306 Quality Report

Team 21

Kinan Cheung, Flynn Fromont and Maggie Pedersen

Table of Contents

1.0 Introduction

In general our design stayed consistent in following the SOLID principles, however we made the following major changes to the original design:

- removal of the unneeded Category class
- creation of the ‘ISpecialItem’ interface and hierarchy
- new specific adapters for activities
- added in an ActivityHelper class to remove duplicate
- created a specific SearchActivity as opposed to an internal component of other activities

We also faced some challenges in converting certain layouts to be responsive and handle landscape views, most notably the DetailActivity layout. This was due to the dynamic nature of the content, some items had a lot more information in the description. This caused some major imbalances in the weighting of the linear layouts used, leading to issues with scalings of the entire page. There was inconsistent scaling for different items. As a result of this, we decided that it would be best to lock the DetailActivity to portrait mode for a more consistent user experience. All the other activities and views had no issues with the resizability and changing between landscape and portrait views.

We received a lot of constructive feedback in the demo that we took into consideration and implemented during the last week. This mainly included aesthetic improvements like rounding image corners, fixing image sizes within cards, and adding more animations that stand out to the user.

2.0 SOLID Principles

2.1 Single Responsibility principle

The Single Responsibility Principle has been abided in our final design from what we laid out in the Design Document. Our classes encapsulated specific responsibilities, the different classes of Podcast, Song and Album provide the specific operations relevant to each of these concepts, while incorporating similar methods through the Abstract class Item. We took this one step further and introduced a new interface and class type, ISpecialItem, which was used to handle the group of responsibilities related to fetching and displaying the important information for items involved in the wishlist and top picks parts of the project. This was done as we deemed it less cohesive to include these in the original item class implementations.

We also followed through from our Design Document with separate classes for each activity that corresponds to the logic for each of the views. Each of these classes greatly improved maintainability and flexibility during development, as it was easy to locate any issues to the bodies of code causing the problem. This also meant that when we wanted to modify or improve the display of information it was neatly encapsulated in the proper section.

A change from the initial plan was the use of multiple specific adapters for the non-ListActivity classes. We used specific adapters for populating the recycler views in the MainActivity, SearchActivity and an additional ImagePagerAdapter which is used to correctly handle and display the images used in the DetailActivity view pager. We believe these additional adapters were needed to handle the requirements and contexts for each view, and by incorporating them the design better follows the Single Responsibility principle. This is because each adapter handles adapting data for the relevant context, and it would be going against the principle to incorporate all these different responsibilities into one large adapter. This would be incohesive and inflexible.

2.2 Open-Closed principle

We attempted to follow the Open-Closed principle as we planned to in the design document. However, as we began to implement some of the features like the number of hearts, and adding/removing hearts, we realised that they did not work the best for the context we were working with at the time. We made the new interface and abstract class for ISpecialItem to avoid modifying the IItem interface or Item abstract class with new responsibilities to keep in line with this principle.

The end result with both of these abstract classes still follow the Open-Closed principle, as both of these classes are able to be extended by subclass types, and allow for their own additions and functionalities, while also not modifying the original abstract classes. It was easy to make modifications in the subclasses as all their basic functionality was derived from the same super class. The design allows for easy addition of any new functionality or additional categories in the future if there are new forms of media that we would like to add.

The rest of the application is designed to work with the IItem and ISpecialItem interfaces, which means any additional functionality can easily be implemented with current views or layouts, or new adapters can be added to work with the current system for new displays.

2.3 Liskov Substitution principle

The Liskov Substitution principle has been properly applied and realised in our project as proposed in the Design Document. This can be seen through the proper realisation of both the IItem hierarchy that was proposed in the Design Document, and the ISpecialItem hierarchy that was added during production.

The IItem hierarchy ensures that the Liskov Substitution principle has been correctly applied as any object that interacts with the interface IItem can instantiate any of its concrete subclasses without violating any of the superclasses policies and methods. An abstract class, Item, is used to implement the interface IItem. This abstract class provides concrete implementation for common methods used by all item types, and abstractions are created for subclass specific methods so that child classes provide subclass specific implementation of these methods. The items that do not actually provide functionality for the abstract methods will throw an exception. This means that the Liskov Substitution principle is achieved as any class that interacts with the IItem interface can instantiate any of the subclasses and use all of the interface defined methods successfully and safely.

The ISpecialItem hierarchy, which was not proposed in the Design Document, but added during production to represent top pick and wishlist items, also ensures that the Liskov Substitution principle has been correctly applied. This is because any object that interacts with the interface ISpecialItem can instantiate any of the concrete subclasses without violating any of the superclasses policies and methods. An abstract class, SpecialItem, is used to implement the interface ISpecialItem. This abstract class provides concrete implementation for common methods for all special item types, and abstractions are created for subclass specific methods so that child classes provide subclass specific implementation of these methods. The special items that do not actually provide functionality for the abstract methods will throw an exception. This means that the Liskov Substitution principle is achieved as any class that interacts with the ISpecialItem interface can instantiate any of the subclasses and use all of the interface defined methods successfully and safely.

2.4 Interface Segregation principle

The Interface Segregation Principle has not been violated, in the final version of the application as there is no interface in which a generic interface is used for multiple classes. There is also no interface which implements another interface due to size and requirements of the project, we did not believe that this would be needed for code maintainability in potential future development so the principle was never considered. The interfaces used are ISpecialItem and IItem where ISpecialItem is a new interface added to manage WishlistItem and TopPick classes which were added through consideration of necessity during the development phase. Both these interfaces have been implemented by their own abstract classes SpecialItem and Item respectively.

2.5 Dependency Inversion principle

The Dependency Inversion principle has been properly applied and realised in our project as proposed in the Design Document, as well as through components added in production. This can be seen throughout our code, as where possible, no class is associated with concrete classes, only to their interfaces.

As proposed in our Design Document, the Dependency Inversion principle has been applied as any class that relies on items is not associated with a concrete class of type Item, but is instead associated with the interface IItem. All classes that are associated with items are associated with the interface IItem, which is implemented by the abstract class Item, which in turn is extended by subclasses of specific item types, Song, Podcast and Album. A key example of the use of Dependency Inversion with items can be seen in the class ItemAdaptor. The class does not rely on any concrete type of Item, only the interface IItem. This allows us to utilise dependency injection to ensure that the components are loosely coupled and that we can display any type of item on the ListActivity screen. The Dependency Inversion principle has been achieved as any class that relies on an item is not associated with a concrete class of that type, but is instead associated with the interface IItem.

The ISpecialItem hierarchy, which was not proposed in the Design Document, but added during production to represent top pick and wishlist items, also ensures that the Liskov Substitution principle has been correctly applied. During production, we added a new hierarchy to represent top pick and wishlist items, ISpecialItem. This hierarchy also allows for the Dependency Inversion principle to be applied, as any class that relies on special items is not associated with a concrete class of type SpecialItem, but is instead associated with the interface ISpecialItem. All classes that are associated with special items are associated with the interface ISpecialItem, which is implemented by the abstract class SpecialItem, which in turn is extended by subclasses of specific item types, WishlistItem and TopPick. A key example of the use of Dependency Inversion with special items can be seen in the class WishlistAdaptor. The class does not rely on any concrete type of SpecialItem, only the interface ISpecialItem. This decouples the objects, promoting a more robust design. The Dependency Inversion principle has been achieved as any class that relies on a special item is not associated with a concrete class of that type, but is instead associated with the interface ISpecialItem.

3.0 Good Coding Practices

3.1 Consistent naming conventions

Consistent naming conventions were followed, to ensure better communication, ease of reviewing and management. This allowed us to understand the intention of other members' responsibilities in the development process, progressing quickly without misinterpretation.

To enforce consistent naming conventions, all field and method names were in camelCase, while class constructor names were in Pascalcase. Each of these elements had searchable names, consistent with their specific functionality or meaning.

3.2 Avoiding duplicate code

Initially, in the early implementations of the application, unexpected functionality caused duplicate code with minor variations, littering the repository. To mitigate this, we created new classes to avoid code duplication. For example, we created the ActivityHelper class which manages items added to the Wishlist and TopPicks from multiple instances and scenes within the application. Initially, similar code was placed in the different Activities namely, DetailsActivity and ListActivity, decreasing maintainability.

Adaptors were used to help display specific items in each view using a set layout which decreased code duplication. Though this was a requirement in the brief, using an adaptor to display specific information in each of the activities using ViewGroups like RecyclerView allowed for dynamic display of items rather than manual coding and impractical implementations.

3.3 Commenting

Our project follows good and consistent commenting conventions. We were a bit slow to start adding comments to the different sections, but after we had completed the majority of the code base, during a team meeting we agreed on a style to all follow for our commenting and documentation. The style involved including the authors for each class, and outlining the responsibility of each class. For each method we outlined its responsibility at the top, mentioned if it was an overridden method, added explanations for each parameter, and any values that were returned.

Due to our descriptive naming conventions we did not need as many comments in the methods themselves, as we designed variable and method names so that the code explains itself. For any section that was harder to follow despite the informative naming conventions used, we added in single line comments which explained the code.

3.4 Proper use of packages

Our design also achieved good coding practice through the proper use of packages, as all classes that have similar functionality are grouped into the same package. For example, all of our Activity classes are packaged into the Activities package and any classes that aid the functionality of these Activity classes are placed inside the Helper package, which is a subpackage of the Activities package. All of the Activity classes are inside the Activities package, all of the adaptor classes are inside the Adaptors package, and all of the data classes used for Firebase are inside the Models package. Since all classes that have similar functionality and work together are in the same package, our project ensures proper use of packages.

3.5 Branching

In the initial stages of development, branching was not used and commits to main were the primary method to implement changes. This was due to misinformation of expected practices, however, after finding out the requirements for us as developers, we immediately changed to using branches. This allowed for better version control across all implementation stages.