear on a screen without the search field and instead a heading with the search term used. However, we replaced this by displaying the search results below the search field. From a user experience perspective, this approach allows easy access to modify and change the search.

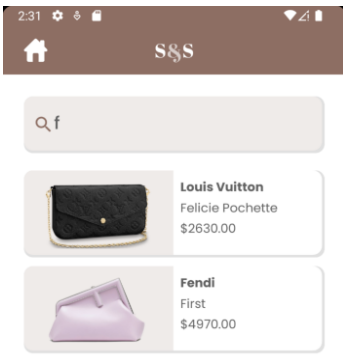


*Figure 5: Final Search Activity*

## 3.4 Cache Functionality

When implementing our application we realised that there was a very noticeable delay when loading the category and product information from the database onto the main page. Another issue was the high amount of repeated code throughout our activity classes to call an onChanged() method to retrieve the changes from the firestore database in order to dynamically change the contents of our popular and favourites list.

To overcome this issue we have taken inspiration from the class demo and implemented a cache which was not a part of our initial design document plans. During the splash activity, the data from firestore is loaded into shared preferences that act as a cache to temporarily store data while the application is in use. This allows for seamless loading of the main activity and more simplified code throughout our activity classes.

## 3.4 Data Providers

Our final application has two data provider classes CategoriesDataProvider and ProductsDataProvider that were not considered as part of our initial design document plans. However, when setting up our firestore database these classes were implemented to easily populate the collection and documents. This solution was a lot more practical than manually updating the fields of the database. Also, having these classes proved helpful when deleting and repopulating the database was required during testing.

## 3.4 Sort and Filter Features

Our design document included plans to implement sort and filter functionalities on the listview activity. During the implementation process, we realised that we were not keeping up to our expected schedule and as a result, these two features were not able to be implemented given the time constraints. As these features were not part of the requirements they were the lowest priority and were dropped from the project.

# 4 Good Coding Practices

## 4.1 Naming Conventions

Throughout our code base, we made sure to be consistent with naming conventions. Following these naming conventions not only ensures consistency but also improves the overall quality and understandability of the code. Names are intuitive and easy to understand, allowing for more productive development. The naming conventions that we used are as follows:

- All activity classes are named _Activity
- All adapter classes are named _Adapter
- All data provider classes are named _DataProvider
- All repository classes are named _Repository
- All view model classes are named _ViewModel with the prefix the same as the related activity
- All interfaces are named with ‘I’ prefix
- Variable, method and package names are in lower camel case
- Classes, interfaces and enums are in upper camel case

## 4.2 Code Commenting

Javadoc comments explaining the purpose, parameters and outputs are written for methods when necessary to ensure a better understanding of their purpose and functionality. Any other code segments that are not trivial have small summarising comments. Code commenting ensures developers don’t waste time reading code to understand its correct use. This helps to reduce errors when using methods. Comments have been omitted for methods or code segments with obvious functionalities to avoid unwanted clutter.

## 4.3 Package Structure

Our project code base consists of a logical package structure with all files sorted under relevant sub packages according to their layer in our class structure. Class interfaces are stored in the same sub packages as their respective classes to ensure they can be identified easily. There is only a small number of class files in each subpackage to maintain a clean structure and ensure it is easy to locate files.

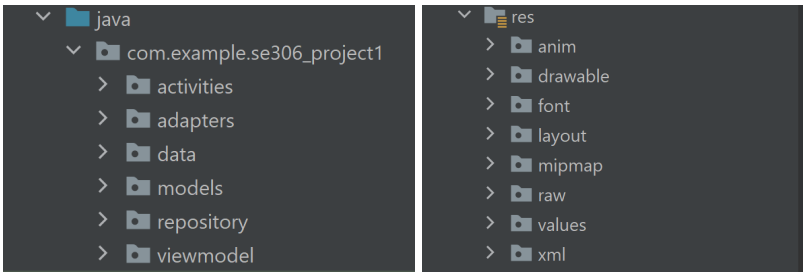


*Figure 6: Package Structure*

### 4.4 Git Branching

To minimise code conflicts and optimise effective parallel programming, our team followed good branching practices. Branch names were based on the associated Trello card for that specific feature or fix. Both the Trello cards and branches were named in the format SS00/branch_name with the number for each incrementing. To ensure this format was followed we set up a branch template in our project Trello board.

We were initially closing branches after they had been merged into the main branch, however this practice was sadly neglected throughout the remainder of the development.

### 4.5 Code Reviews

In our GitHub repository, we enforced a branch rule to only allow merging into the main branch from pull requests. All code needs to be reviewed and approved by at least one other member of the team. This helped to maintain the quality of the code on the main branch. It also ensured that team members were aware of changes happening in the main branch and could keep track of the progress.

Unintentionally, this practice was not maintained as small commits were made to the main branch due to convenience. However, in future projects we will aim to follow this practice as we realise how necessary it is to ensure the main branch is kept stable.

In the future, we would use squash merging to keep the main branch tidy and easier to read as our branch commit history for this project is very messy and difficult to understand.

## 5 Exceeding Requirements

### 5.1 User Favourites

To exceed requirements and go beyond the expectations of this project, we have implemented a ‘favourites’ feature. On the details activity users can select the heart icon to set an item as a favourite. Once a user selects at least one favourite product, their favourites can be viewed and accessed from a horizontal recycler view on the main page under the popular list.

Products identified as a favourite are labelled with a heart icon everywhere throughout the application to ensure they are recognizable by the user. This includes when the product appears in the popular list, list activity or search results.

Implementing user favourites was important to us to improve the user experience on the application by providing a personalised feature.

## 6 Conclusion

Developing ‘Sammy & Sun’ has taught us a lot about the importance of writing high-quality code with good software architecture. We have learnt a lot about the value of attention to detail when ensuring SOLID principles and good coding practices are met. Our project

demonstrates a careful consideration of these requirements from our class structure and code implementation to our professional and modern GUI design. Although there may be areas to improve, for the most part this project demonstrates effective implementation of good software architecture for a high-quality, maintainable and extendable solution.