University of Waterloo

Faculty of Mathematics

2B Work Term Report

Comparison Between Architectures and levels of abstraction in Android App Development

theScore

Position: Android Developer

Toronto, ON

Written by:

Fahad Adnan

20838180

2B Honors Computer Science

July 2021

Memorandum

To: PD11 Evaluators

From: Fahad Adnan

Date: July 7, 2021

Re: Comparison between architectures and levels of abstraction in android app development

---

This is my second work term report for my 2B co-op placement at theScore from May-June 2021 as a Mobile Intern in the Android Team. The work term report titled: “Comparison between architectures and levels of abstraction in android app development” is the second of four work term reports to be completed for my Bachelor of Computer Science degree.

The Android Team I was working within had internal teams for each of the two apps theScore offers being their media app and their betting app, along with a Growth team working on their media app. As an intern, my responsibilities consisted of completing tickets regarding bug fixes, feature work, and maintenance of the app involving implementing new technologies such as Android Jetpack Navigation and working with tools such as GraphQL. My responsibilities also included working with other teams including the Design and Analytics Team to resolve issues when completing tickets.

The purpose of this report is to examine popular architectures currently used and examine the implementation in theScore apps to determine benefits and issues with their current architecture setup. The report was written entirely by me using cited sources and interviews with several employees regarding the app architecture, thank you all for your insight when writing this report.

From,

Fahad Adnan

20838180

Table Of Contents

Executive Summary

The purpose of this report is to compare android app architecture patterns used in the past to the MVVM implementation in theScore and analyze the benefits and losses of their architecture along with examining new technology related to architecture such as Jetpack Compose.

In the past the primary language used by developers was the default MVC architecture as it was a popular and easy to integrate choice (Model-View-Controller, 2020). As time went on many developers noticed issues within the architecture including lack of modularity and poor handling of the lifecycle which brought upon further development and new forms of architecture. This created new iterations of the architecture giving us MVP (model-view-presenter), and a very commonly used architecture MVVM(model-view-viewmodel) along with many others (Emmax, 2017). Android developers at theScore took advantage of new technology and rebuilt their legacy media app built with Java to use modern technologies and practices available in Android such as Android Jetpack and MVVM architecture.

The report compares the many different types of architectures in the context of Android development analyzing what issued created the introduction of the new types of architecture while examining how theScore handles their implementation of MVVM compared to previous architecture and the MVI(model-view-intent) architecture I used in my previous co-op.

The report finds that with the plethora of advantages including the increase in modularity and with the introduction of Android Jetpack, MVVM and MVI are very good choices in modern android development. It also finds that theScore architecture implementation, although initially difficult to get adjusted to for new employees, provides many benefits in terms of reusing code and keeping the project modular, and allowing 100% line coverage to be achieved on the projects. It also finds that although the architecture is not a necessity for smaller personal projects, large projects with multiple contributors should consider architecture when starting their projects.

Introduction

1.1 Company Background

theScore has been a leader in the social media for sports setting having 10 million users on their media app along with their present venture into sports betting. I worked both on theScore media app and the upcoming theScore Bet app being released in Canada upon the legalization of single-event betting which was announced in Bill C-128 which is present in some US states such as Colorado and Tennessee (Bloomberg, 2020). Throughout my time as an Android Developer, I have worked with many types of architectures including MVC (model-view-container), MVI (model-view-intent), and MVVM(model-view-viewmodel), however, the architecture I have been using for the past four months at theScore although classified as MVVM is an interesting implementation which I will go further into within this report.

1.2 Role of Architecture

Architecture in a coding interpretation is a way of structuring code to be modular, specifically that there is a separation of concerns between the logic in your app so code related to one specific aspect of the app is handled in one portion of the app (Architecture, 2021). The architecture you use can play a large role in the implementation of your project as a major concern many don't think about when starting a project independently is setting up a testing suite for your project. Ideally, you want to write tests that aren't coupled to other features of the app so having a separation of logic meaning that you can create an environment for isolating and testing one portion of the app. This becomes increasingly important as an app size grows with more contributors and can be achieved through good architecture principles such as separation of concerns (Architecture, 2021). Separation of Concerns is a computer science design principle for separating a project into different sections where each section addresses a different concern(Wikipedia Foundation 1, 2021). In android specifically, concerns can include handling user input, handling database requests, and updating the user interface.

Analysis

2.1 - Outdated Yet Powerful Architecture for Android

2.1.1 - MVC (Model View Controller)

Prefacing the adoption and usage of MVVM the popular choice of architecture was model-view-presenter and prefacing that was the widely used model-view-controller architecture (Emmax, 2017).

MVC splits your application into 3 sets of concerns being the model, the view, and the controller (Wikimedia Foundation 4, 2021). The **Model** handles all the data-related logic needed for the program including making API calls and data stored locally on a user's device (Point, 2016). The **View** takes the information from the model and displays it in the UI (user interface); views are supposed to be kept as simple as possible and their main concern is rendering the UI and registering user input (Emmax, 2017). Views take register user input such as when they click a button and send this to the **Controller** which handles all the business logic (Model-View-Controller, 2020). The model has no knowledge of the View and vice-versa where the Controller is the portion that interacts with both of them to pass information from the view to the model, and the model will send information to the controller to update the UI (Model-View-Controller, 2020).

2.1.2 MVC Example

An original example of mine regarding the relationship with these three components is as follows. In an app a button click of the login button would trigger a click which would be sent to the controller along with the appropriate information (e.g username and login) and the controller would call a function to handle logging in with those credentials in the model. The model would access internal/external data and return to the controller that there was a successful login or some error code. With the response from the Model, the controller either navigates the user to the home page or shows a relevant error message in the View. This example summarizes how MVC is used in practice.

2.1.3 - MVP (Model View Presenter)

The prevalent issues with MVC brought upon the MVP architecture. The main issue with MVC was that it became difficult to test the controller due to the high coupling with the View, specifically the usage of Android APIs in the controller (Emmax, 2017). The controller would have access to the view hierarchy so that it could update them, but this makes it harder to unit test and any changes to the view would require changes from the controller as it directly updates the view (Wikimedia Foundation 2, 2021). With these problems, MVP came to fruition where the main difference is that now the controller would not update the view but instead there would be an interface between the two so that the controller didn't need to know about the android specific views (e.g TextView, Button, EditText) (Geeks, 2020). The model is much easier to test as it is modular, but the controller is directly coupled to the view meaning that it becomes much harder to test so separating the updating logic for views into the **View** which has direct access to the UI elements makes more sense (Emmax, 2017). The distinguishing difference in naming is that the controller was not renamed to the presenter, otherwise, MVP is based on MVC with the following changes

1) The presenter (previously called controller) no longer directly updates or knows about the views
2) There is an interface in which the presenter sends the change which is translated by the view
3) These translated changes are used to update the UI in the View

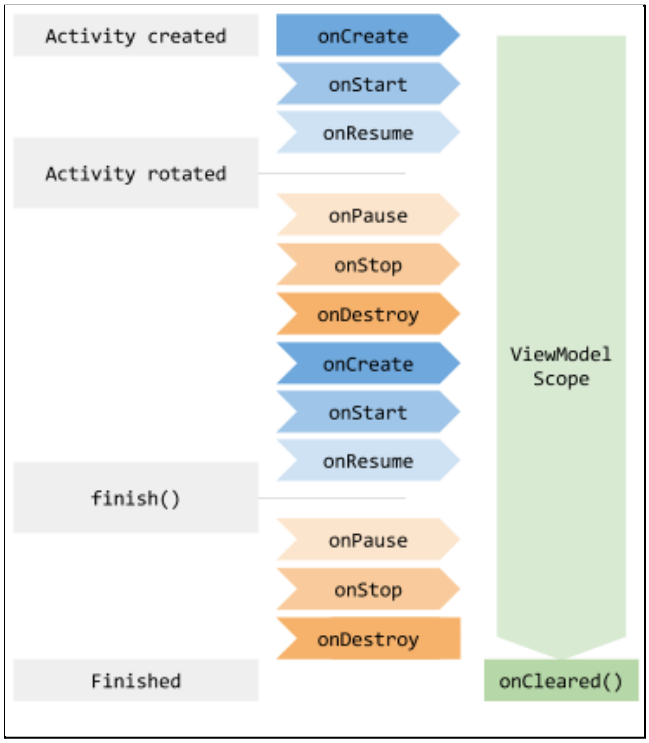
The MVP architecture was used extensively and was overall very good, but with the introduction of Android Jetpack and lifecycle aware components MVVM became the new standard as it resolved issues of the unknown state of the app and added new features such as shared presenters (called viewmodels in MVVM) (Android, 2021).

2.2 - Android Lifecycle affecting Architecture

Android UI controllers all have their individual life cycles including activities, fragments, and services (Fragment, 2020). The issue with this is that if activities or fragments store some data when they are destroyed only a small amount of information can be serialized, stored, and regenerated by Android via onSaveInstanceState (SaveInstance, 2021). An example is a screen rotation which would destroy the activity and recreate it, with this having a large amount of data that is needed in that activity/fragment may lead to information being lost. This is why we typically have the View portion not store the data but instead have the middle layer between the Model and View store the information for the fragmented state whether it be the controller, presenter, or viewmodel (Architecture, 2021).

A benefit of having an independent lifecycle from the UI controller is that if an activity is destroyed and recreated there is the possibility of memory leaks with network/thread calls that aren't cleaned up, so having a middle layer that doesn't follow the lifecycle of the view makes coroutine calls cleaner. There are fewer calls as the controller/presenter/viewmodel doesn't get recreated in rotations and cleaning up coroutine calls that would cause memory leaks also becomes easier (Kotlin, 2020).

2.3 Android Jetpack and ViewModel



Android Jetpack was released in 2018 as a suite of libraries and with its release included the widely used ViewModel library. This simplified implementing architecture as the middle layer lifecycle was handled by the library getting created on the first onCreate and removing itself after the UI controller is completely destroyed (Architecture, 2021). ViewModel objects are designed to outlive the lifecycle of the view; they are kept in memory until the UI controller lifecycle they are scoped to is destroyed permanently, so this creates low coupling to the View(activity/fragment) making it easier to unit test (Android 2021).

The image to the left shows the lifecycle of an activity as it rotated and the associated viewmodel lifecycle from (Android, 2021). Notice how its lifecycle is only affected when the activity is initially created and destroyed so storing information in the viewmodel is clearly better as it persists longer and doesn't need to be re-fetched (Android, 2021).

2.3.1 Databinding

Android Jetpack also released the data-binding library which allows you to set a variable typically located in your viewmodel that will track the view so that when the view changes the associated variable in the viewmodel will change, and when the viewmodel variable changes so will the view's value (Databinding, 2020). There can be one-way data binding or two-way, but regardless it is a very useful feature that Android now supports for the Viewmodel library that can help implement MVVM.

2.3.2 Sharing Data with ViewModel

Another powerful aspect of viewmodel is that they can be shared between fragments. In my experience, there are usually instances where multiple fragments shown on the screen have some shared data between them. So with a shared view model, you can use the same viewmodel for multiple fragments where communication is handled by the activity scope (Android, 2021). There are many benefits to sharing data as it makes the app run faster and the fragments don't need to know about each other but instead only about the SharedViewModel that's attached to them (Android, 2021)

2.4 MVVM(Model-View-Viewmodel)

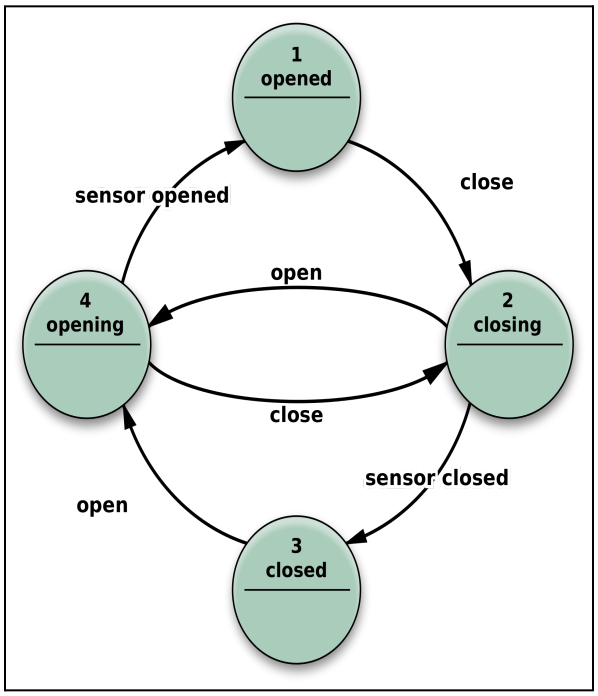
This architecture design is similar to MVP but with a few changes, first off this architecture is more event-driven as it uses data binding and tends to be more popular in recent years for Android development (Android, 2021). In this architecture the view binds to observable variables and functions exposed in the viewmodel via data-binding, in MVP there was an interface and a back and forth but as explained before data-binding makes this much easier (Emmax, 2017). The data-binding library can be used for implementing MVVM but the lifecycle library from Android Jetpack also gives you the ability to observe elements in the viewmodel (Lifecycle 2021).

This way there is a two-way data binding between the view and viewmodel. The viewmodel is the renamed presenter but now it provides observable data needed by the view, but it is not tied to the view like it was in MVC (Model-View-Viewmodel, 2021). With this new architecture, you essentially have no dependency on the view and this makes it much easier to test.

2.5 MVI(Model-View-Intent)

I used this architecture in my previous co-op at Tacit Corporation. As they were a SaaS company they were always updating technologies used on new projects, including architecture. MVI is an event-driven architecture where the concern of each portion is split up differently.

In MVI the model is responsible for retrieving/storing data acting as a backend to the app being a gateway to the API usage and database just like MVVM. The difference is that the model is also responsible for storing the state of the app, where a state could be the state of a button, a loading state, or any other item (Ray, 2019). In MVI the model is the single source of truth, there is only a single observable to manage when updating values. This is important as before where other portions could be altering the state now there is only the model which alters the state of the screen making it easier to test.



2.5.1 Finite State Machine

The main concept that inspired MVVM is the finite state machine. In a FSM interactions move you to different states and the main idea regarding this is so that there is a number of known states you can test as shown in the image to the left (Wikimedia Foundation 3, 2021). If you want to test a certain situation in a unit test all you would need to do is set the state of the viewmodel and you could recreate the situation, you could also know at which state a crash occurs making it easier to test (Ray, 2019), with multiple aspects affecting the state, such as in other architecture, this becomes much harder.

2.5.2 MVI Updating States

In my experience, the implementation of MVVM and MVI are similar as they both use the viewmodel and databinding libraries, but the difference is the cyclical flow of MVI and the single source of truth. MVI states are immutable so there is a cyclical flow between the three components: The view observes user actions and notifies the viewmodel which runs the appropriate commands that trigger some business logic updating to a new state, and that new state is used to update the view (Ray, 2019). It may seem that updating the entire view is going to add some overhead but when updating the view, the previous state is used so that only the changed aspects need to be updated (Ray, 2019).

2.6 Abstraction with architecture in theScore

Starting off as an Android Developer at theScore the codebase was initially overwhelming as although their codebase was an implementation of MVVM the level of abstraction was very high. For reference, I was not familiar with Single Activity architecture at the time so seeing a single fragment being used was quite confusing. Single Activity Architecture is the architecture design of using a single activity with fragments that swap in and out of that activity. Activities are simply holders for fragments, so recreating a new activity when you go onto a different page could be avoided.

2.6.1 Issues with theScore Bet Navigation and moving to Single Activity

Once I joined the betting app team I noticed that the codebase didn't use the new Navigation library released by Android Jetpack, and an issue came up with fragment navigation regarding the backstack. The fragment backstack is a list of fragments you have been to and its primary purpose is so that when you click back on the app or on your device you can go back to the previous fragment you were on (Fragment, 2021).

The issue was a circular backstack issue which could be described by 3 fragments (FA, FB, FC).
You start on FA, navigate to FB, navigate to FC, then navigate to FA from FC, you can continue this process indefinitely which would make a messy navigation issue for the user. It could possibly crash the

app if it goes on long enough, otherwise, it would just frustrated users who click back and cycle. The navigation library gives a cleaner interface for navigation and provides functions such as popUpToInclusive which would look at the previous backstack and remove elements accordingly so that you don't have circular issues (Navigation 2021).

Say you navigate from FA to FB to FC and then back to FA, popUpTo would remove FB and FC leaving two instances of FA and popUpToInclusive would remove FA, FB, and FC leaving one instance of FA (Navigation 2021). Another issue was the bundle logic in which information sent to fragments when creating an instance of them was disorganized and hard to manage. I resolved these issues by implementing navigation and SafeArgs to pass data to fragments and altering previous routing logic to use navigation graphs altogether making a cleaner navigation structure.

The overarching plan is to merge the MainActivity and AccountActivity so apart from a few small activities the vast majority of the logic would be contained in a single activity architecture.

2.6.2 Single Fragment Implementation in theScore Media App

A completely new structure I saw when working at theScore was that they were using a single fragment. This was quite strange to me as typically an XML file is correlated to one specific fragment, this way you can update the views accordingly. The reason they are able to have one fragment is that the fragment has view-holders that handle all the UI logic and was created depending on the data sent in. For most apps I have worked with, a fragment is a collection of lists and recycler views with the exception of a top and bottom bar. Instead of making new fragments with new adapters, you could just have a general fragment with a few general adapters and specific viewholders depending on the data.

When you start the fragment you can include extra information, in theScore, a config (configuration class) was sent in with information about the page including the type of viewmodel required. With the config, the appropriate view model is attached according to the page, along with

adapters, where there are mappings from the config to these generalized values. Following MVVM data would be taken from the viewmodel and you would use databinding to attach it to the specific variables, but instead here to make it more modular the adapter observes one variable which is a list of items. Similar to MVI there is mainly one observable value instead of a large number of observers, but holding true to MVVM the data being observed are a list of data and not the states of the fragment/items. Then they override the typical onCreateViewHolder method to add their own view-holder based on the type of data, also as this function takes a click listener they are able to separate the user event logic from the fragment.

From this implementation they get a greater degree of separation of concerns where the fragment is solely responsible for setting up the adapters, viewholders, viewmodels depending on the config and the individual viewholders are responsible for updating the UI and registering clicks. With this, there is another level of abstraction within the project as you don't need to know about the views that register their clicks. An interesting aspect to note is that the adapter sends in the viewmodel as an interface with one function to handle clicks so that you can register clicks in the viewholders and the business logic can be handled in the viewmodel.

2.6.3 Viewmodel Delegates in theScore Media Android Project

As a continuation of the use of abstraction in theScore Media app project, along with a single fragment being used, there are only a few viewmodels being used. The way they separate into specific functionality is ViewmodelDelegates. Depending on the config being sent into the fragment an appropriate view model delegate is attached. The difference between the implementation at theScore and traditional MVVM is that the viewmodel and viewmodel delegate separate the responsibility held by only the viewmodel in MVVM. The viewmodel primary purpose is to handle shared features such as showing empty states, but the viewmodel delegates are the separate portions that directly interact with the model and make their specific calls to get data, they also handle the specific clicks passed to them by the common viewmodel.

Looking over their architecture they adhere strongly to the separation of concerns creating more levels of abstraction which does cause difficulty adjusting for new developers but proves beneficial when testing.

2.7 Advantages and Disadvantages of theScore architecture

From the perspective of a new developer, the main disadvantage to this architecture is that there are so many levels of abstraction and without proper documentation, it is hard to adjust to the codebase for new developers who haven't worked with MVVM, however it provides many benefits.

The first of which I noticed was how the amount of boilerplate code was reduced, where boilerplate code refers to portions of repeated code present in multiple areas with little to no variation (Wikipedia Foundation 1, 2021). Through the singular fragment and usage of viewmodel delegates they have generalized many portions of their code such as adapters and methods to show empty screens, and now are able to create a greater degree of separation of concerns, decreasing the overall amount of code. Another major benefit was the ease of testability, one major reason architecture is developed in its specific way is to reduce coupling making it easier to test code. In theScore, every new pull request is accompanied by many tests so much that there is 100% test coverage on all lines via coverall.io, a tool that detects which lines are covered by a project test suite (Coveralls, 2021).

While many companies may not have a test suite theScore makes it their utmost priority to assure that their code is well tested before being pushed to consumers, note that the tests solely comprise of unit tests, commonly called UI tests which run on an AVD(Android Virtual Device) tend to be flaky so they aren't included. To explain the degree of testing, almost every kotlin file, excluding simple ones used for storing data classes, has an associated testing file.

Overall I prefer MVI due to its simplicity in comparison to the architecture w/ theScore app, but although there is a learning curve regarding this architecture it is more maintainable and easily testable once set up.

2.8 - Architecture for new developers

As more companies move their legacy java codebase to modern practices Kotlin and appropriate architecture are becoming more important for Android developers to learn. In my previous work term report, I mentioned that Java would be a good starting point due to the large number of examples that can help beginner Android developers but over the past year, Kotlin support from Google has increased significantly (Lardinois, 2019).

I would recommend that new developers shouldn't worry about architecture yet as there are many fundamentals to cover such as Activity and Fragment lifecycle, and how specific UI controllers interact together (Activities, Fragments, Services, etc). Once they have learned the basics they should learn about MVC and ideally learn either MVP, MVVM, or MVI. Although these architectures are designed for larger projects not necessary for most small side projects it's important to learn the principles about them and how they are implemented with tools such as ViewModels, LiveData, Databinding, etc (Jetpack, 2021).

For developers looking for internships or work experience, they should familiarize themselves with Android Jetpack as it has become an industry standard and an unspoken requirement for working with Android professionally over these past few years (Jetpack, 2021).

2.9 - New Developments for Android Development related to Architecture

2.9.1 - Jetpack Compose regarding theScore view bindings issue

Recently there was a crash regarding view bindings, where view bindings are a new library that generates getter and setters for views in xml files creating cleaner code (Binding, 2021). The issue was that a portion of the betting app was imported as a module into the media app and some files or classes had the same name; in this case, it was viewed in the XML files which had the same id. This caused issues as some areas of the app had binding while others didn't and at times we were getting the incorrect view. The method used to fix this was to add a prefix to all elements in that specific betting module so that the naming was different, although this was a janky fix this was an extremely rare issue with little support so it was the only viable fix.

A developer suggested jokingly switching over to Jetpack Compose, which would've taken months for a small fix, but it was a possible answer. With Android Jetpacks' new library, Jetpack Compose, you would no longer need to have an XML file of elements but instead have them declared directly in your Kotlin code making for faster loading elements that take much less code (Jetpack, 2021).

Compose like MVI uses the single source of truth principle as with composable objects state is explicit and passed to the composable meaning that changing the UI is done directly through the composable (Compose, 2021). Jetpack Compose also has a live preview meaning that if you want to see your items in a different state you can see them in real-time instead of having to install the app and getting to that item with its specific state, which accelerates development (Compose, 2021). Jetpack Compose is still receiving updates with the first release date being this July, so it is still a very new tool, but would be beneficial to implement in the future.

3.0 Conclusion

In conclusion, over time architecture in Android development has switched from using the infamous MVC pattern to MVP and currently the most popular MVVM and MVI architectures. theScore has a very interesting implementation of MVVM which provides benefits in testing, modularity, and accelerating the speed of production with the downfall being a learning curve for new developers. In my opinion, there should be more documentation regarding the project in order for developers to be able to get a better grasp of the project, although the initial walkthrough with an experienced developer was beneficial for me readable documentation available to reference would have been beneficial

For new android developers, they should focus on learning the language and core concepts, but once familiar with the basics should learn the popular architecture as it is proving to be an unspoken requirement in the workplace due to its widespread adoption. As time goes on and more support is provided by Android, interesting and powerful libraries such as Jetpack Compose are being released changing the way we create and manage our projects. As more developments come, the benefits in code organization and unit testing architecture become more important as a mobile developer.

References

Ray, W. (2019) · Article (20 mins) · Intermediate, & Olivares, A. (n.d.). *Mvi architecture for android tutorial: Getting started*. raywenderlich.com. https://www.raywenderlich.com/817602-mvi-architecture-for-android-tutorial-getting-started.

Android, D. (2021). *Viewmodel overview : Android developers*. Android Developers. https://developer.android.com/topic/libraries/architecture/viewmodel.

Architecture, A. (2021). *Guide to app architecture : Android developers*. Android Developers. https://developer.android.com/jetpack/guide?gclid=Cj0KCQjwvO2IBhCzARIsALw3ASptezcmio-eGBsBK_jE9--psj1bEYpOm5D5N44UmN3OXnklnrLd-s8aAiT7EALw_wcB&gclsrc=aw.ds.

Binding, A. (2021). *View binding : Android developers*. Android Developers. https://developer.android.com/topic/libraries/view-binding.

Bloomberg, B. (2020). *theScore Applauds Passage by the Senate of Historic Legislation to Legalize Single Event Sports Betting in Canada*. Bloomberg.com. https://www.bloomberg.com/press-releases/2021-06-22/thescore-applauds-passage-by-the-senate-of-historic-legislation-to-legalize-single-event-sports-betting-in-canada.

Compose, J. (2021). *Why compose : Jetpack compose : Android developers*. Android Developers. https://developer.android.com/jetpack/compose/why-adopt#less-code.

Compose, J. (n.d.). *Jetpack compose : Android developers*. Android Developers. https://developer.android.com/jetpack/compose?gclid=CjwKCAjw3_KIBhA2EiwAaAAlilj8_xSh5z5CSq9kOzi8KkTf1Ri1oN0gPbvARxP7fPjL_xptyCwURhoC820QAvD_BwE&gclsrc=aw.ds.

Coveralls, C. (2021). *Test coverage history and statistics*. Coveralls.io. https://coveralls.io/.

Databinding, A. (2020). *Data binding Library : Android developers*. Android Developers. https://developer.android.com/topic/libraries/data-binding.

Emmax. (2017). *MVC vs. MVP VS. MVVM on Android*. Realm Academy - Expert content from the mobile experts. https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/.

Fragment, A. (2021) *Fragment manager : Android developers*. Android Developers. (n.d.). https://developer.android.com/guide/fragments/fragmentmanager.

Fragment, A. (2020). *Fragment lifecycle : Android developers*. Android Developers. https://developer.android.com/guide/fragments/lifecycle.

Geeks, G. (2020, October 29). *MVP (model View presenter) architecture pattern in Android with example*. GeeksforGeeks. https://www.geeksforgeeks.org/mvp-model-view-presenter-architecture-pattern-in-android-with-example/

Jetpack, A. (2021). *Android Jetpack : Android developers*. Android Developers. https://developer.android.com/jetpack?gclid=CjwKCAjw3_KIBhA2EiwAaAAlim0QV4YS3sKaJwxcvDChOKPyW4_EEXQFMuiwlkv_hXJPgjLTn5CMehoCKXgQAvD_BwE&gclsrc=aw.ds.

Kotlin, A. (2020). *Use Kotlin coroutines with LIFECYCLE-AWARE components*. Android Developers. https://developer.android.com/topic/libraries/architecture/coroutines.

Lardinois, F. (2019, May 7). *Kotlin is now Google's preferred language for Android app development*. TechCrunch. https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development/.

Lifecycle, A. (2021). *Handling lifecycles with Lifecycle-Aware Components*. Android Developers. https://developer.android.com/topic/libraries/architecture/lifecycle.

Model-View-Controller, A. (n.d.).*MVC (model VIEW Controller) architecture pattern in Android with example*. GeeksforGeeks. (2020, October 27). https://www.geeksforgeeks.org/mvc-model-view-controller-architecture-pattern-in-android-with-example/

Model-View-Viewmodel, W. (2021, June 15). *Model–view–viewmodel*. Wikipedia. https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel.

Navigation, A. (2021) *Navigate to a destination : Android developers*. Android Developers. (n.d.). https://developer.android.com/guide/navigation/navigation-navigate#pop.

Point, T. (2016). Mvc framework - introduction. https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm.

SaveInstance, A. (2021). *Understand the Activity Lifecycle : Android developers*. Android Developers. https://developer.android.com/guide/components/activities/activity-lifecycle.

Wikimedia Foundation 1. (2021, June 1). *Separation of concerns*. Wikipedia. https://en.wikipedia.org/wiki/Separation_of_concerns

Wikimedia Foundation 2. (2021, April 25). *Model–view–presenter*. Wikipedia. https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter.

Wikimedia Foundation 3. (2021, July 30). *Finite-state machine*. Wikipedia. https://en.wikipedia.org/wiki/Finite-state_machine.

Wikimedia Foundation 4. (2021, June 28). *Model–view–controller*. Wikipedia. https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

Acknowledgements

I would like to thank my mentors Edward and Vitaliy for providing me with a challenging and engaging work environment where I was able to make an impact on the codebase. I would also like to thank the developers on the betting platform Jai, Dariush, Rod, and Manas who aided me in my learning throughout the term along with guiding me in enforcing better code quality through my pull requests. I would also like to thank all the developers on theScore Media app with special thanks to Waseem, Jeffy, and Jordan on that team for hopping on calls to explain issues in my PRs and helping me with my report. Thanks to Mynkie, Timothy, Rubel, and David, Tim on different teams (Analytics, Design, QA, iOS, Project Management) for being great co-workers.