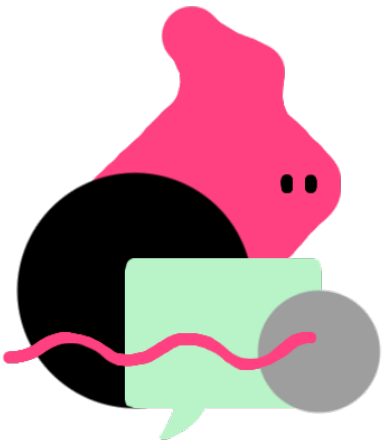
QUALITY REPORT

SOFTENG 306
PROJECT 1



TICON

TEAM 15
Juwon Jung, Hayoon Seo, Julie Kim

SOLID Principles Applied and Realised

Single Responsibility Principle

The *Single Responsibility Principle* has been achieved as intended in our designed document. Because of the nature of our application, only selling *emoticons*, there were no differences in the properties of items in different categories. Thus, we only needed one class, *Emoticon*, for all items in the application, which meant that we did not require extra classes that extended *Emoticon* class.

ITaskListener interface allows the *Activities* classes to be separated from any implementation details of emoticon methods, or in other words, the *Activities* classes do not need to know how data implantation works. This is very useful as the users will not recognise the changes being made in backend, as frontend will not be changed.

The *DataProvider* only contains methods that are related to fetching and updating data from the database, indicating that the *DataProvider* has a single responsibility in dealing with the database, and also eliminates this functionality from all other classes.

Open-Close Principle

The system is designed to be easily extended to, however not available for modifications that would affect other parts of the system. This is achieved through the use of *Emoticon* model class, which allows for easy extractions, so that additional changes can be easily extended, and minimal change is broadcasted to other entities in the system.

The overview is that, because the system is designed for *Activities* class in frontend to only interact with interfaces and model classes in backend, the risk of erroneous changes occurring being visible to the users are minimised.

We initially designed an *Emoticon* interface, however, in development, realised that it was redundant in our current application's context, and *Emoticon* model class was sufficient, and therefore removed it.

Liskov Substitution Principle

The *Liskov Substitution Principle* states that any object that interacts with the superclass or its interface should be able to instantiate by any of the subclasses without violating any of the super class policies and methods. To ensure that this principle is not violated, the subclasses that extend or implement a class have been directed to its parent class. This principle has been kept to by having no subclasses extending the *Emoticon* model class, as no specific extension subclasses have been required. Although we could potentially have had subclasses, it would have been a needless complexity smell.

Integration Segregation Principle

Having individual interfaces for each *Activities* class, and allowing access to only the method the class requires would be a way to follow the Integration Segration Principle. However, we realised this was not only difficult to achieve, but also very redundant, because our scope of the app is only for a user to browse through various items, and does not have to buy or upload anything to the database. All the users of the current system will perform the same actions, and thus, the views and operations will also be the same.

Dependency Injection

This principle has been displayed in various places throughout the design. For example, *MainActivity* is not associated with concrete classes, but with the *Emoticon* model class. This ensures that the interaction is loosely coupled and no direct instantiation occurs. This is the same for all other *Activities* classes. Deliberately passing objects into our *Activities* classes means that they do not have to go out to fetch objects. We do this by injecting bundles with required information into the activities.

Similarly, lists of data is injected into the adaptor classes and into its constructor, so that it can use the data immediately without having to deal with the *DataProvider*, or *Firestore*.

The data from *Firestore* is fetched directly into the *DataProvider* methods, which we have set so that the data can be immediately converted into the model class objects, by matching categories and fields.

Inconsistencies from Design Document

Even with a very thoroughly considered design document, some inconsistencies were inevitable as we learnt and understood the application development process. However, overall, there are no major deviations from the initial design outlined in the design document.

MainActivity

Originally, the design contained a *Popular* section as a recyclerView, and *Categories* section took up more than half of the screen. We came to the conclusion that the home page should be more eye-catching, while maintaining the categories section to be accessible. This was achieved by adding a *New* section as a listview, similar to the *Popular* section, and shifting the *Categories* section to the top as a single horizontal scroll-view.

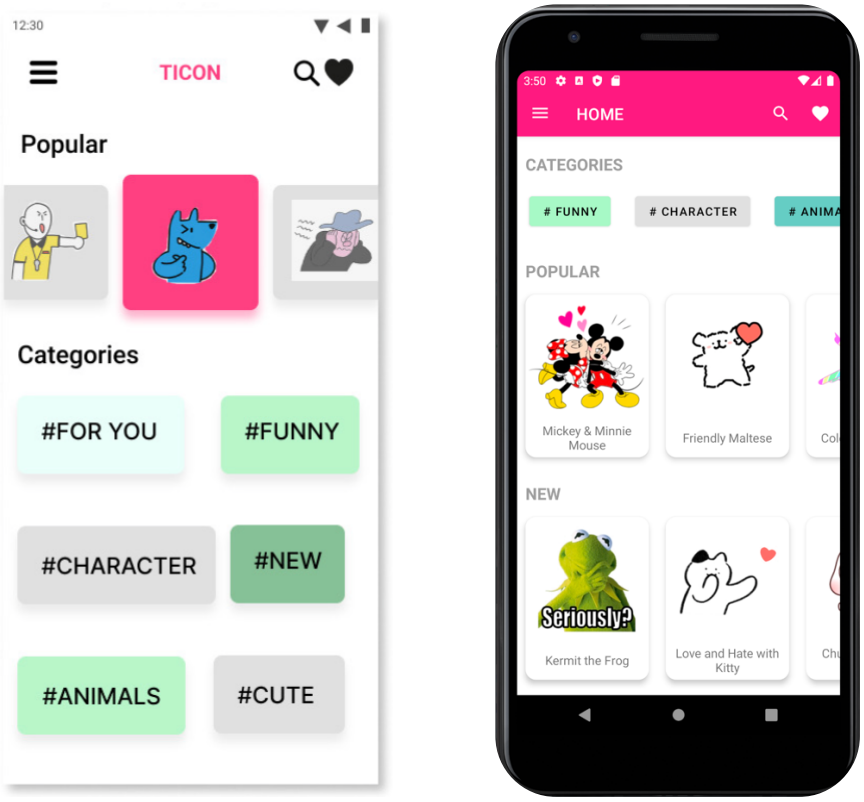


Figure 1: Initial and Final Design of Homepage

SearchActivity

The top-bar is kept throughout the user's search, which allows for increased consistency in the design and to give users more freedom in navigating. The sorting option has been removed as the feature seemed redundant within a search, as users are already able to specifically search and explore emoticons.

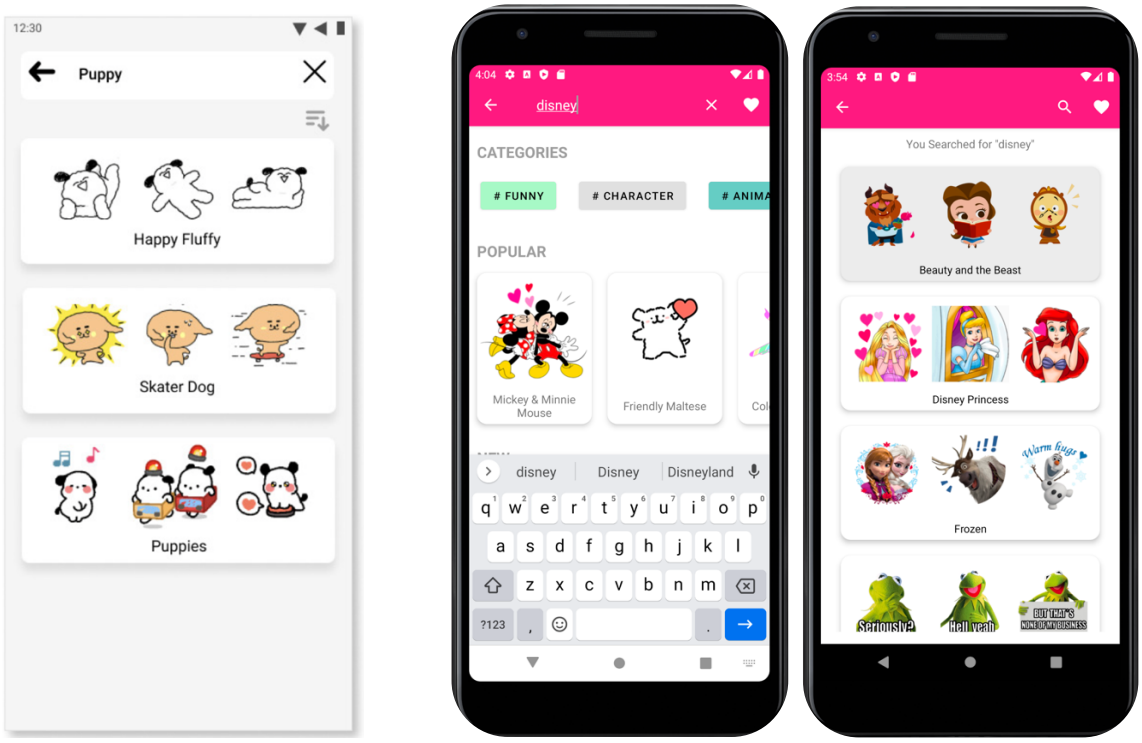


Figure 2: Initial and Final Design of SearchActivity

Wishlist and My Emoticons

The design has remained mostly consistent. A new activity, *MyEmoticons* has been created which we did not realise we needed while designing the class diagram. *MyEmoticons* activity has a similar functionality to the *WishlistActivity*. By creating this new class, the *Single Responsibility Principle* hamintine, as it als particularly ealingwith displaying the pge.

a*MyEmoticons* has been aThe *Search* functionality has been replaced with *Home* functionality, which takes the user back to the *Homepage*. This was because the search functionality's use is not so useful, and replacing it with a *Home* button will provide better access, and thus navigation, to other functionalities within the sidebar drawer in *Homepage*.

Another difference is the sorting functionality, and the heart icon within the cardview in Wishlist. This was mainly due to time constraint, as it were not essential for the user to browse through their *Wishlist* and *My Emoticons* as thoroughly as they would do through categories and search functionality.

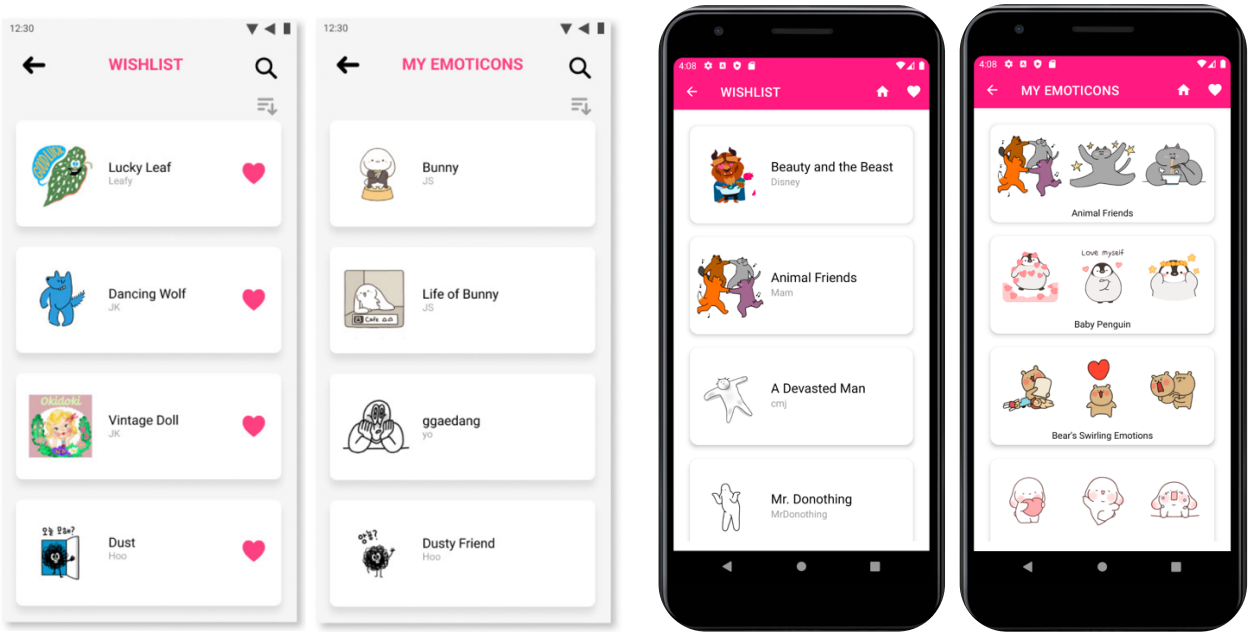


Figure 3: Initial Design and Final design Wishlist & My Emoticons

Coding Practices

For efficiency and effective application development, our team had discussed and set ideal coding practices prior to our development. These coding practices have been kept to throughout the entire development.

Abstraction

- **Consistency in Naming of Fields, Methods, and Classes**

  One of the very first conventions we decided on was the naming convention.

  The model classes determined the ways how we set up collections, documents, and fields in the system.

  In relation to Firestore, we made sure:

  - the collections matched the categories we have designed; e.g. '*animals'* category, when fetched, is stored as a collection '*animals'*.
  - the documents were consistent with the emoticon names.
  - the document name matched the corresponding *id* field. This was so that we were able to get the document with its *id* in backend.

  Classes were named according to their purpose. For example, *SortByPopularity* reflects its functionality clearly.

  Long fields, with more than one word, were named using CamelCase convention.

  Methods were named with the convention of *'[action] + [object] '*. For example:

  ```
  public static void setIncrementViews(String collection, String documentID)
  ```

  where '*action = set'*, and '*object = views'*.

- **Logical Commenting**

  Although commenting is necessary to increase the readability and usability of the code, however too much commenting can often result in elongating the code, and decrease the readability. Our team needed a convention to follow when deciding whether commenting is needed or not.

  - For **classes**, Javadoc was necessary, no matter how simple the code is. It is the general convention to do so, and also helps in the auto-generation of documentation for our system. Javadoc commenting was very simple and helpful, as it summarised all the code in a class, and thus, increased the readability, and prevented inaccurate changes.
  - For **methods**, commenting for only necessary cases. Very abstract and hard-to-understand code was commented. For example, simple getters and setters were not commented.
  - Comment on specific lines, or blocks of code, where the cod logic may be hard to read for others. For example, there are different blocks of code to set up the *Recylerview* differently depending on the categories and sort-by.

- **Segmentation of Code into Logical Blocks**

  Code were organised into blocks by functionality. Heuristically, every block of code will modify an input in a way, or provide logical assistance to other functionalities.

  Segmentation had to be done logically per different situations. It was prominent in data fetching and GUI related codes, as it is crucial because the organization of data is especially sequential. For example, the sorting functionality had to be applied in a specific order.

- **Correct Indentation**

  Code indenting was done mostly by analysing and editing the code by eye, as the code was being written. In addition, Android Studio's auto indenting functionality was used to check if there were any we have missed. Unnecessary blank lines were avoided, and overall, correct indentation improved the readability and maintainability of our code.

- **Logical Package Structure**

  The *java.example.ticon* package structure is as follows:

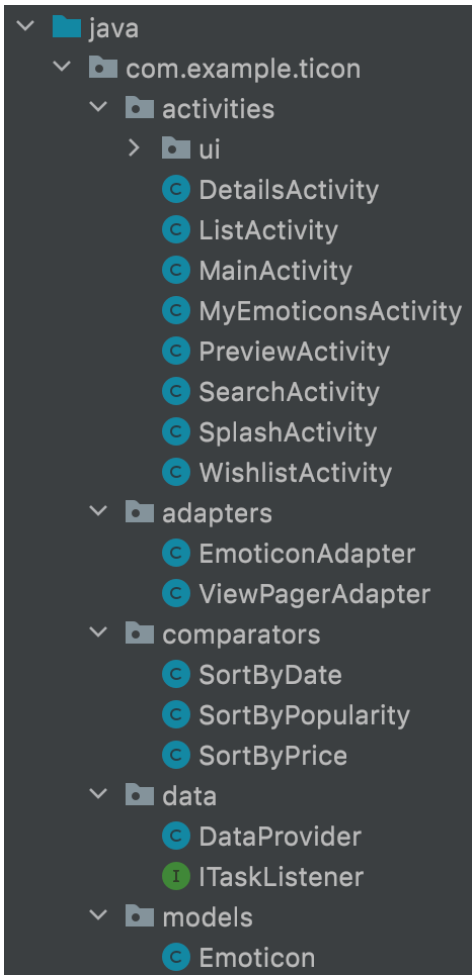


Figure 4: Java Package Structure

Essentially, this follows an adapter design pattern as shown below:

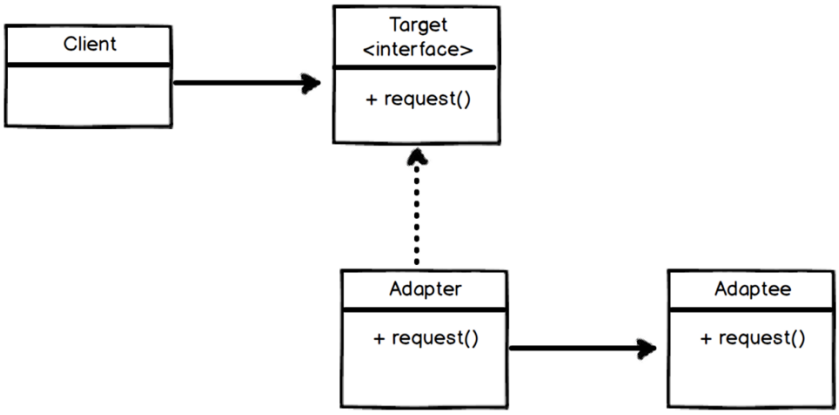


Figure: Adapter Design Pattern

Activities directory contains the files which the user will directly interact with. In the above diagram, this will be the *target*. *Adapters* directory contains the middle layer which fetches the data from backend, or FireStore, to assist in the frontend data functionalities. *Comparators*, *Data*, *Helpers*, and *Models* together supplement backend logic and any other assistance required from frontend. Although it may seem like a lot of directories, it was the best way of dividing up the system logically, as each directory supports different actions and requirements of the android application system linked to an external database.

- **Avoiding Deep Nesting**
  The maximum number of nested loops in any part of the system is two.
- **Long Lines of Code**
  Long lines of code can be an inconvenience when a developer needs to read and understand the code, and are often more prone to errors while developing. We utilised Android Studio's built-in Wrapping and Braces function, to increase readability.
- **Access Modifiers**
  Access modifiers had to be logically assigned to ensure effective and correct encapsulation and interaction between the different objects. For example, we have various listing options to use in our *Recyclerviews*, and accidently accessing an incorrect list would cause problems. We tried to use private and protected access modifiers where possible.

Code Development

- Regular communication
- Regular and clear documentation
- Pair programming
- Branching
- Pull requests

The development process has led to improving code quality in many different ways. We pair programmed much more than we expected, and often times, we were able to catch mistakes, prevent code smells, and fix bugs. Another set of eyes lead to reduction in the complexity of code also. Communication was crucial to ensure a common understanding of all elements and functionalities at a given time. All our meetings started with a stand-up, where each member was able to update what has been completed, and what they will do from now.

Version control has been used very effectively throughout our development. We used various branches, where each member could work on different tasks, and pull requests were created for major merges and changes.