A Clean Approach to Flutter Development through the Flutter Clean Architecture Package

Shady Boukhary*, Eduardo Colmenares†
Department of Computer Science, Midwestern State University
Wichita Falls, TX
Email: *sb199898.sb@gmail.com †eduardo.colmenares@msutexas.edu

***Abstract*—Flutter is a popular UI framework for developing mobile applications by Google. It has caught traction in recent years. However, Flutter developers have to deal with a state management issue when developing their applications. In order to solve this problem, multiple architectues have been developed. This paper proposes a new Flutter architecture based on the Clean Architecture by Uncle Bob. The Flutter Clean Architecture proposed in this paper is packaged and released through a Flutter package. The architecture is tested by developing a full application from scratch using the package and documenting the process. The Flutter Clean Architecture provides a solution to the state management problem as well as a potential overall choice for Flutter mobile application architecture.**

***Index Terms*—Flutter Clean Architecture, State Management, Dart, Dependency**

I. Introduction

This paper aims to propose, test, and evaluate an architectural model and a state management technique for Google's *Flutter*, an open-source iOS and Android mobile application development framework. Flutter's first release was on Tuesday, December 4, 2018 [6]. It has been growing ever since, gaining over 65,000 stars and 7,000 forks on GitHub [5].

Instead of attempting to replace native iOS and Android applications, Flutter introduces a new way of developing applications, allowing developers to write their applications entirely using Flutter or embedding them into existing native applications [6]. This allows developers to write native iOS and Android code if needed, removing framework specific limitations when it comes to native functionality. Flutter is powered by the Skia 2D graphics engine [6]. In addition, it is powered by Dart, an object-oriented, class-defined, garbage-collected lanugage that can be compiled ahead-of-time (AOT) to native 32-bit and 64-bit ARM code for iOS and Android [4]. Optionally, it can also be transpiled to JavaScript for the web [3].

II. The State Management Problem

Flutter is composed mainly of widgets [10], which are the basic building blocks of all Flutter applications. A widget can either be stateful or stateless. Stateless widgets have no internal state that can be modified after the widget has been built. On the other hand, stateful widgets can be dynamically modified by modifying their internal state without reinitialization. Stateful widgets are important when building any interactive application in Flutter. The Flutter framework creates states for its widgets that can be read synchronously when the widget is built and can be modified throughout the lifecycle of the application [10]. Since Flutter's UI is a widget tree, setting the state of a parent would re-render all of its children, which has performance drawbacks. In addition, it is difficult to render a specific child when an event occurs in another child without rendering all children using the parent [10]. Without a proper architecture, managing widget states is very difficult and unmaintainable. There are many existing architectues that tackle this problem that vary in efficiency and maintainability [8]. This paper proposes an architectues through a package that solves the state management problem and provides an overall architecture for Flutter applications.

III. Flutter Clean Architecture

The Flutter Clean Architecture provides a method by which to both structure an application architecturally and to solve the State Management Problem - which was be explored in section II - through a Pub Fluter package available to developers [1]. The architecture is influenced by The Clean Code Blog [11], authored by Robert C. Martin - one of the authors of the Agile Manifesto [7]. The architecture aims to provide a concrete and adapted implementation of the Clean Architecture for Flutter.

The fundamental purpose of the architecture is separation of concerns and scalability. By dividing the system into layers that separate business logic from platform-specific implementation, three crucial principles of Software Engineering are maintained:

A. Framework Agnosticism

With a layered architectural model, the mobile application being developed can be framework-agnostic. The business logic of the application does not depend on external libraries or frameworks. In the case of Flutter, the business layers of the application are written purely in Dart - without knowledge of the existance of said logic within a Flutter application.

Writing framework-agnostic business logic greatly improves portability, as Dart code can be reused in a web application and transpiled to JavaScript [3]. Even in extreme cases as abandoning Flutter in favor of another framework such as React Native, the framework-agnostic business logic would merely need to be translated into JavaScript, enabling the reuse of both the code and the logical flow of the application.

978-1-7281-5584-5/19/$31.00 ©2019 IEEE
DOI 10.1109/CSCI49370.2019.00211

In addition, with a framework-agnostic architectural model, frameworks can be simply used as tools, rather than existing at the center of the application because the business logic of the application will not include any framework-related code. Rather, the application does not expose the framework to the business logic layers, providing only interfaces and abstract classes. Maintaining a pure Dart layer in th application would safegaurd it from frequent changes that occur on the framework level.

B. Testability

The Clean Architecture for Flutter should also be highly testable. Due to the complete separation of business logic from the User Interface (UI), Database, and other elements, testing becomes effortless as business rules and UI can be tested independently as shown in section IV. When testing inner layers, dependencies can be easily mocked by inhertiting from the abstract classes which are injected into the inner layers. This enables isolated and reliable testing of business rules regardless of the implementation. For example, authentication-related rules can be tested in isolation from the actual implementation of authentication, whether through an SDK such as AWS Mobile or an HTTP call to an API.

C. Isolation from External Modules

In the Clean Architecture, the business layer is at the center of the architecture. The business layer is isolated from the implementation. Therefore, the UI, Database, Web Server, and other elements are outer modules of the application and do not impact the business logic. One major advantage of isolation frome external modules is the ability to swap modules with others without impacting the business logic. For example, an HTTP module that calls an API to retrieve data from a Database on a server can be swapped with a local Database such as SQLite or an SDK-exposed Database such as Firebase, which would be a far cry from traditional HTTP calls to an API. In the Clean Architecture, outer modules or layers can be swapped without affecting the flow of the application or the business logic.

Likewise, the UI has no impact on the business logic, as it exists within its own layer. Changes in the UI layer do not affect the flow or the business logic of the application, making it much more maintainable.

As shown in Fig. 1, the Clean Architecture is structured like an onion. The inner layers are comprised of more abstract parts, while the outer layers are more verbose and implementation-specific.

For example, the outermost layer is the Data and Platform layer. This layer is comprised of platform-specific implementations of the enterprise and business logic. In the context of a mobile application, the Data and Platform layer would include the GUI, classes that interact with the iOS or Android platform such as GPS, and classes that interact with databases locally and through the web. The concrete implementations of interactions between the application and the platform/web are present in classes called repositores. The repositories implement interfaces present in the Domain layer also called

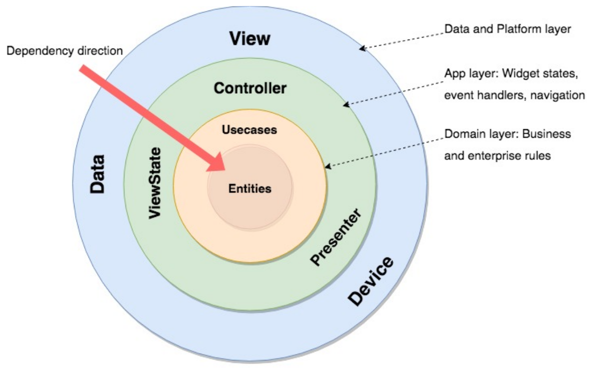

Fig. 1. Flutter Clean Architecture Layer Diagram

repositores. The repositores in the Platform and Data layer are injected into the inner layers and used polymorphically in order to abide by the dependency rule. Repositories which interact with the web/databases are called Data repositores, while repositores which interact with the platform are called Device repositores. Futhermore, the Data and Platform layer also contains the GUI of the application. The implementation of this layer in the context of Flutter will be discussed in section IV. This layer is the most prone to change, as it contains very platform-specific code.

The next layer is the Application layer. The application layer consists of consists of Flutter widgets, Widget States, event handlers, view controllers, presenters, and navigation controllers. This layer is responsible for handling all user events issued from the GUI and redirecting them to their appropriate use cases in the inner layers. When an event is triggered in the GUI, the View, the controller handles it either by updating the UI or passing it to the presenter, which executes the appropriate usecase and prepares the result appropriately for the UI. This will be explained in more detail in the section IV. The navigation controllers are responsible for carrying data to be passed between different views. This layer is very prone to change. However, it is less likely to change than the Data and Platform layer.

The next layer is the Domain layer, which is in turn divided into two parts. The outer part of the Domain layer is the business rules layer, wherein the usecases for the application reside. The usescases for the application contain the main business logic for the application. In addition, it orchestrates the flow of data throughout the application. For example, if one usecase of the application is logging in, this layer should contain the logic for logging in. Every usecase present in the requirements document for any application should be implemented in this layer.The domain layer should not be aware of any outer layers and is not aware of the platform either. To explain, if the Clean Architecture is being used in a Flutter application, the Domain layer should not be aware of the existance of Flutter. The domain layer is implemented in pure Dart in that case. In the case of iOS and Android, it would be implemented in Swift and Java/Kotlin respectively. The domain layer cannot use any platform-specific libraries. However, it can use any language-specific ones such as RxDart

```
lib/
  app/                          <--- application layer
    pages/                        <-- pages or screens
      login/                        <-- some page in the app
        login_controller.dart         <-- login controller extends ‘Controller’
        login_presenter.dart          <-- login presenter extends ‘Presenter’
        login_view.dart               <-- login view
    widgets/                      <-- custom widgets
    utils/                        <-- utility functions/classes/constants
    navigator.dart                <-- optional application navigator
  data/                         <--- data layer
    repositories/                 <-- repositories
      data_auth_repo.dart           <-- example repo: handles all authentication
    helpers/                      <-- any helpers e.g. http helper
    constants.dart                <-- constants such as API keys, routes, urls, etc
  device/                       <--- device layer
    repositories/                 <--- repositories
    utils/                        <--- any utility classes/functions
  domain/                       <--- domain layer (business and enterprise)
    entities/                   <--- enterprise entities (core classes of the app)
      user.dart                   <-- example entity
      manager.dart                <-- example entity
    usecases/                   <--- business processes e.g. Login
      login_usecase.dart          <-- example usecase extends ‘UseCase’
    repositories/               <--- abstract classes that define functionality
  main.dart                     <--- entry point
```

Fig. 2. Folder Structure for Flutter Clean Architecture

for the Dart language. Along with usecases, this layer also contains repositories. These repositores outline the behavior needed by the usecases. The repositores are interfaces that do not contain any implementation. For example, an Authentication Repository would be needed by several usecases such as a login usecase, a logout usecase, a signup usecase, and so on. The Authentication Repository would act as an interface used by the repositores present in the outermost layer, the Data and Platform layer, which when implemented, would contain the code needed to login with the database used by the application. The purpose of the repositores is to outline the behavior needed by the usecases and to allow for dependency injection from the outermost layer into the business rules layer. The usecases would then be able to call the methods present in those repositores through polymorphism. This is the most crucial part of the architecture as it allows for a significant level of abstraction and separates platform-specifc and database-specific implementation from business logic. This layer is unlikely to change, as it is not affected by any UI or database changes. This layer should only be affected when the fundamental functionality of the application is changed. For example, if the application no longer requires the users to sign up, this layer would need to be modified.

The second portion of the Domain layer, which happens to be the innermost layer of the application, is the enterprise rules layer. This layer is comprised of enterprise entities that are used throughout the application. The entities only interact through usecases that connect them. An example of an entity would be a User class. Like the business layer, the enterprise layer cannot contain any platform-specific code. This layer is also unaware of any other layer. For example, a login usecase would instantiate a User entity if login is successful. This layer is the least prone to change, as even changes in the application functionality should not affect it. The only reason to modify this layer is if enterprise rules change. This entity can then be used by the application without being aware of the platform. Despite how tempting it might be, a User entity cannot retrive its own data from the database or interact with any other layer of any kind, as that would violate the dependency rule: inner layers are unaware of outer layers.

To reiterate, Source code dependencies only point inwards. This means inner layers are neither aware of nor dependent on outer layers. However, outer layers are both aware of and dependent on inner layers. Outer layers represent the concrete mechanisms by which the business rules and policies (inner layers) operate. The more one moves inward, the more abstraction is present. The outer one moves, the more concrete implementations are present. Inner layers are not aware of any classes, functions, names, libraries, etc.. present in the outer layers. They simply represent rules and are completely independent from the implementations. This rule is absolutely vital for the success of this architecture.

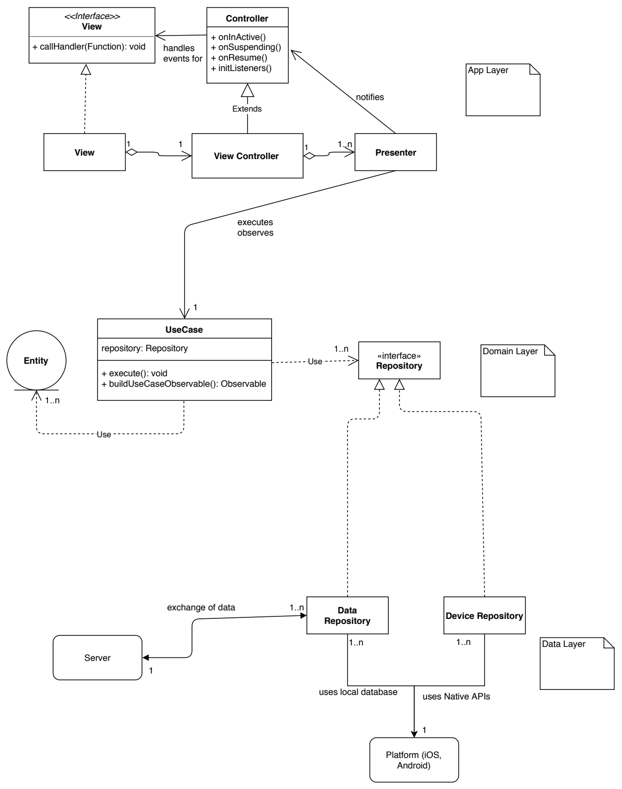

Fig. 3. Flutter Clean Architecture Flow Diagram

IV. Implementation

The Clean Architecture can be implemented differently on different platforms. This paper proposes a Flutter-specific implementation designed to fit the structure of Flutter and its inner workings. The folder structure for Flutter can be found in figure 2. In addition, the class diagram for the architecture can be found in figure 3.

A. Data and Platform Layer

The data layer is comprised of the data and device subdirectories. In this layer, classes which implement interfaces found in the domain layer are present.

B. Application Layer

The application layer is the most framework-dependent layer and is comprised of 3 basic types of classes provided by the Flutter Clean Architecture package [1]. The View class represents only the UI of the page. The View builds the pages's UI and styles it. It is injected with a Controller class that handles the its events through the constructor. The View has a Controller. There are 2 different types of View. The first is the View class, which is the root widget of the page representing the View. The second type is the ViewState class, which is a specialized template of the Controller corresponding to the view. The ViewState is a State. Therefore, it contains the build method required to buld the UI. The View is a StatefulWidget. Its purpose is to pass arguments to the ViewState and provide it with static data it needs, as well as inject it with the Controller. The View has a ViewState. To summarize, both the View and ViewState comprise the UI of the page. In addition, the ViewState class maintains a Global Key that can be used to access its root widget, which can then be later used by the Controller. This is helpful in order to display dialogs programatically.

The Controller provies the needed member data of the ViewState. The Controller also implements the event-handlers of the ViewState widgets, but has no access to the Widgets themselves. The ViewState uses the Controller, not the other way around. When the ViewState calls a handler from the Controller, it wraps it with a callHandler(fn) function if the ViewState needs to be rebuilt by calling setState() before calling the event-handler. The callHandler(fn) method will handle refreshing the state. [1] Every Controller extends the Controller abstract class, which implements WidgetsBindingObserver. Every Controller class is responsible for handling lifecycle events for the View and can override:

1) onInActive()
2) onPaused()
3) onResumed()
4) onSuspending()
5) onDidPop()
6) initListeners()

The Controller has a Presenter. The Controller will pass the Repository to the Presenter, which it communicate later with the Usecase. The Controller will specify what listeners the Presenter should call for all success and error events as mentioned previously. Only the Controller is allowed to obtain instances of a Repository from the Data or Device module in the outermost layer. The Controller has access to the ViewState and can refresh the UI via refreshUI(). Alternatively, handlers can be wrapped in callHandler() which automatically refreshes the UI after it's completed. [1]

The Presenter communicates with the Usecase as mentioned at the beginning of the App layer. The Presenter will have members that are functions, which are optionally set by the Controller and will be called if set upon the Usecase sending back data, completing, or erroring. The Presenter is comprised of two classes:

1) Presenter
   The presenter contains event handlers implemented inside the Controller and Usecases to be executed by the Presenter itself. The Presenter initializes and executes the usecase and provides it with an Observer¡T¿ that responds to either the Usecase's success or failure.
2) Observer
   Handles the finishing state of a Usecase. The Observer implements 3 functions: onNext(), OnComplete(), and onError(), which correspond to all possible states of a Usecase. If the UseCase returns data, it will passed to onNext(). The Observer notifies the Controller, which can then update the View.

Any other utility classes are present in their own subdirectory.

C. Domain Layer

As stated in section III, the Domain layer contains Entities, UseCases, and Repositories. Entities are simple classes, while Repositories are interfaces. On the other hand, UseCases extend the UseCase class provided by the package [1]. The UseCase is implemented with RxDart and is executed via its public execute() method. It notfies of its state through the Observer provided by the user. When extending the UseCase class, the user implements the business logic by overriding the buildUseCaseObservable() method [1]. The rest of the architecture is a matter of following the structure in figure 2.

V. Testing Methodology

In order to test the package, a complete application was built for the Hotter'n Hell yearly event in Wichita Falls, TX. Hotter'n Hell is a racing event in which thousands of people visit the city yearly to participate in bike races. The applications contains a lot of functionaly, including logging in, signing up, resetting password, registering for events, viewing events, paying for events, navigating to events, checking out sponsors for events, adding events to favorites, turn-by-turn navigation for races held at the event, and so on. The application was built using the Flutter Clean Architecture packaged developed by this paper's authors in order to test its efficiency during development. Flutter version 1.5 was used to develop the application along with version 1.0 of the Flutter Clean Architecture package. The application is over 25,000 lines of code and is available publicly on GitHub [1]. In addition, the application was tested on various devices including an iPhone 6, iPhone 8, iPhone X, Galaxy S10, Galaxy Note 5, Google Pixel 3, and Google Pixel 3 XL. The application was also tested on various iOS and Android versions, including iOS 9, iOS 10, iOS 11, Android 7, Android 8, and Android 9. Unit testing was used to test individual layers of the application. Unit tests were categorized into data tests, widget tests (application layer), and domain tests, which tested both the entities and the usecases. Usecase dependencies that are injected into the usecases were mocked using the Mockito package for Dart. The testing process is meant to test the ease of testing of the application through unit tests and acceptance tests - as well as the architecture's affect on work distribution and development speed. The SCRUM methodology was used to develop the application [9].

VI. Results

Using the Flutter Clean Architecture resulted in a relatively fast application development process of 2 and a half months. By taking advantage of its extreme modularity, different layers were able to be assigned to different programmers and programmed in parallel. In terms of development efficiency, the architecture was excellent at parallelizing the development process and ensuring the most effective development strategy was being used.

Using the architecture also allowed for easy detection of bugs. Because of the independent layers, bugs in the application were quickly narrowed down to their corresponding layer, where they were easily found by investigating the different components. For example, bugs in the application layer were quickly narrowed down to either the Controller or the View. In addition, due to the architecture's independence from databases, we were able to switch from SQL to MongoDB with ease, as the application was built around business rules rather than a specific database. Switching from SQL to MongoDB did not have an impact on any of the business logic of the application. Relatively minor changes in the Data layer had to be made.

In terms of testing, the architecture proved to be very effective. Unit tests were written separately for Entities, UseCases, Data, and Device implementations. Testing the UseCase classes independently by mocking their dependencies was very helpful in making sure the business logic was sound, regardless of whetehr the platform-specific implementations of the dependencies of the usecases were working or not, for those dependencies were mocked and injected into the usecases. To test the platform-specific implementations, tests for the Data layer were written separately from other tests. This allowed us to explicity target platform-specific implementations regardless of what the business logic was. Separation of such tests proved to be very effective in quickly identifying and squashing bugs found during the development process.

Futhermore, using this architecture instead of a more widely-used one such as MVC provided more effective and profound separation of concerns, which further helped separate business logic from platform-specific logic and amounted to easy testing. When tested on different platforms, no noticeable lag was present due to the added overhead and testers reported snappy performance.

VII. Conclusion

In conclusion, the implementation of the Clean Architecture through the use of the Flutter Clean Architecture package proved to be very effective and advantageous. Combined with ease of testing, the Clean Architecture sped up the development process by maximizing the potential for working in parallel with other programmers. The Flutter Clean Architecture is a solution to Flutter's State Management problem, while also being an option for a Flutter mobile application architecture. However, it is important to note that due to the added overhead of the architecture that is manifested through the large number of added classes and extra code, when it comes to small codebases that are not meant to be scaled by a large factor, other architectural choices are recommended.

References

[1] Boukhary, Shady. Flutter Clean Architecture Pub Package. April 30, 2019. Accessed on September 19, 2019. [Online]. Available: `https://pub.dev/packages/flutter_clean_architecture`

[2] Boukhary, Shady. Hotter'n Hell Mobile Application. May 28, 2019. Accessed on September 19, 2019. [Online]. Available: `https://github.com/ShadyBoukhary/Axion-Technologies-HnH`

[3] Google, Dart's official Website. Accessed on September 18, 2019. [Online]. Available: `https://dart.dev/`

[4] Google, Dart Language Tour. Accessed on September 18, 2019. [Online]. Available: `https://dart.dev/guides/language/language-tour#important-concepts`

[5] Flutter, Flutter GitHub Repository. May 7, 2019. Accessed on September 19, 2019. [Online]. Available:
`https://github.com/flutter/flutter`

[6] Google, Google Developers Blog. Accessed on September 20, 2019. [Online]. Available:
`https://developers.googleblog.com/2018/12/flutter-10-googles-portable-ui- toolkit.html`

[7] Laplante, Philip A. (2014). Requirements engineering for software and systems (2nd ed.). Boca Raton, Florida: CRC Press. p. 168. ISBN 9781466560819. Retrieved February 26, 2015.

[8] Flutter, List of State Management Approaches. Accessed on September 21, 2019. [Online]. Available:
`https://flutter.dev/docs/development/data-and -backend/state-mgmt/options`

[9] Sutherland, J. (2014). Scrum - The Art of Doing Twice the Work in Half the Time. Random House Business.

[10] Flutter, The official Flutter Website. Accessed on September 21, 2019. [Online]. Available:
`https://flutter.dev/docs`

[11] Uncle Bob, The Clean Coder Blog - The Clean Architecture. Accessed on July 16, 2019. [Online]. Available:
`https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html`