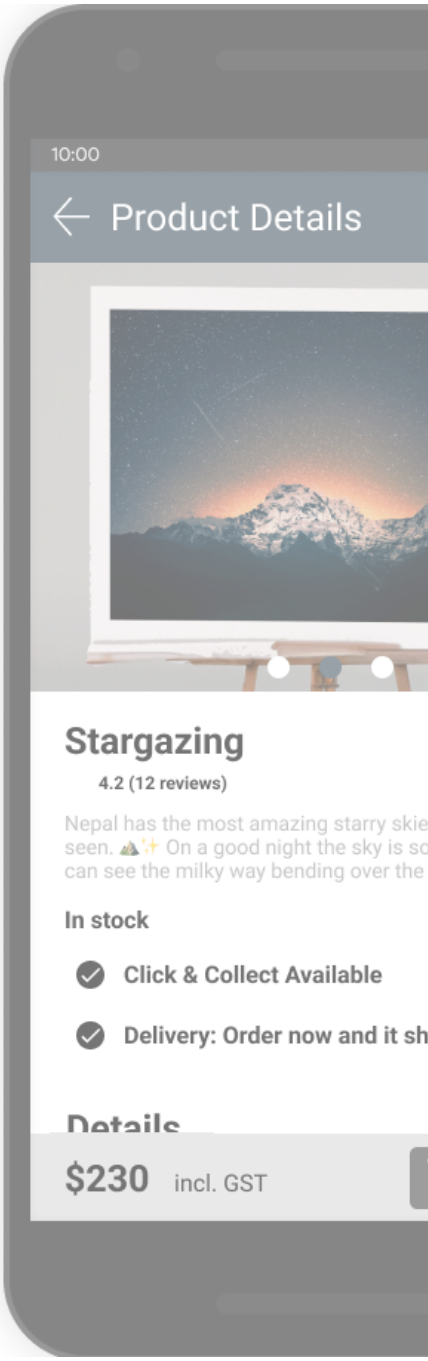
SOFTENG 306

Team 17 Quality Code Report

GALLERIA

Team Members

Brendan Zhou - bz503
Raymond Zhang - xzha897
Raymond Feng - rfen629



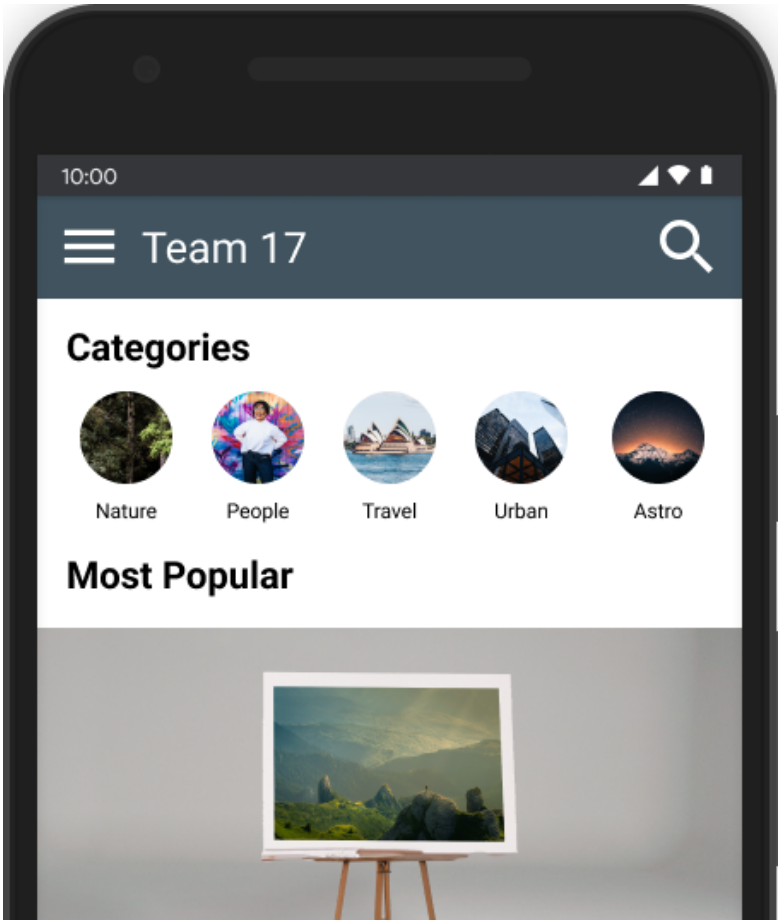


Table of Contents

1. Introduction

YouSee Soft is a company that is specialized in Engineering Mobile Applications. This year Team 17 has been tasked to participate in developing a native application that can be used to showcase and sell products online. Over the course of these six weeks Team 17 has developed an application called Galleria where that distributes and sells different types of artwork.

This report will outline the steps taken during the development and design cycle of our application Galleria. We will cover the realization of the SOLID principles, our reflection on the SOLID principles and the different good coding practices that were applied throughout our development.

2. Solid Principles

In the interests of developing robust and maintainable code, the SOLID principles were key guidelines used to inform the design of our application. Furthermore, in all situations applicable we have attempted to follow best practices such that we produce high-quality code. Explanations of the considerations and implementation of each of these principles are detailed below.

The Design Document has explained the SOLID principles thoroughly so in this section it will cover the implementation during our development.

2.1 Single Responsibility Principle

When designing and implementing the classes of our application we ensured that each of the classes executes one responsibility only. This was achieved by using our MVVM(D) model that was further explained in the design documentation. An example of this in our architecture and code is each of the entity classes where they serve as a template for the information that each of the entities should uphold.

The Single Responsibility Principle is prominent throughout all of our application's architecture. Classes are separated into layers with a specific purpose such as a repository which is used to access each entity types used in our application with the repository only handling one task which is to access information in our database. The instances of repository classes in our application are specific to the type of data they handle, such as UserRepository or ProductRepository, to maintain a strict scope of responsibility to exactly one data type. Similarly, in the domain layer, the associated ProductUseCase and SearchUseCase classes integrate with the repository classes through an interface such that they do not need to know implmenetation details and are decoupled. This supports Single Responsibility Principle further as these domain classes need to only fulfil their business logic rather than worry about integrating with the concrete instance of the repository class it depends on.

The data classes we implemented (e.g AlbumArt, PaintingArt) all extend from a generic abstract Product class, so no superfluous fields or information is retained and accidentally accessed. This also clearly distinguishes the responsibility and purpose a given object class fulfils, in terms of what it should represent.

Classes that are new to our design and not foreseen in our design document include the Adapter class (ViewPagerAdapter and SimpleListInfoAdaapter), and ViewModel classes. These additions further improve the adherence to Single Responsibility Principle, as adapter classes remove the responsibility of an Activity class to dynamically manage data objects and layout types in GUI components (RecyclerViews), and ViewModel classes removed extra burden from Activity classes. This meant Activity classes were entirely dedicated to the construction and monitoring

of the GUI, whilst ViewModel classes would exclusively handle data and business logic related to the current GUI screen.

2.2 Open-Closed Principle

The Open-Closed Principle has been clearly maintained throughout the whole implementation of our application to minimise coupling between our classes and show the clear interaction in our MVVM(D) model. Interfaces have been used acrossed our whole architecture to allow the implementation of the classes to be open for extension, and closed for modification. By having this layer of separation it supports decoupling, as the interfaces make it easier to add more classes and extend onto the base functionality similar to the interfaces we had for each of our product categories photographic, albums, paintings and AI generated. This was useful because reduced the disturbance and made it easier to implement due to less changes required in the overall code base. As a result, we were free to make changes as needed to concrete classes such as in the view model layer, without worrying about flow-on effects to integrated classes such as repositories due to the interfaces between them. In fact, once our interfaces were defined, they were not changed as although we expected and did see behaviour of concrete classes evolve, the integration and interaction between layers of classes did not. This application of the Open-Closed Principle therefore greatly supported the development process by reducing the amount of refactoring required.

2.3 Liskov Substitution Principle

The Liskov Substitution Principle ensured that there is sufficient levels in the depth of our class hierarchy meaning that all the subclasses can act as substitutes or replacements for the superclass without causing issues. This is evident by the implementation of our Repository interface which is referenced inside our UseCase classes with an abstract subclass IRepository, further extended by another interface class IProductRepository. Furthermore, we see the Liskov Substitution Principle in practice when considering data object classes, such as how extended classes such as ProductDetailDto can replace ProductInfoDto, and specialised product classes (e.g AlbumArt, AIArt) can replace its superclass Product with no issue in the functionality of our application. These promoted intuitive design and structuring of our code, following the general philosophy of the SOLID principles.

2.4 Interface Segregation Principle

By adopting the MVVM(D) design architecture all the interaction between our modules and layers are done layer by layer to promote robust development. For any of the given classes, it only has the methods and functionality required to perform it's intended task which makes the design of each module more independent and readable as less dependencies are required. These interfaces include our UseCase, IUserReposistory,

IProductRepository interfaces etc. This benefits the overall code base and quality of our design because it reduces the amount of coupling and makes it easier if required to make changes in our design in the future. As our application only supports one type of application, the importance of Interface Segregation Principle is not fully realized, but our design fulfils this principle in need of future extension. An example is how our repository interfaces extend off a base IRepository interface class, with the intent being that if meaningful extension is required, a new interface specialised to that purpose (without superfluous/redundant code) can be rapidly extended off of IRepository. This prepares our application well for adhering to the Interface Segregation Principle, as due diligence has been observed in this regard.

2.5 Dependency Inversion Principle

The Dependency Inversion Principle was promoted by adopting the MVVM(D) design architecture because it promotes the encapsulation of lower layers from the higher layers. In our application, there are interfaces created to facilitate the communication between each of the layers to ensure higher abstraction. This it prevents the classes from directly depending on each other which would increase the code rigidity, this is evident in our design in the usecases associated with the different repository classes and data interface classes. This further supports our adherence to the Dependency Inversion Principle as changes or additions to concrete classes can be done with minimal impact on the client's interaction with the system, fitting within the existing framework. A design choice related to supporting the Dependency Inversion Principle we made was the creation of the DIProvider class (Dependency Injection Class). This class provides a centralised point for concrete classes to be injected into dependent classes which take the interface as the parameter/type. From a functional point of view, these dependent classes can operate as intended and not receive side-effects from external changes given the injected object fulfils the interface. More importantly, from a design point of view, this greatly improves flexibility and freedom as we can adjust these external classes and make changes where necessary within DIProvider such as via adding more class instances, that can then be easily injected and consumed by the exiting dependent classes.

3. Inconsistences

In comparison to our initial design and scope of classes specified in our design document, the following changes were made throughout the development process for valid reasons:

3.1 RxJava and the Observable Pattern

We noticed quite early on when designing our data persistence layer that we could not return our model objects directly from the repositories, as that would result in a blocking call to the database. The two options we considered were to either (1) launch all repository operations on its own thread and creating these threads inside the ViewModel layer, or alternatively (2) create “Observables” that wrap a repository response, and that you can subscribe to its result (similar to a callback). We decided to go for option (2) after considering recommended android practices from the app architecture guide.

With option 2, Firestore has a built in “DocumentSnapshot” object that you can add an onSuccess listener to, however we decided we couldn’t pass this back from the repository as it would be breaking the abstraction between our persistence layer and other layers (in that our domain and views should not know that Firestore is being used as our database). Similarly, the documents returned from the DocumentSnapshot object are not in the right form, and would be database model objects (e.g. ProductDbo in our codebase), as compared to normal model classes (PhotographicArt or other children of the product class). Another option was to use a LiveData wrapper provided from the Android library as these are also observable, but after some research and viewing the latest android app architecture documentation on exposing data from repositories[1], we decided that LiveData was primarily a UI concern and should not be forced into the repository layer. As such, the official Android recommendation was using a library called RxJava, which brought in these Observable wrappers.

We opted to use the “Single” class for returning any data from repository layers that were blocking calls, as all our data operations were one shot operations with a single return item. We were also able to extensively leverage RxJava’s built in operators that allowed us to manipulate the promised data within Single classes, particularly inside our UseCase classes in our domain layer, so that we can manipulate our model objects in a way that returns the exact data our UI layer needs (e.g. getting all products saved by a user involves accessing both UserRepository and ProductRepository). Finally, in the ViewModel layer, we converted the RxJava observables into LiveData which are able to be lifecycle aware (particularly useful for databinding in Android - see our implementation of LoadingViewModel for details).

---

[1] https://developer.android.com/topic/architecture/data-layer#expose-apis

3.2 Adapters

In the initial development of our system design model the team did not consider the use of adapters in the View Model classes due to the lack of experience in Android Development. After research adapters were essential because they provided a bridge between the UI components and data source that helps us fill data into the UI component. It holds the data and sends the data to an Adapter view and from that the data can be displayed it various differents components and views such as the ListView, ViewPagerView and GridView etc.

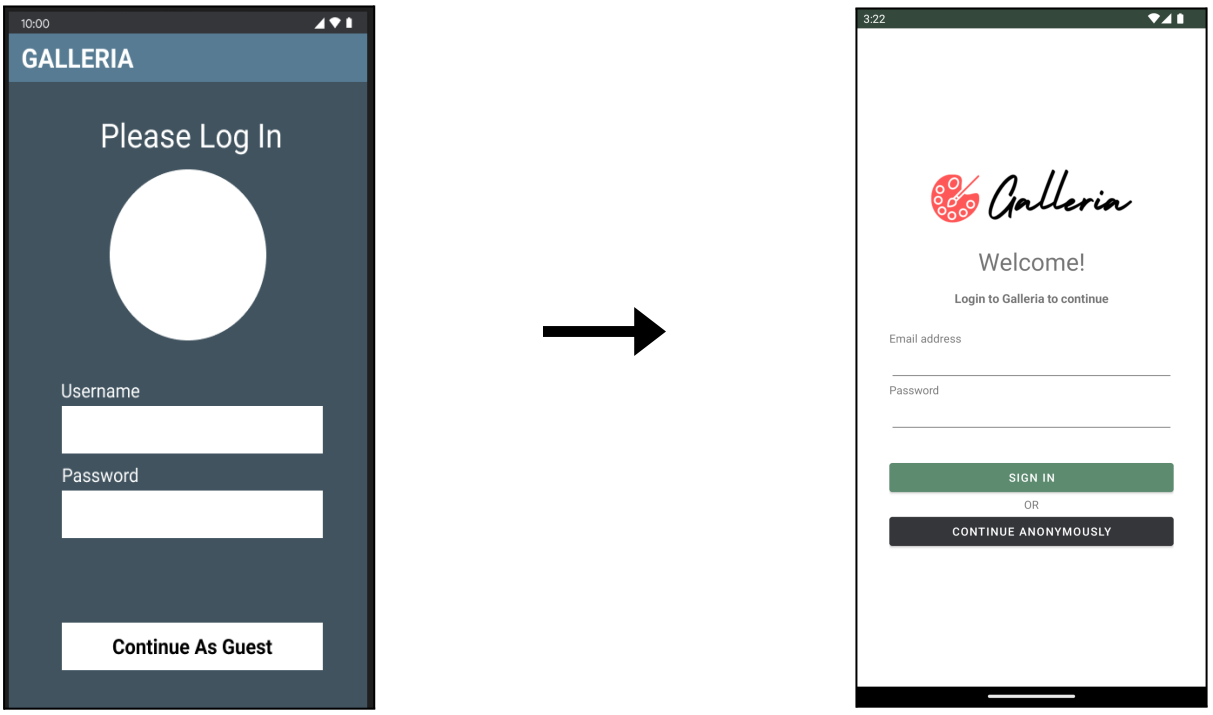
3.3 ICachedRepository

In our initial design we had an interface for a ICachedRepository to handle all the caching for our products which resided in our domain layer. However, after considered the team realised instead of having an interface for our CachedRepository it would be more suitable to make it an abstract class and have the ProductRepository class in our infrastructure to extend it. This was because caching is a primarily infrastructure/persistence level concern.

3.4 Authentication Use Case

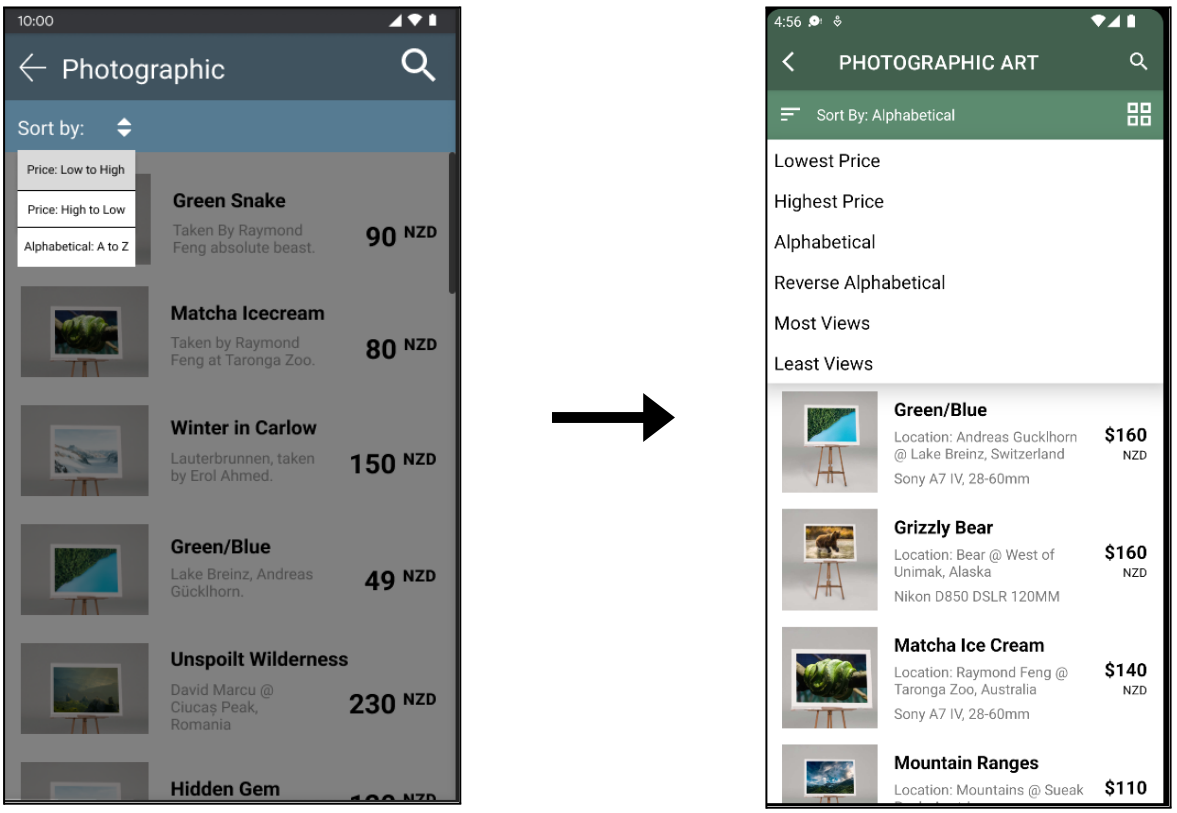
In our system design model we had included an AuthenticationUseCase class where it would handle all the authentication such as registering and logging in for each of the users from the Firestore. However, due to reconsideration regarding the schedule and timeframe of the development of the application we have decided to halt the implementation on the AuthenticationUseCase. Additionally, due to our use of the MVVM(D) design architecture and model the code base is designed in way for if in the future we would like to implement this UseCase it be easy to implement since our classes are implemented separately in different layers meaning that there is low coupling and rigidity.

3.5 Login Page



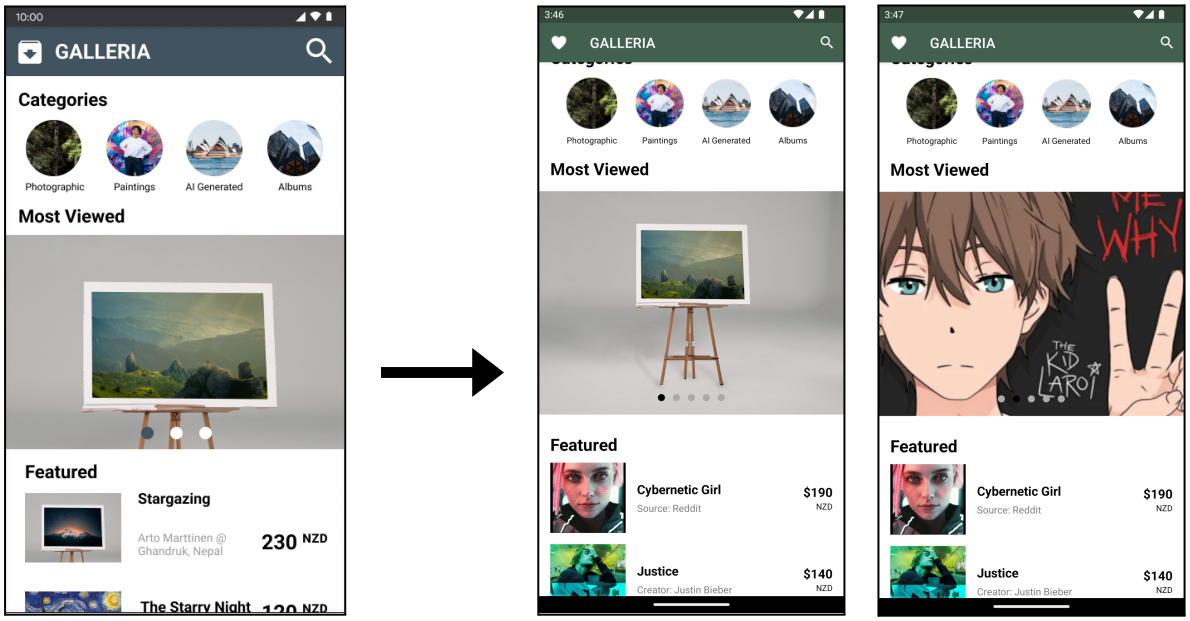
The redesign of the Login Page was implemented as the original design (on the left) did not fit our design aesthetic. The final design on the right is much easier on the eyes for the user and provides an easier understanding of the intended userflow.

3.6 Filter Redesign



The filter additional feature has also been redesigned to make it more consistent with the overall design and theme of Galleria. The text used and font are matched with our font used throughout the application and and it covers a whole section in the top, whereas, before it only covered a small portion of the screen making it awkward for user interaction.

3.7 Main View Pager Addition



Another inconsistency regarding out initial design to our final design would be the addition of the Android automatic sliding View Pager allowing users to either swipe to see the most viewed products or have it automatically sliding through our top 5 most viewed products.

The most viewed products are determined by the amount of views each of the artworks have in the database and will be filtered according to the amount of views, with the top artwork in the Galleria being the highest etc.

4. Re-Measured System Metrics

We designed our system using a mixture of MVVM (Model-View-ViewModel) and Clean Architecture principles, which in itself promote the use of SOLID principles and good design practices. In particular, this provides really great separation of concerns by splitting classes into separate layers that can only communicate through one direct layer (above and below).In your Re-Measured System Metrics we will highlight the changes and differences from our previous System Metrics in our Design Document.

Key:
Green - Original System
Yellow: - Updated System

4.1 Updated System Metrics

	Measurement #1	**Measurement #2**	**Measurement #3**
Class Name	**NOC**	**DIT**	**CBO**
Activities			
Activity	7	0	0
LoginActivity	0	0	1
MainActivity	0	0	1
SearchBarActivity	0	0	1
SearchBarActivity	5	0	1
SearchResultActivity	0	0	1
SearchResultActivity	1	0	1
CategoryResultActivity	0	0	1
ProductDetailActivity	0	0	1
SavedProductsActivity	0	0	1
View Models			
LoginViewModel	0	1	2
-	-	-	-
MainViewModel	0	2	2
LoadingViewModel	4	0	0
ListResultViewModel	0	1	2
SearchResultViewModel	0	2	2
CategoryResultViewModel	0	2	2
ProductDetailViewModel	0	2	2
SavedProductsViewModel	0	2	2
Data Transfer Objects (DTO)			
UserDto	1	0	0
CategoryDto	1	0	0
ProductInfoDto	6	0	1

AIArtDto	0	1	0
PhotographicArtDto	0	1	0
PaintingArtDto	0	1	0
AlbumArtDto	0	1	0
ProductDetailDto	1	1	0
Use Cases			
AuthenticationUseCase	0	0	2
-	-	-	-
SearchUseCase	0	0	2
ProductUseCase	0	0	7
Entities			
User	2	0	0
Category	0	0	0
SearchAutoCompleteTerms	1	0	0
CurrencyCode	0	0	0
AIArt	0	1	0
AlbumArt	0	1	0
PaintingArt	0	1	0
PhotographicArt	0	1	0
Product	5	0	0
Repositories / Data Persistence			
IRepository	3	0	0
IUserRepository	1	1	0
ICachedRepository	1	1	0
ISearchHistoryResult	1	1	0
IProductRepository	1	2	0
Database Objects (DBO)			
ProductDbo	1	0	1
ImageDbo	0	0	1
SearchHistoryDbo	1	0	0
UserDbo	2	0	0
Infrastructure			
CachedRepository	2	1	0
FirestoreRepository	3	0	0
UserRepository	0	1	0
SearchHistoryRepository	0	1	0
ProductRepository	0	1	0
Average	0.82	0.61	0.71
Maximum	7	2	7

5. Good Practices

In order to ensure that Galleria was developed in a manner that would lead to high quality code, our team followed best practices and conventions in both our technical implementation as well as in the holistic development process. Each of these practices assisted with improving our source code and process, supporting the SOLID principles as well as minimising code smells where possible. A selected list of examples of such practices are outlined below.

Good Practices - Technical Implementation:

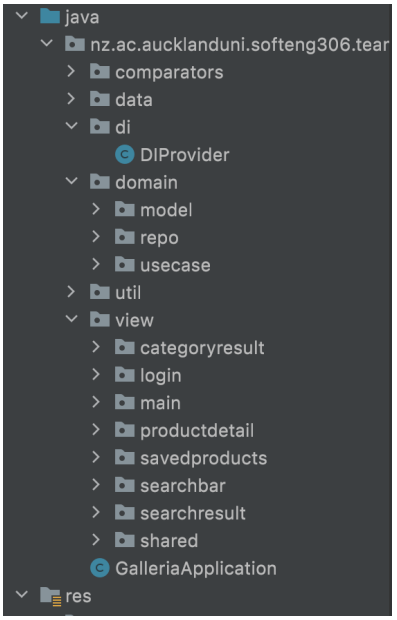
- Layered Architecture
- Code Segmentation
- Meaningful Naming Conventions
- Indentation & File Layout
- Inheritance Hierarchies
- Access Modifiers
- Scalability

Good Practices - Development Process:

- Feature Branches
- Pull Request Reviews & Branch Protection
- Project Tracking
- Pair Programming
- Testing

5.1 Layered Project Structure

Our initial design document contained a system design representing a concept of how classes of code would relate and interact with each other in the context of our application. This conceptual design was strictly adhered to in our implementation, and manifests in our project's package structure where classes are organised into their respective layers within the design's architecture. This promotes better understanding and maintainability of the code base, as classes are grouped by purpose and depth of specialisation. Consequently, it is intuitive for developers working on the project to understand where a given file sits within the high-level system design, and gives a good foundation for further expansion. Additionally, having our code separated into separate layers, and by depth of complexity and purpose, we promote better reusability of code following OOP practices such as composition.



5.2 Code Segmentation

An important aspect of high quality code is the readability of the code, of which logical code segmentation and separation is a large contributor. This was realised in our project across all meaningful classes and methods, as we followed a loose convention of segmenting code into chunks based on either tightly coupled functionality, or a maximum level of complexity.

The reasoning behind this process is that for the purposes of understanding code in retrospect or for a fresh pair of eyes, code would be segmented together into logical groupings where it fulfils one granular purpose or is otherwise not so long that is becomes undigestible. An example of this is in the various Activity classes within our project, where in the onCreate method several lines of code are grouped together at a time. The sequential code relating to the set up of meta data from keys passed in from an intent are chunked together, due to them being tightly coupled under one single purpose. Similarly, code relating to the initialization of the toolbar are segmented together and spaced apart from the previous code, for identical reasons and to further distinguish the difference in purpose of these groupings of code. Furthermore, having code segmented appropriately is important to show the workflow of how a certain class functions, especially so in view classes where the set up of the GUI must follow strict rules and order, or in the case of database interaction and the data pipeline where there are strict timing and ordering requirements. In these scenarios, code segmentation assists further with readability and understanding by providing an intuitive ‘step-by-step’ structure that guides thinking.

5.3 Meaningful Naming Conventions

Beyond the structure of our code, the practice of having specific naming conventions and consistency of their usage within our project was key to aiding readability. This was applied across all class names, field names, method names and directory names in our project, with the emphasis on being concise, meaningful and informative.

Classes and code related to data was named directly after their physical representation in the Firestore database, such that it was intuitively mapped to what exists already. For example, Product was a data class which represented the Product document within the identical Firestore collection, with identically named fields representing the fields in Firestore. Associated classes to Product but within our system design such as ProductDbo and ProductRepository expanded on this convention. Similarly, data classes in our domain layer as well as fields of classes wer all declaratively named to make it obvious that they represented information and had no relation to business logic.

Across the entire repository, methods were named in the following action-oriented style:

<Verb> + <Object> + (optional)<Context/Meta-information>

Examples of this are ‘loadViewModelData’, whilst ‘listSavedProductsByUser’ provides an example where an optional contextual component was added to the method name. As they are all method signatures, we have ensured they followed convention and were written in camel case. Where used in our project, enums, constants and properties were all set in upper case to distinguish them from variables which also followed camel case.

5.4 Indentation & File Layout

Indentation was strictly followed in our code to follow good practices, as other languages would enforce correct indentation, as well as to make it syntactically clear what depth of nesting code belonged to. This is crucial especially in areas of our project such as our repository classes which handle data input/output to Firestore, where a notable degree of nesting is unavoidable. Appropriate indentation promotes readability and maintainability, as well as making it clear the local scope of a given segment of code.

Similarly, file layout was also important to help make classes well structured. All fields are placed at the top of each class, with constructors underneath. Methods always run on creation or most often were placed higher up in the class, such that it was easily accessible and quick to find, whilst more generic getter/setter methods and less commonly called methods were placed further down the file.

5.5 Inheritance Hierarchies

As our android application was very modular and the screens were strictly defined, with overlapping functionality, we ensured that the SOLID principles were observed through creating inheritance hierarchies in relevant areas. Following this practice ensured that we did not have duplicated code (thus avoiding needless repetition), and downstream classes were less opaque due to specialisation. This is observed in our view classes which inherit a lot of pre-defined classes from AndroidX, but also our self-defined classes such as SearchBarActivity. In other areas, our repository classes extend from a cached repository class to separate out the more general functionality of caching needed across the board, and specialise to their data type further downstream, as well as across an interface to concrete use case classes.

5.6 Access Modifiers

In order to support good design principles and concepts such as encapsulation, the use of access modifiers and the 'final' property were applied in relevant parts of our project. These prevented unwanted access and visibility of code to external classes or functionality where it is not needed, as well as ensuring that permanent objects were not overridden or updated erroneously. Following these practices added another layer of safety to the functionality of various components within our application, improving it's quality.

5.7 Scalability

A delicate balance taken whilst developing Galleria was considerations made for scalability, whilst avoiding the code smell of Needless Complexity. This meant that we deliberately avoided adding code to improve scalability and performance where the impact would be negligible or unrealised in the context of our application's demo and expected usage (<10 users). However, where changes could be made to directly affect performance of the application in a meaningful way that could be experienced by a single user, these additional requirements were taken on board. A prime example of this is the implementation of a caching repository class, and related caching functionality to support faster data exchange/loading in our application. There is a notable improvement in speed for single users, and data loads instantly in our int ended use which is an improvement to the experience. However, this also promotes an acceptable standard of speed in the case our application needs to handle either a much larger set of data or a much larger number of concurrent users. Implementing this functionality early on in the formative stages of our application meant avoiding technical debt and improving extensibility, supporting good design standards, scalability and maintainability.

5.8 Feature Branches

To ensure an accountable and trackable timeline of development, feature branches were employed to uphold good development practices whilst adding stability to our development. This allowed concurrent development on separate aspects of the application, without disturbing others, and whilst keeping a working version of the application in our main branch. It also granted us the ability to isolate and revert features if needed, and to better allocate workload amongst the team.

5.9 Pull Request Reviews & Branch Protection

Reviewing of pull requests and automatic branch protection rules set up on Github ensured that a quality-first approach was taken to development. This prevented

conflicts within the team, and improved transparency across the team such that developers could understand incoming changes as well as hold each other accountable to meeting standards and conventions we agreed upon. Furthermore, these practices ensured that errors and bugs were not accidentally overlooked and introduced into our main branch, and promoted more collaboration and testing between develoeprs in the team.

5.10 Project Tracking

In order to have a well organised pipeline of development tasks and overview of how our application was progressing, we employed best practices following an agile development process using a project management/tracking tool. In our case, we used Github Projects and followed a KanBan style approach to defining, assigning and monitoring significant tasks and requirements. This promoted fair workloads, visibility of each other's responsibilities, and vastly improved communication between developers. Each developer had greater knowledge of who to talk to for a given question or need, the status of what functionality is available to integrate with, and any changes or decisions that needed discussing. By adhering to this, the overall productivity and efficiency of the team was improved, supporting the successful delivery of the project.

5.11 Pair Programming

As the development process was iterative and forced us to reconsider our initial design choices specified in the design document, a single developer could not progress on a single task. To overcome these challenges, pair programming was used at stages of development such as major functionality where we were inexperienced, or in evaluating multiple implementation options for a given task, such that developers could work together and support each other. This practice was beneficial to the success of the project due to the ability to quickly validate and justify design choices, and to fill in gaps of knowledge/experience in any one area of a single developer, improving the speed of development.

5.12 Testing

In order to ensure our application met our standards according to our requirements, testing was a crucial practice employed to check for both stability and functionality as intended. More importantly, we collectively followed a process where other developers tested code and behaviour written by oneself, to provide better quality assurance more akin to black box testing. This prevented bias and increased accuracy in testing as the testers would have a fresh perspective and be more perceptive to issues as compared to the developer who is already accustomed to the new feature, as well as having knowledge of the underlying implementation. In terms of the holistic quality assurance process, this only precedes a formal code review by the other developers

mentioned above in pull requests, which would then scrutinise the implementation quality as confirming the end behaviour was correct.