Tung Bui Duy

Reactive Programming and Clean Architecture in Android Development

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

27 April 2017

Helsinki Metropolia University of Applied Sciences

Author(s) Title Number of Pages	Tung Bui Duy Reactive programming and clean architecture in Android development 47 pages
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Peter Hjort, Principal Lecturer

Software application becomes more and more complex nowadays. To provide a good software application that is easy to scale, developers need to design a good software architecture.

The purpose of the project was to find a good Android architecture that can be used in later projects of the team C63-Studio. The project was implemented by refactoring a legacy Android application of team C63-Studio using clean architecture, dependency injection and reactive programming.

From the refactoring experience, the team realized advantages and disadvantages of applying clean architecture and reactive programming. While there are some disadvantages when applying this architecture for small applications, the advantages for applications that require scalability easily overwhelm the disadvantages. Based on the analysis, the C63 studio development team decided to use this architecture for later Android projects.

Keywords	Android, Reactive Programming, Clean Architecture, MVP, RxJava, Dagger2, Retrofit

Contents

1 Introduction

The trend of using a smartphone instead of a computer for daily tasks is growing. The complexity of tasks that mobile applications need to handle has also increased dramatically. To build a feature-rich mobile application becomes a demanding task.

However, developing a mobile application can easily go wrong if developers do not architect their application carefully. This thesis examines how to apply clean architecture and reactive programming to reduce the complexity of developing a mobile application.

The client in this project is C63 studio which is a start-up studio which focuses on developing software applications. It develops mobile games, mobile software as well as web applications. The studio looks for a solution that helps to reduce the cost and complexity of developing a mobile application.

The goal of the project is to find a good Android architecture that helps to ease development efforts in later projects. C63 Studio development team decides to pick an old application and refactor this application using a new architecture. After that, the team will add more instrumental tests and unit tests to see how easy it is to provide an application with a good test coverage.

2 Sunshine Application

Sunshine is a weather application that fetches data from Open Weather API and displays it to the users based on their location and their setting preferences. It is a simple but complete application. However, the app needs to migrate to get rid of deprecated methods and add some login features, as well as have a good test coverage.

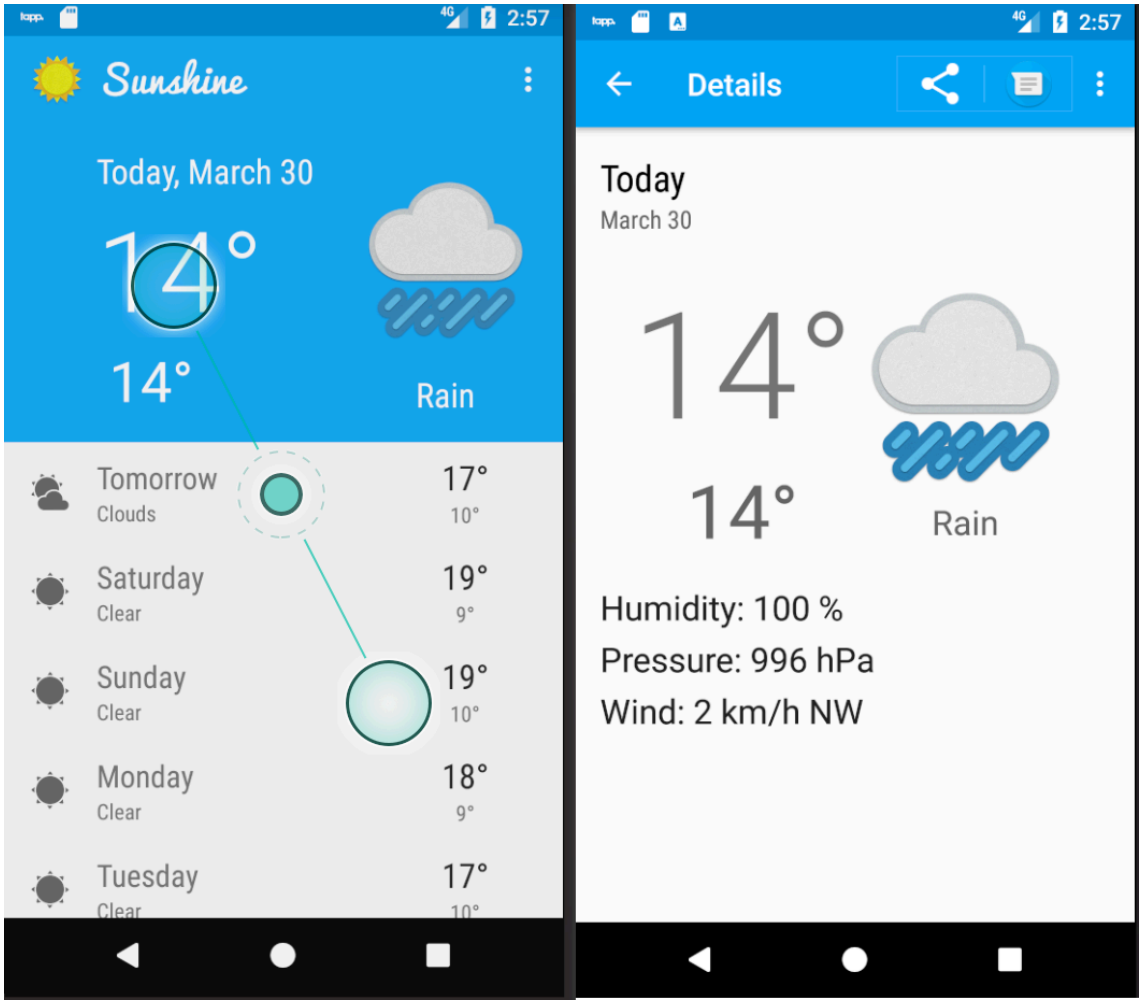

Figure 1. Main feature of Sunshine application

The Sunshine application was built based on Model - View -Controller architecture. As illustrated by figure 1, the application has two main screens: Home screen and detail screen. The weather data is saved into a sqlite database and synchronized with the server by using a sync adapter. The sqlite data is provided to the application through the content provider.

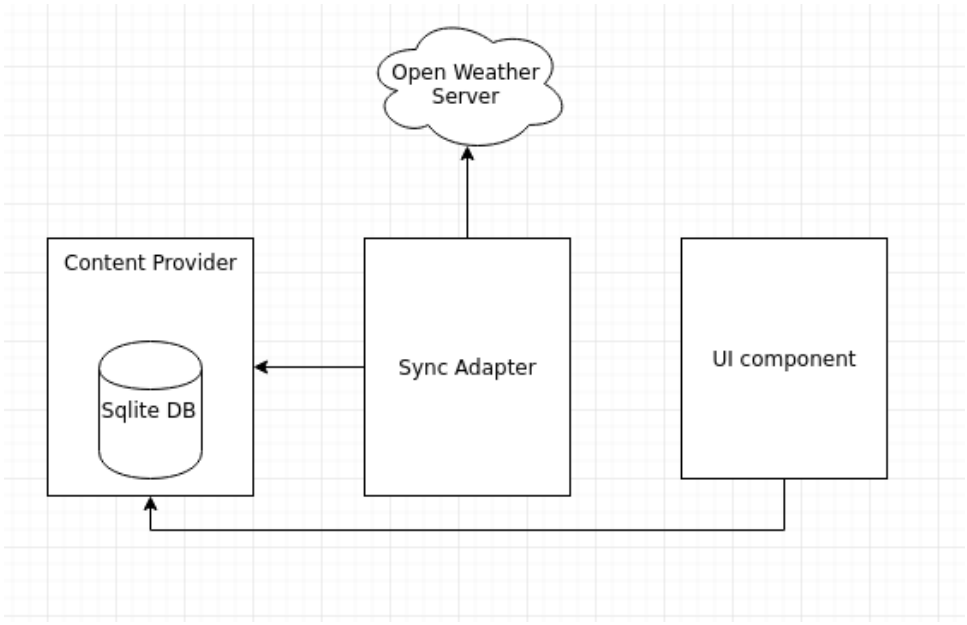

Figure 2. Overview of Sunshine application

Figure 2 provides an overview of how Sunshine application is architected. The application is well architected for its functionalities. However, there are some problems.

```
// For the forecast view we're showing only a small subset of the stored data.
// Specify the columns we need.
private static final String[] FORECAST_COLUMNS = {
        // In this case the id needs to be fully qualified with a table name, since
        // the content provider joins the location & weather tables in the background
        // (both have an _id column)
        // On the one hand, that's annoying.  On the other, you can search the weather table
        // using the location set by the user, which is only in the Location table.
        // So the convenience is worth it.
        WeatherContract.WeatherEntry.TABLE_NAME + "." + WeatherContract.WeatherEntry._ID,
        WeatherContract.WeatherEntry.COLUMN_DATE,
        WeatherContract.WeatherEntry.COLUMN_SHORT_DESC,
        WeatherContract.WeatherEntry.COLUMN_MAX_TEMP,
        WeatherContract.WeatherEntry.COLUMN_MIN_TEMP,
        WeatherContract.LocationEntry.COLUMN_LOCATION_SETTING,
        WeatherContract.WeatherEntry.COLUMN_WEATHER_ID,
        WeatherContract.LocationEntry.COLUMN_COORD_LAT,
        WeatherContract.LocationEntry.COLUMN_COORD_LONG
};
```

Listing 1. Forecast database detail exposed in Forecast Fragment

```
@Override
public Loader<Cursor> onCreateLoader(int i, Bundle bundle) {
    // This is called when a new Loader needs to be created.  This
    // fragment only uses one loader, so we don't care about checking the id.

    // To only show current and future dates, filter the query to return weather only for
    // dates after or including today.

    // Sort order:  Ascending, by date.
    String sortOrder = WeatherContract.WeatherEntry.COLUMN_DATE + " ASC";

    String locationSetting = Utility.getPreferredLocation(getActivity());
    Uri weatherForLocationUri = WeatherContract.WeatherEntry.buildWeatherLocationWithStartDate(
            locationSetting, System.currentTimeMillis());

    return new CursorLoader(getActivity(),
            weatherForLocationUri,
            FORECAST_COLUMNS,
            null,
            null,
            sortOrder);
}
```

Listing 2. How ForecastFragment fetched data

As shown by listing 1 and 2 as well as figure 2, the UI components needs to know implementation details of how data is stored and accessed in the application. While the application still functions properly, this kind of implementation introduces tight coupling issue. Changing in how data is stored and accessed requires changing in the user interface as well as changing in the user interface may require changing how data is accessed.

3 Architectural and Design Patterns

There are some commonly solved challenges when developers develop a highly maintainable application. These solutions are usually referred to as patterns: design patterns and architectural patterns.

3.1 Design Patterns

Design patterns provide proven solutions for occurring challenges that developers encounter every day when developing software applications.

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing at the same way twice" [5, 27].

Published in 1994, the book "Design Pattern: Elements of Reusable Object-Oriented Software" (the four authors, as well as the book are often referred to as Gang of Four - GoF) has impacted heavily how software applications are designed. As stated in the GoF, a design pattern needs to have at least four components: pattern name, problem, solution and consequence. The pattern name needs to be able to describe the design problems. The pattern name also helps to increase the effectiveness of communication between developers. Secondly, the design problem needs to explain its context as well as when to apply the pattern. The solution explains the design, relationship, responsibilities and collaborations. It needs to be abstract, no specific implementations are allowed. The consequences (results and trade-offs) are very critical for developer when they make decisions regarding applying this pattern for their problems.

Design patterns can be categorized into three categories: creational patterns, structural patterns and behavioral patterns. The creational patterns describe how to create objects. The structural patterns describe how to simplify the realization of relationship between objects/entities. The behavioral patterns describe how to assign responsibility between objects.

One of the main characteristics of highly scalable architecture is high cohesion and loose coupling. High cohesion means a software component should focus on doing

only one thing. Loose coupling means that a software component should not know about internal implementation of other components to form software. The façade pattern, observer pattern and dependency injection are often used to form software that favors high cohesion and loose coupling components.

3.1.1 Façade Pattern

Being a structural design pattern, the façade pattern wraps a complex system with a simpler interface and makes it easier to use by the client. The façade interface should not be implemented as a “god” object which knows too much or does too much. Instead, it should be a simple facilitator. It also needs to utilize composition.

While the façade pattern helps to simplify the usage of a complex system, by providing only one unified interface to the client, it might miss some features that a “power user” wants to have. Also, the system needs to be complex to utilize the power of the façade pattern

3.1.2 Observer Pattern

Dividing a system into cooperating classes makes maintaining consistency between related objects more challenging. Observer pattern, also known as a Publish-Subscribe pattern or the dependents pattern helps to define a dependency between objects so that when an object changes its state, its dependencies receive a notification and update automatically. [5,431-432].

There are two key components in observer patterns: observer (or subscriber) and provider (or subject). The provider is the object with many dependent objects, and the observer is the object that waits for a state changing notification from its provider. [5, 433]

The observer pattern is heavily used in UI development. It helps reduce complexity of maintaining consistency between the states of objects by abstracting coupling between them. It also provides support for broadcast communication. However, developers should use it carefully. Unexpected updates can occur since the provider does not know anything about its observer.

3.1.3 Dependency Injection

Dependency injection is a design pattern that implements the inversion of control principle (Inversion of Control: Don't call us, we will call you). Separating the creation of dependencies from the behavior makes testing an application much easier because objects no longer depend on anything one passed to them. Typically, dependency injection will have: a service which will be used by the client, an interface which defines how the client uses the service and an injector which constructs services and inject them into the client. [6]

Even though dependency injection reduces coupling between objects and increases the flexibility of being configurable, it may make an application become challenging when tracing problems because it separates behavior from constructor. It may also increase the complexity of the application structure, and should not be implemented if the application does not need to have more features later as well as have a good test coverage.

3.2 Architectural Patterns

Architecture patterns, which are a subset of design patterns, define how to define the structural organization of a software application. [16]. Architectural patterns are very important in designing software application. Without a clearly defined architecture from the very beginning, the source code will turn out to be a collection of unorganized codes which are tightly coupling and hard to change and maintain. In Android application development, the model view controller is the most popular architectural pattern. [1]

The Model View Controller (MVC) which is first introduced in the Smalltalk-80 programming environment played an important role in the modern graphical user interface architecture pattern. Composed of three components: model, view, controller, MVC aims to help developing a better organized and more maintainable code. The main idea behinds the MVC architectural pattern is separated presentation which aims to ensure the domain and business logic are separated completely from the presentation. [2]

Figure 3 illustrates the relationship between the model, view and controller in MVC architectural pattern. The model is responsible for providing business logic for the application and notifying controllers about data change. The view is responsible for presenting information to the user. It provides requests to the controller and waits the controller request to update. The controller acts as the glue between the view and model. It accepts user inputs from the view or data update from the model as events, translates though events to requests which are later sent to model/view to update their content.

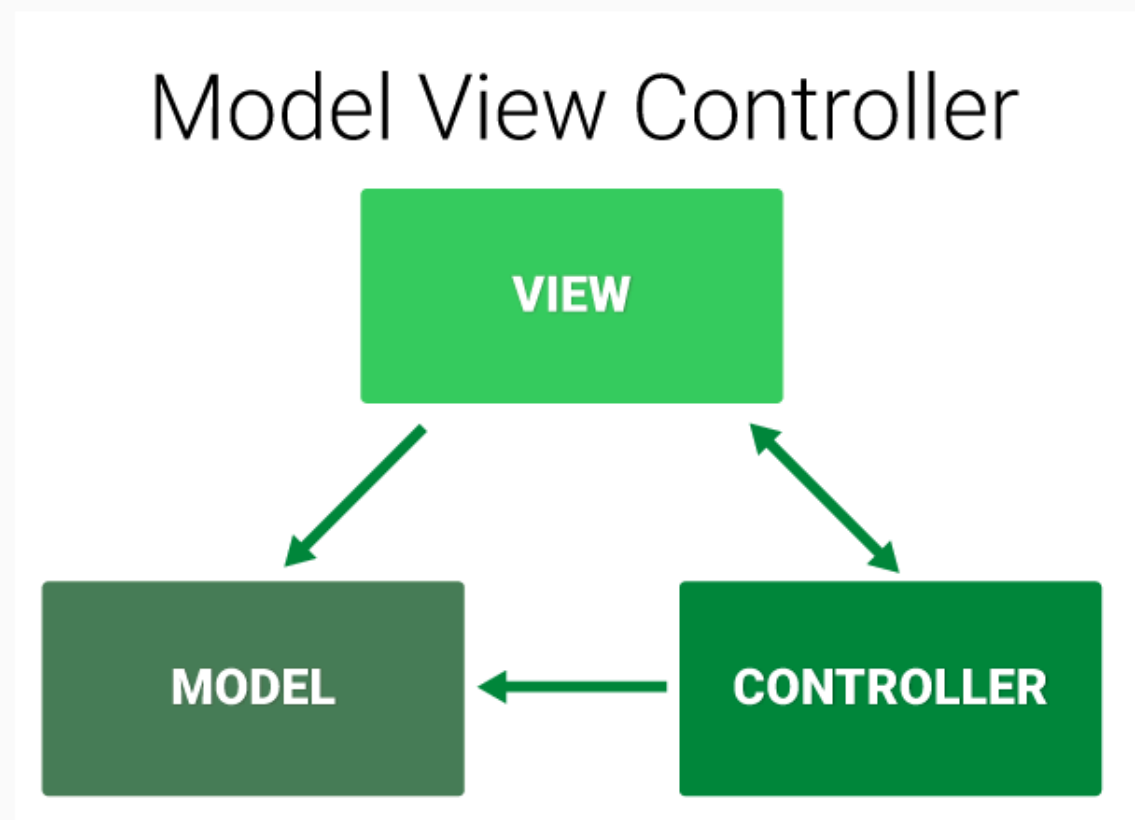

Figure 3. An Overview of Model View Controller. [3]

The main benefit of a separated presentation is to provide a decoupled and reusable code base which is easy to maintain and easy to test. The business logic can be tested without caring about any dependency with the view. Adding new features to the application or refactoring are easier for developers and cheaper for organizations. For example, if the application goes through re-design while remains business logic, developers just need to write a new controller and a new view and keep using the same models which are carefully developed and tested due to their loose coupling. Separated

presentation will decrease the amount of work for the developer, as well as reduce the cost for the organization. [4]

4 Clean Architecture and Reactive Programming

4.1 Clean Architecture

There are numerous architectural patterns introduced over the last ten years like hexagonal architecture [19], onion architecture [21] or screaming architecture [20]. Though their detailed implementations are different, all share the same object: separation of concerns. They should all produce a system which allows business logic to be testable as well as independent of the framework, UI or any third-party agencies and services.

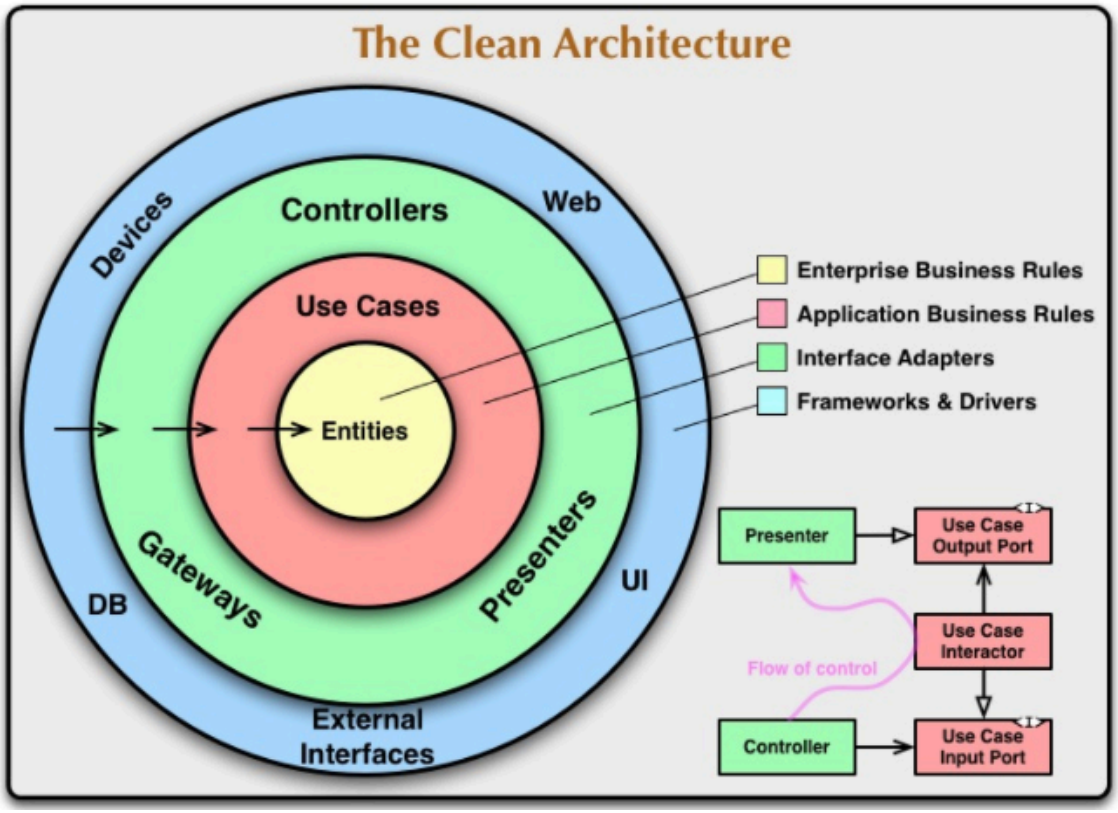

Figure 4. The Clean Architecture [7]

Figure 4 describes different layers and data flow in clean architecture. The entities layer provides application business rules. They are normally simple objects or a set of data structure and function. No operation change in the application should affect the entity layer. The use cases layer provides application-specific rules. Changes in this layer should not affect the entities. Also, this layer should not be affected by some ex-

ternal changes like UI changes or framework changes. The interface adapter layer provides a way to transform data from the format of entities and use cases to an external agency. Code in this layer should not know anything about what frameworks or drivers are being used by the application. The framework and driver layer is composed of frameworks and tools like database. Developers just normally glue the code to the inward layers.

To make this architecture work, developers should respect the dependency rule – dependencies should only point inward. The code in the inner layer should be unaware of the code in outer layer, as well as should not be affected by any changes in the outer layer. Developers should use the dependency inversion principle [22] when they need to cross the boundaries.

Though following clean architecture makes an application intrinsically testable, it also introduces varieties complexities to code base. It is over-engineering for a small application.

4.2 Model View Presenter

The Model View Controller pattern is perfect for Android development, until the application scales beyond simple use cases like showing data. An Android application is much more complex than just showing some data to the user, and the application logic usually does not belong to the view or model. The controller becomes a “god object” which does too much and know too much very easily, which is hard to change, hard to scale, as well as hard to test since the logic is tight coupling. Slimming down these controllers is a challenge for Android developers who seek to improve the quality of the source code. Besides, in Model View Controller, the View sometimes has some connection with the data, making it hard to test the business logic. Loader, which is illustrated in figure 5, is introduced in Android 3.0, tries to make it easy to load data asynchronously in an activity or fragment.

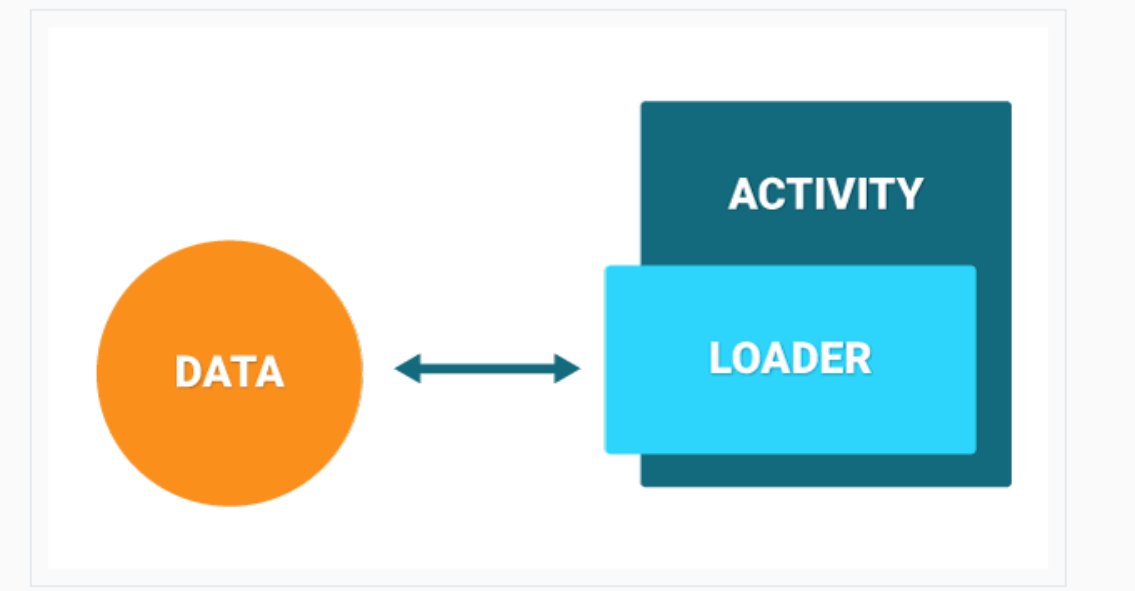

Figure 5. Android Loader [3]

The controller usually gives the view a model object and lets the view decide how to show it. It creates a dependency between the model and view, and violates the separation of the concern principle, making the application difficult to test its business logic. The view clearly does not need to know how the model structure, as well as the model does not need to know how its properties will be displayed to the end user. They should only communicate through the interface.

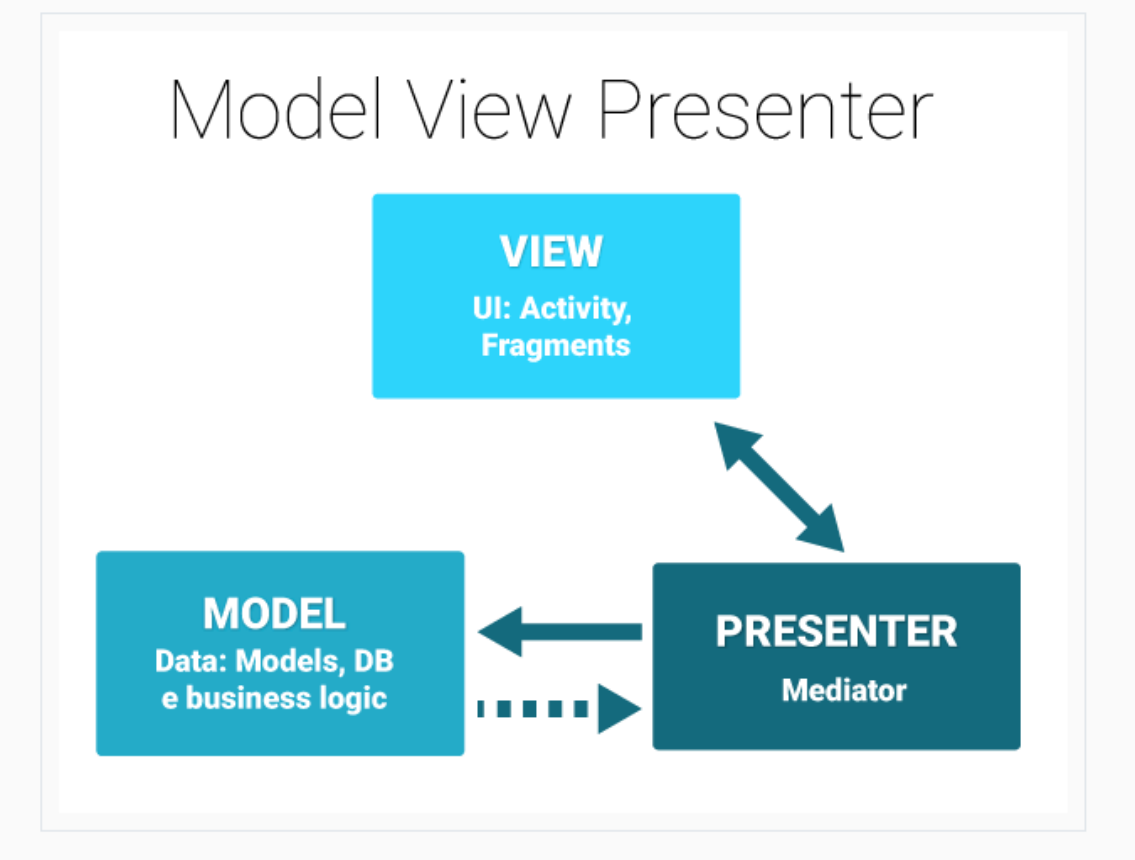

Figure 6. Overview of Model View Presenter [3]

The Model View Presenter (MVP) pattern is a pattern which shares the same idea with MVC, but with a more modern paradigm that leads to a better separation of concern. As illustrated by figure 6, the View is completely passive in this pattern. It cannot retrieve any data from the model, and just stays passive until the presenter tells it to do something. The Presenter, tightly coupling with a single view, acts as a mediator between the model and view. It retrieves data from the model, tells the view to render these data, and processes user action forwarded by the View. The view and the model finally become loose coupling which makes changing, adding and testing features easier.

4.3 Clean Architecture in MVP

Letting the model keep all the business logic makes an application easier to test. However, it introduces a massive model instead of a massive controller. Clean architecture

can help to slim down the model by introducing a domain layer between the presenter and data model, referred as the domain layer.

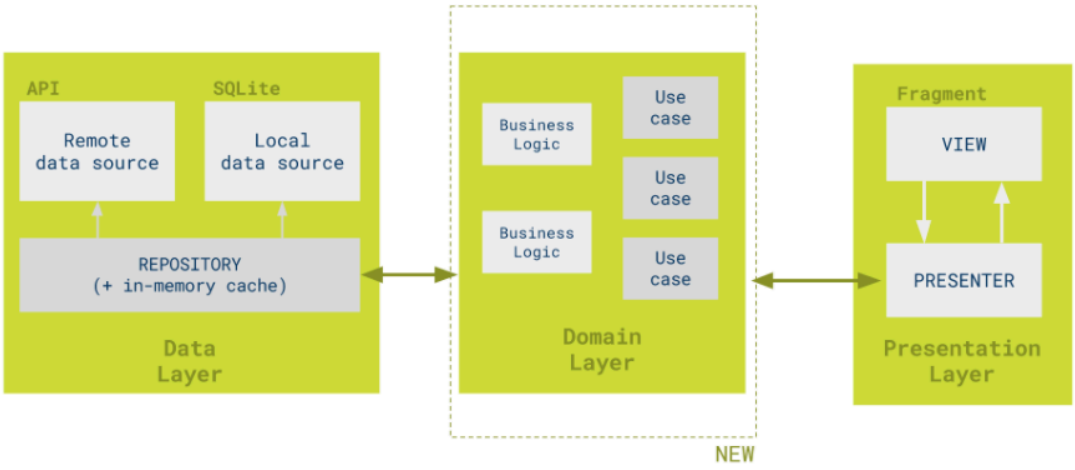

Figure 7. Applying clean architecture in MVP [8]

As shown by figure 7, the domain layer holds all the application logic, referred to as use cases. It communicates with the presenter through use cases, while retrieves and processes data from the model. The model layer now holds only business logic. This business logic is now entirely independent of interaction with the user, making the application more flexible and maintainable.

By separating the model layer into the domain and data layer, we separate the business logic from how data is stored. It becomes easier to mock and stub data to test business logic. However, developers should only consider applying clean architecture by introducing more layers when the application needs to scale to have more features as well as have more developers, because the main drawback of clean architecture is over-engineering for simple applications.

4.4 Challenges in Developing a Responsive Application

Android applications need to be highly responsive, not only because users want their action response immediately, but also the android system guard will display an Application Not Responding (ARN) dialog if the application is not responsive after a while. The Android Developer guide specifies a special time range: “100 to 200 ms is the threshold beyond which users will perceive slowness in an application” [8]. To provide good user experience, the operation in an Android application should be run asynchronously.

The Android application code will be run in the main thread, if not specified which thread it should run in. Basically, the main thread, also referred to as the UI thread, is responsible for rendering user interface and handling user interaction.

Android provides support for Thread and Runnable, and uses them as the basis of AsynTask, HandlerThread, IntentService and ThreadPoolExecutor. Android also provides Handler, Looper to make communicating between threads less demanding.

However, multithread programming is demanding. There are two challenging problems in multithread programming, which are correctness issues and liveness issues. Threads communicate by sharing access to their memories. While efficient, this form of communication introduces two kinds of errors: thread interference and memory consistency errors [24]. A different name for thread interference is race-condition. It happens when more threads share a value and try to modify it. Memory consistency errors happen when different threads have inconsistent views of data which should be the same.

A common solution for correctness problems is providing a lock for this shared value. However, locking the resources introduces liveness issues (an application's ability to execute in timely manner is known as liveness) [24]. The most commonly known liveness issue is deadlock. When two or more threads have circular dependency, these threads will block forever, waiting for each other. Less common than deadlock is starvation and livelock. Starvation happens when a thread is unable to access shared resources due to other "greedy" threads, hence not being able to make progress [24]. Meanwhile, a livelock is similar to a deadlock, except that these threads are not blocked but busy in response to the change of another thread, and none of them make any progressing.

4.5 Event Driven Architecture

Event-driven architecture, also known as message-driven architecture, is a software architecture in which the data flow is based on events which are contextualize operations on state. The main benefit of event-driven architecture is that it makes the application more maintainable and scalable, because there is no point-to-point integration.

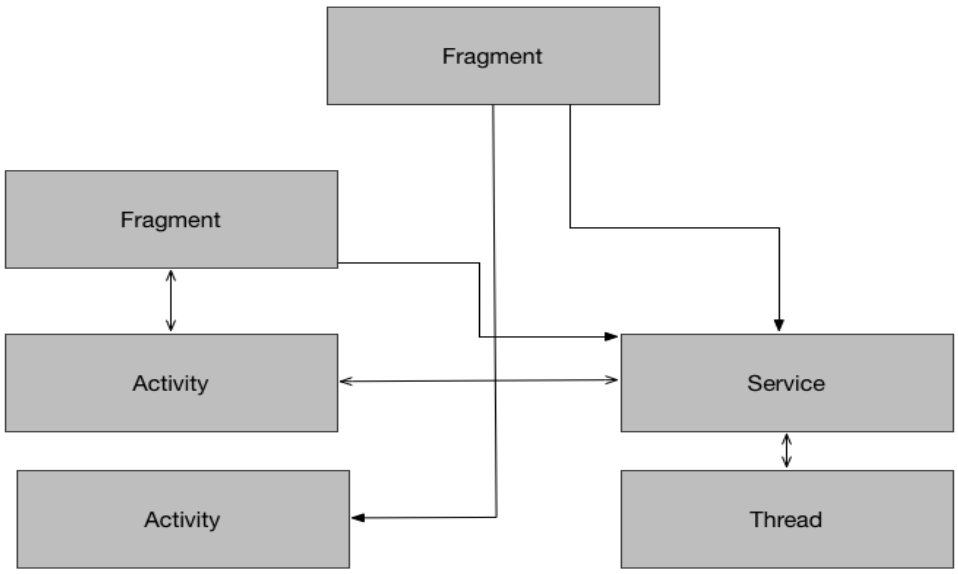

Figure 8. Tight coupling in Android application development

It turns out that event-driven architecture is very suitable for Android development. The Android itself is partly event-driven. However, as illustrated by figure 8, developing an Android application with event-driven architecture requires high coordination between multiple components in different threads. Besides, each event needs a specific handler, which leads to another common problem: callback nesting (a callback inside a callback inside another callback). Android developers use the reactive programming technique to reduce the tight coupling, the complexity of multithread programming as well as to avoid callback nesting.

4.6 Reactive Programming

Reactive programming is one of the most popular solutions to build applications which use event-driven architecture. Reactive programming is a broad concept, but in general

a program is reactive if it is event based, reacts to input and is viewed as a flow of data instead of a flow of control. [23]. It focuses on composition and transformation stream of data.

In reactive programming, the system reacts to an event and defines a behavior combining them. The system state changes over time based on the flow of events. A stream is a sequence of ongoing events ordered in time. Events are processed asynchronously by defining a function that will be executed when an event arises. A stream may be unbounded and time-dependent. It should also focus on transformation of data and must be traversed only once. ReactiveX is a general library which provides support in reactive programming by using the observable model.

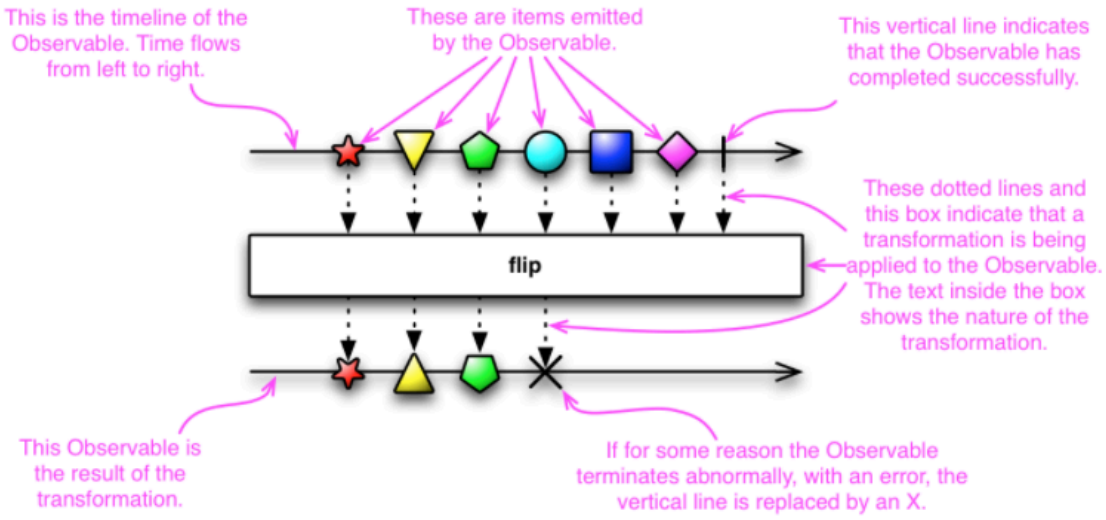

Figure 9. Overview of Observable [10]

The observable model which extends the observer pattern provides support for the developer to handle asynchronous data streams. As illustrated by figure 9, an observable model defines a mechanism to retrieve and transform data. The observers will capture and response to items emitted by the observable model whenever they are ready by subscribing to the observable

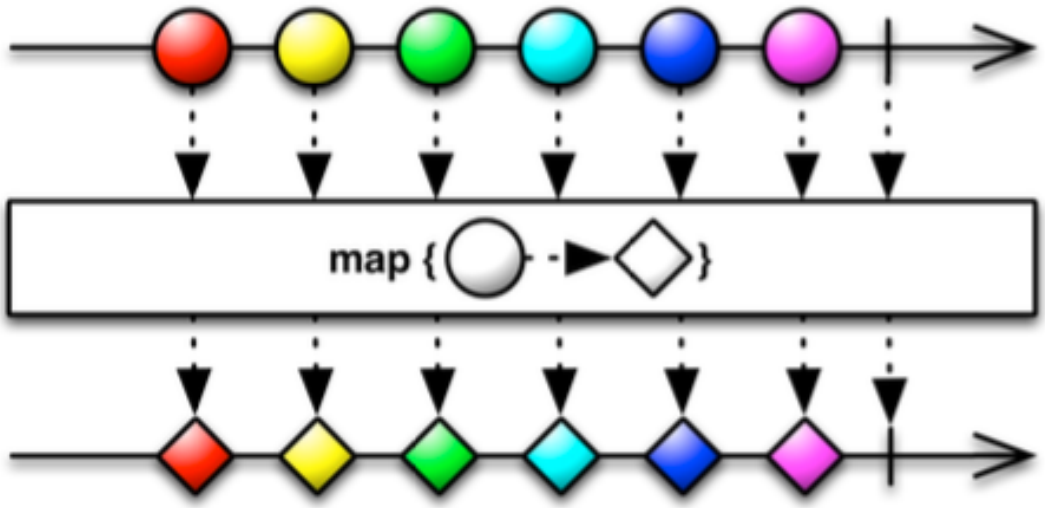

Figure 10. Map operator [11]

The event transformation is achieved by using an operator or a chain of operators. Mostly an operator works on single observable, and transforms it into another observable, as shown by figure 10.

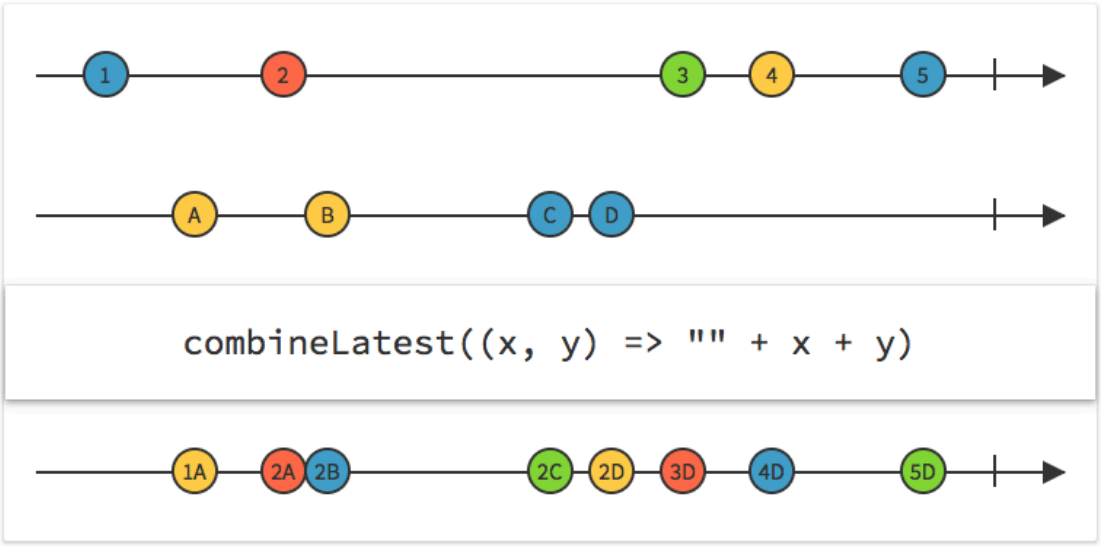

Figure 11. Combine latest operator [11]

There are some operators that operate on multiple observables, transforming them into one observable, like combine latest operator, merge operator and zip operator as illustrated by figure 11. [11]

By transforming, filtering, and combining operators, developers can compose and transform asynchronous sequences together in a declarative manner with all benefits of callbacks but without the drawbacks of nesting callback handlers (callback hell).

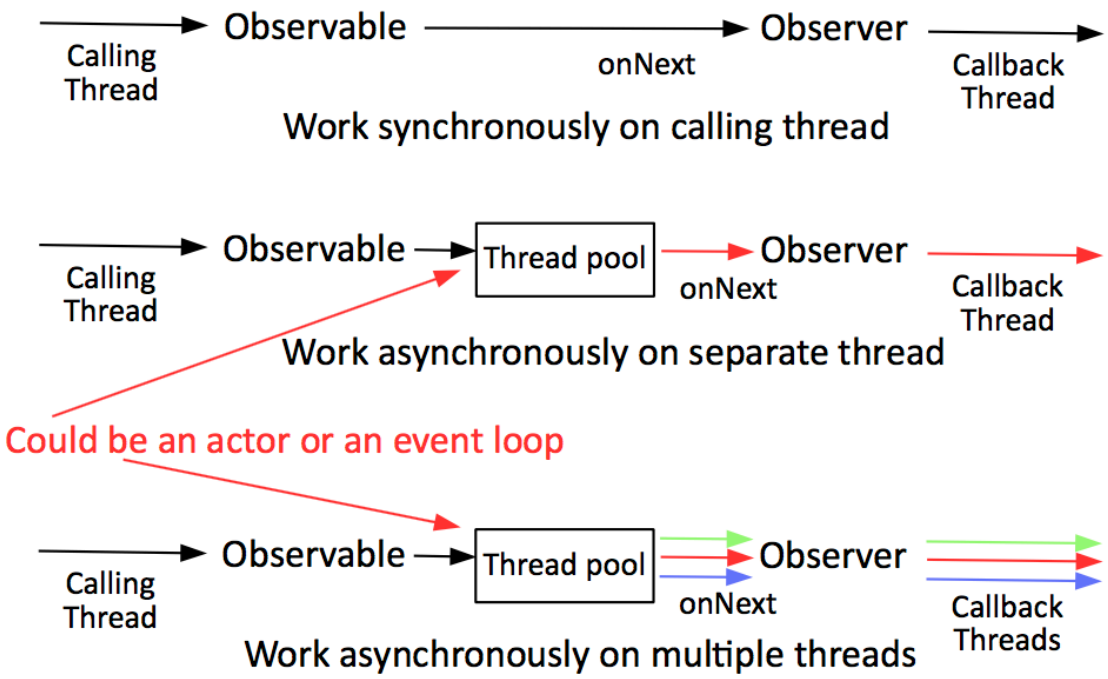

Figure 12. Overview of concurrency in ReactiveX [12]

As shown by figure 12, an observable is sequential, which means there are no concurrent emissions. By scheduling and combining observables, developers can have the advantage of concurrency while retaining sequential emission. Because all the operators should be side-effect free, developers do not need to deal with any concurrency issues.

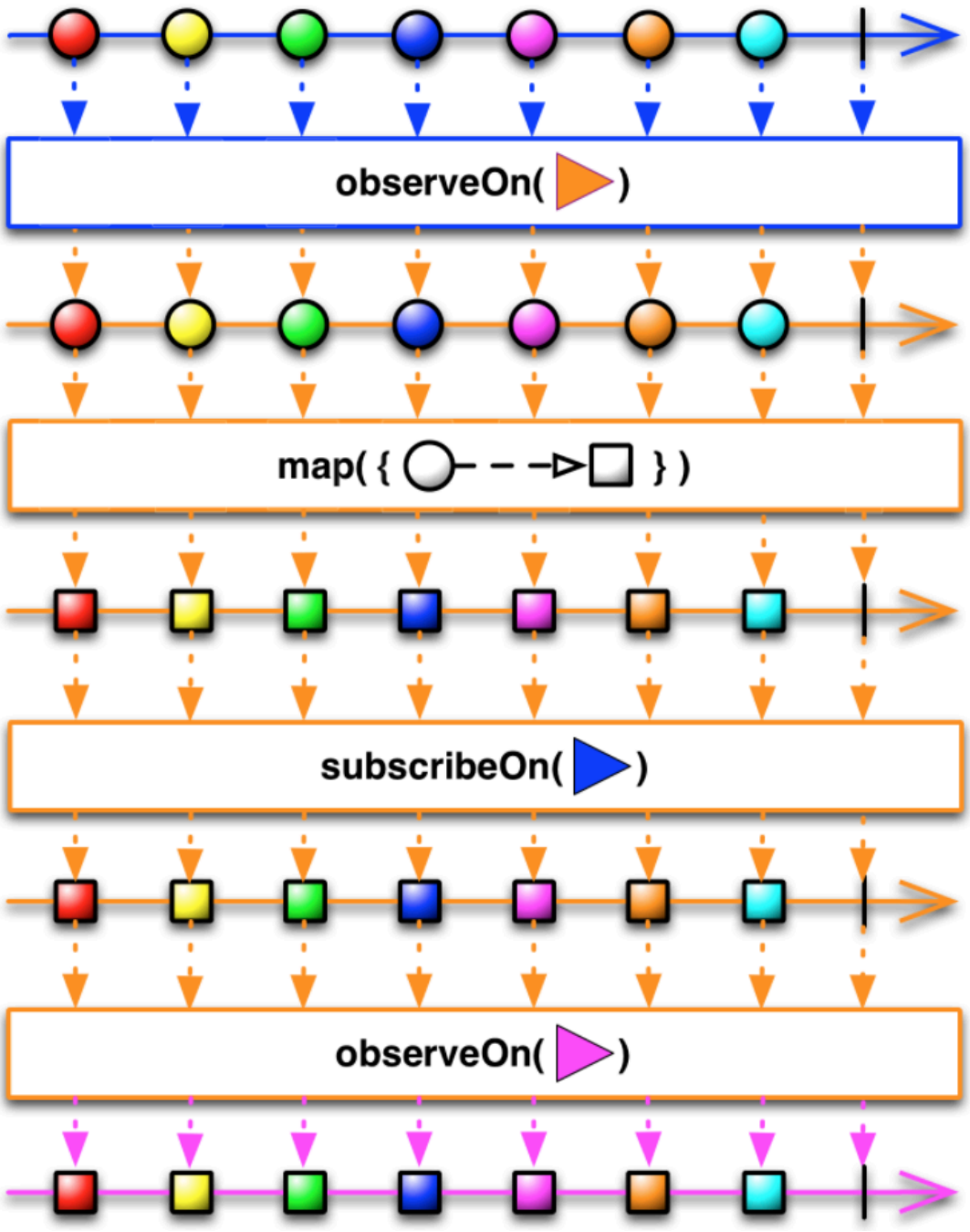

Figure 13. Multithread with operator observeOn and subscribeOn [11]

As illustrated by figure 13, ReactiveX provides two operators to work with multiple threads. Using observeOn and subscribeOn operators, developers can instruct which schedulers the operator should work on. If developers do not specify which schedulers operators operate on, the observable and its chain of operators will operate on the same thread where the subscribe-method called. The subscribeOn will specify the

scheduler on which the observable should operate, while observeOn will specify the scheduler on which the observable should send a notification to its observers.

5 Library use in developing an Android Application

5.1 Jack compiler and Java 8

Figure 14 provides an overview of Jack - a new Android toolchain which replaces the previous toolchain which consists of javac, Proguard, jarjar and dx.

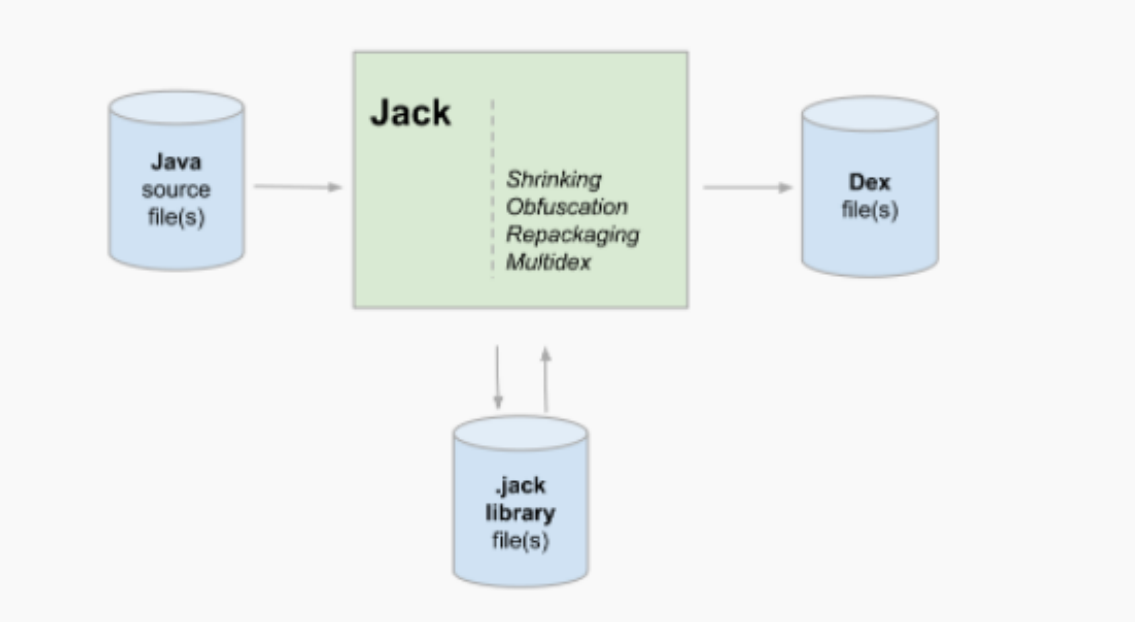

Figure 14. Jack overview [13]

Jack library format is .jack, which includes a pre-compiled dex code which fastens the compilation process (pre-dex) as illustrated by figure 15.

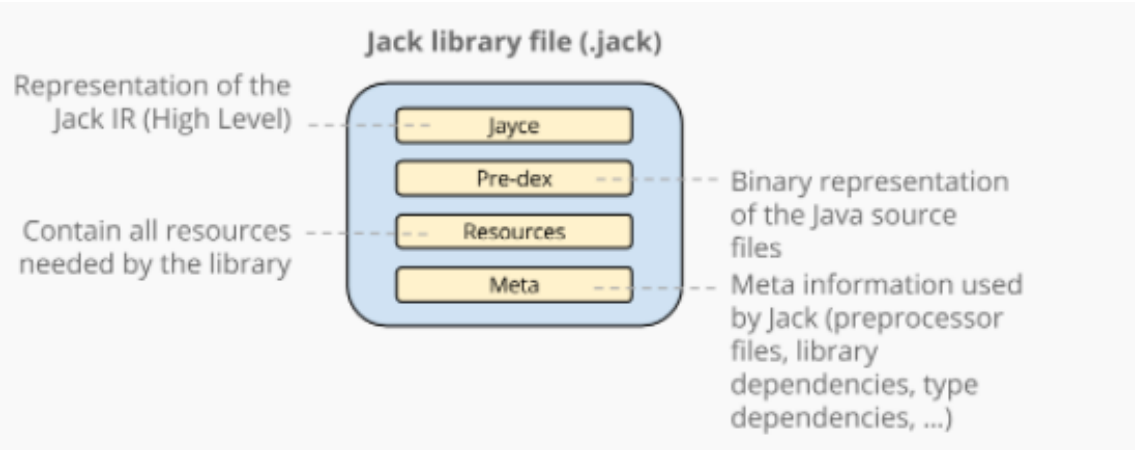

Figure 15. Jack library file [13]

Figure 16 describes how Jack generates a pre-dexed library. Jill tool is used to compile the existing jar library into a new library format which is used by Jack to generate a pre-dexed library.

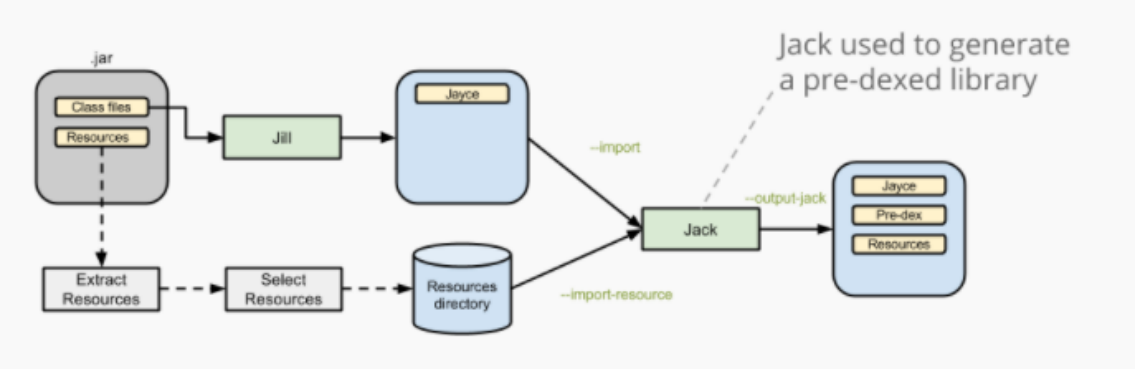

Figure 16. Workflow to import an existing .jar library [13]

```
android {
  ...
  defaultConfig {
    ...
    jackOptions {
      enabled true
    }
  }
  compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
  }
}
```

Listing 3. Example gradle build file to support Jack compiler and Java 8 [14]

Java 8 features are not supported by Android. However, with the Jack compiler developers can use some of Java 8 language features, with two most important features: lambda expression (available also on API level 23 and lower) and java.util.stream. Listing 3 describes how developers can enable Java 8 features in Android.

5.2 RxJava and RxAndroid

RxJava is ReactiveX implementation in the Java language. RxJava has two major stable versions: RxJava 1+ and RxJava 2+. The RxJava 1+ is widely used, however, in this application, C63Studio decides to use RxJava 2+ to experience the latest technology which will be helpful in later project decision making.

RxJava 2.0 [17] is built on top of Reactive Stream. Handling unlimited data streams is a demanding task in an asynchronous system. Reactive Stream provides support to manage the exchange of data streams between threads while guarantees the receiving side is not forced to buffer arbitrary amounts of data. The main difference between RxJava1 and RxJava2 is the introduction of Flowable. The observable is now non-backpressure (backpressure is a situation when an observable emits items more rapidly than operator or subscriber can consume them). The developer now should consider when to use the observable and flowable when architect data flows of the application.

RxAndroid [18] is a specific binding for RxJava. While RxJava provides five schedulers to work with (io, trampoline, computation, immediate and newThread), RxAndroid provides AndroidSchedulers.mainThread() which makes the communication between threads in Android much easier.

```
Subscription s = Subscriptions.empty();

Observable<Object>taskObservable = RetrofitClient.getTasksByTaskListId(taskListId);

subscription = taskObservable.subscribeOn(Schedulers.io())
        .observeOn(Schedulers.computation()
                    .map(tasks -> new UserTask(user, tasks))
        .observeOn(AndroidSchedulers.mainThread())
        .doOnSubscriber(() -> {
            if (view != null) {
                view.onLoadingStartedf();
            }
        })
        .subscribe(userTask ->
            if (view != null) {
                view.onLoadingSuccessfully(userTask);
            }
        },  throwable -> {
                if (view != null) {
                throwable.printStackTrace();
                view.onLoadingFailed(throwable);
                }
        });
```

Listing 4. Multithread in RxAndroid

In listing 4, the task is started by performing a network operation in a scheduler which is specified to IO-bound work. This scheduler is implemented by an Executor thread-pool (Executor is an object that executes submitted runnable tasks while thread pools are worker threads which are used to reduce the cost of thread creation. [24]) that can grow if needed. Then, the transformation task is handled in the computation scheduler. Finally, when the data is transformed, it will be shown in the UI thread. Communication between threads is very intuitive, and the only thing the developer should care about is releasing the subscription when the view is not presented to avoid memory leaks.

5.3 Dagger2

Dagger 2 is a dependency framework maintained by Google. It relies on Java annotation processors and the compile-time checks to analyze and verify dependencies. There are two generalized steps to get Dagger 2 running:

1. Create a Directed Acyclic Graph of dependencies (DAG).
2. Annotate injectable methods.

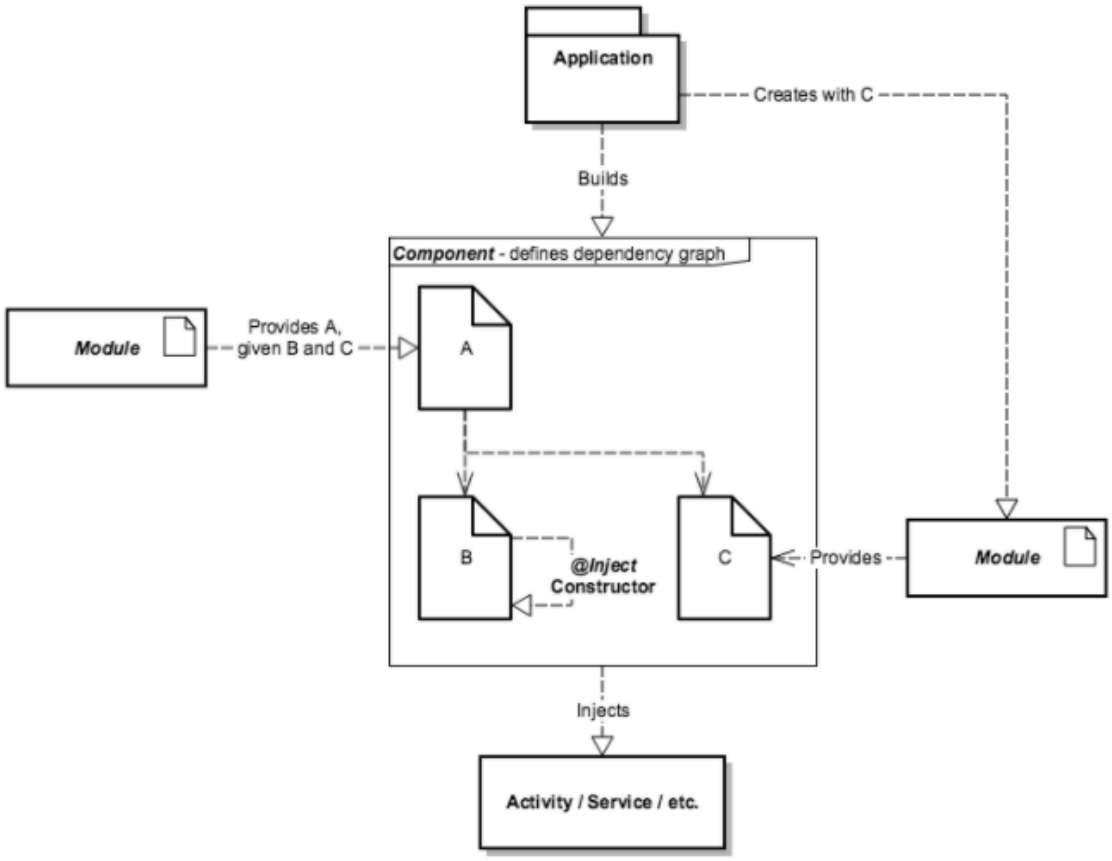

Figure 17. Overview of injection in Dagger 2 [15]

As illustrated by figure 17, there are three important concepts in Dagger2: Modules, Components and Injection targets. Modules, uses with @Module and @Provides annotations, defines a mechanism for providing dependencies. Injection targets use @Inject annotation to request dependencies. The component stays in between injection targets and modules, defines how the dependencies provided by modules are injected into objects which require dependencies.

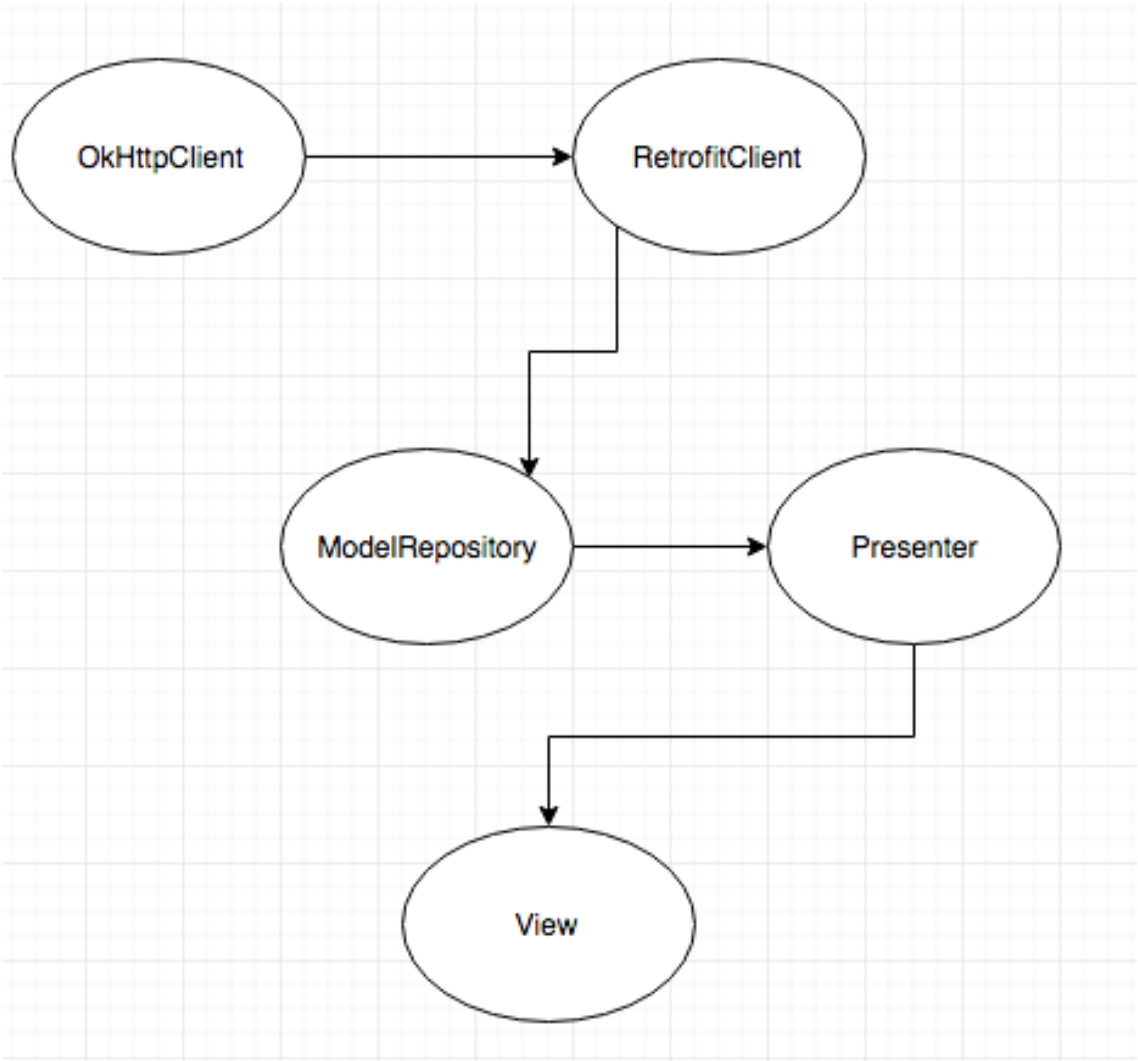

Figure 18. A dependencies graph in an Android MVP feature requires API call

Performing network operations is a common task in Android development. Basically, a model repository will be defined to perform network operations. The network call in Android is often handled by the Retrofit library and OkHttpLibrary. The presenter will need a model repository, and that presenter will be used by the view.

```
@Module
public class Module {
    @Provides
    @MyScope
    public OkHttpClient providesOkHttpClient(){
        return new OkHttpClient();
    }

    @Provides
    @MyScope
    public Retrofit providesRetrofit(OkHttpClient httpClient){
        return new Retrofit.Builder()
                .baseUrl(/*Base url*/)
                .client(httpClient)
                .build();
    }

    @Provides
    @MyScope
    public ModelRepository providesModelRepository(Retrofit retrofit){
        return retrofit.create(ModelRepository.class);
    }

    @Provides
    @MyScope
    public Presenter providesPresenter(ModelRepository repository){
        return new Presenter(repository);
    }
}
```

Listing 5. A module in Dagger2

Figure 18 and listing 5 show how developers use Dagger 2 to build dependencies graph for performing network operations. The dependencies are defined in a module. The OkHttpClient, Retrofit, ModelRepository and Presenter instance will be created. A Retrofit instance depends on the OkHttpClient instance which is recognized by Dagger2 with annotation @Provides. The retrofit instance will then be used to create the repository instance, and the repository instance will be used to create the presenter instance.

```
@MyScope
@Component(modules = Module.class)
public interface Component {
    void inject(View view);
}
```

Listing 6. A component in Dagger 2

Listing 6 illustrates the component which is the injector in Dagger 2. It defines the target which will be injected. Dagger2 will then use this interface to generate code and adding prefix Dagger which is responsible for instantiating and injecting dependencies.

```
public class View extends Activity{
    @Inject Presenter presenter;
    Component mComponent;

    @Override
    public void onCreate(){
        super.onCreate();
        mComponent = DaggerComponent.build();
        mComponent.inject(this);
}
```

Listing 7. An injection tagets in Dagger2

The View will then use the presenter which is instantiated and injected by DaggerComponent as shown by listing 7.

5.4 Retrofit2 and OkHttp3

OkHttp3 is an HTTP client that provides support for Http/2 or Connection pooling if Http/2 is not available. It also provides support for caching as well as transparent GZIP to reduce download sizes.

```
OkHttpClient httpClient = new OkHttpClient.Builder()
  .certificatePinner(
    new CertificatePinner.Builder()
        .add("*.example.fi","sha1/asdasdJASDASdas/=")
        .build())
  .build();
```

Listing 8. Add Certificate Pinner to OkHttp

```
OkHttpClient httpClient = new OkHttpClient.Builder()
    .addInterceptor(chain ->
        Request req= chain.request();
        CacheControl cc = new CacheControl.Builder()
            .maxStale(14, TimeUnits.DAYS)
            .build();
        req = req.newBuilder().cacheControl(cc).build();
        return chain.proceed(req);
    })
    .cache(new Cache(new File (context.getCacheDir(), "cache")
                    , 10*1024*1024))
    .build();
```

Listing 9. Cache request.

As illustrated by the codes in listing 8 and listing 9, it is very easy to add support for https and caching in OkHttp3. It is also very easy to add an interceptor to modify the request header as well as the response header.

```
package c63.studio.fi.modus.core.authentication;

import android.support.annotation.NonNull;

import c63.studio.fi.modus.core.authentication.Model.AccessToken;
import io.reactivex.Observable;
import retrofit2.http.POST;

interface AuthenticationService {

    @NonNull
    @POST("accounts/get-token")
    Observable<AccessToken> signIn(String email, String password);

    @NonNull
    @POST("accounts/register")
    Observable<AccessToken> signUp(String email, String password, String name);
}
```

Listing 10. An example of retrofit client

Retrofit is a library which transforms Rest API into a Java interface. By extending the Façade pattern, retrofit reduces the complexity for working with rest API. Listing 10 shows how developers can use retrofit to construct a network operation.

```
@NonNull
@Provides
@Singleton
public AuthenticationController providesAccountController(
        @NonNull final CredentialsController credentialsController) {
    AuthenticationService authenticationService = new Retrofit.Builder()
            .baseUrl(BuildConfig.SERVER_ENDPOINT)
            .addCallAdapterFactory(RxJava2CallAdapterFactory
                    .createWithScheduler(Schedulers.io()))
            .addConverterFactory(GsonConverterFactory
                    .create(GsonFactory.create()))
            .client(new OkHttpClient())
            .build()
            .create(AuthenticationService.class);
    return new AuthenticationController(authenticationService, credentialsController);
}
```

Listing 11. Create a retrofit

Retrofit2 works very well with RxJava, which makes asynchronous networking in Android become easier. Listing 11 shows how developers add a custom call adapter to

make Retrofit work with RxJava. It also shows how to add an instance of OkHttp3 to Retrofit client to utilize benefits of OkHttp3

5.5 Testing in Android Development

Testing is one of the most important parts in software development and helps to guarantee the quality of the software. There are numerous frameworks and libraries providing support for unit testing and instrumental testing in Android development.

Unit tests are the fundamental tests in one's app testing strategy. By creating and running unit tests against the code, the developers can easily verify that the logic of individual units is correct. Running unit tests after every build helps the developers to quickly catch and fix software regressions introduced by code changes to the app.

For testing Android apps, the developers typically create these types of automated unit tests:

- **Local tests:** Unit tests that run on one's local machine only. These tests are compiled to run locally on the Java Virtual Machine (JVM) to minimize execution time
- **Instrumented tests:** Unit tests that run on an Android device or emulator. These tests have access to instrumentation information, such as the Context for the app under test.

```
@RunWith(RoboelectricGradleTestRunner.class)
@Config(constants = BuildConfig.class)
public class ModusAuthenticationFragmentTest(){
    private HomeActivity homeActivity;

    @Before
    public void setUp(){
        homeActivity = Roboelectric.setupActivity(HomeActivity.class);
    }

    @Test
    public void testLoginButton(){
        Button loginBtn = (Button) homeActivity.findViewById(R.id.loginBtn);
        Intent expectedLoginIntent = new Intent(homeActivity, LoginActivity.class);
        Intent actualLoginIntent = Shadows.shadowOf(homeActivity).getNextStartedActivity();
        assertTrue(actualLoginIntent.filterEquals(expectedLoginIntent));
    }
}
```

Listing 12. Testing with Roboelectric

Listing 12 illustrated testing with Roboelectric. Roboelectric is a unit testing framework which reduces the building, deploying and launching time by running an Android test inside the JVM instead of Android devices or simulators. Roboelectric uses shadows classes which extend or modify the behavior of clases in Android SDK to avoid the need of devices or simulators. By using roboelectric, we can run instrumental tests like normal local tests.

```
@Before
public void setUp(){
    service = Mockito.mock(AuthenticationService.class);
    presenter = new AuthenticationPresenter(service);
}

@Test
public void testLoginSuccessfully() {
    when(service.signIn("test@email.fi", "testpassword"))
      .thenReturn(Observable.just(DummyToken.dummyToken()));

    AuthenticationView view = Mockito.mock(AuthenticationView.class);

    presenter.setEmail("test@email.fi");
    presenter.setPassword("testpassword");

    presenter.login();

    verify(view).startLogin();
    verify(view).loginSuccessfully(DummyToken.dummyToken());
}
```

Listing 13. Mocking with Mockito

Mocking frameworks provide support to make dependencies easier in unit tests by mocking the behavior of real objects. Mockito is one of the most popular mocking frameworks used in Java development, especially Android development. As illustrated in listing 13, Mockito mocks the behavior of AuthenticationService and testing behavior of the view. Without the help of mockito, it is difficult to create the service and view object.

While unit testing provides verification for the logic of individual units, User Interface (UI) testing assures that the application meets its functional requirements and achieves a high standard of quality such that it is more likely to be successfully adopted by users.

Typically, there are two types of UI testing:

- *UI tests that span a single app:* This type of test verifies that the target app behaves as expected when a user performs a specific action or enters a specific input in its activities. It allows the developers to check that the target app returns the correct UI output in response to user interactions in the app's activities. UI testing frameworks like Espresso allow the developers to programmatically simulate user actions and test complex intra-app user interactions. [25]

- *UI tests that span multiple apps:* This type of test verifies the correct behavior of interactions between different user apps or between user apps and system apps. For example, the developers might want to test that the camera app shares images correctly with a 3rd-party social media app, or with the default Android Photos app. UI testing frameworks that support cross-app interactions, such as UI Automator, allow the developers to create tests for such scenarios. [25]

6 Applying Clean Architecture and Reactive Programming in Sunshine

As stated in chapter 2, the Sunshine application has some drawbacks which makes the application difficult to test and scale. The application needs to be refactored using clean architecture before adding new features.

6.1 Overview of new architecture

Sunshine follows clean architecture by separating direct access to the content provider from the forecast fragment as illustrated by figure 19. The application logic will be separated into different use cases which follow the single responsibility principle. The business logic can access data through repositories which abstract how to save and load data.

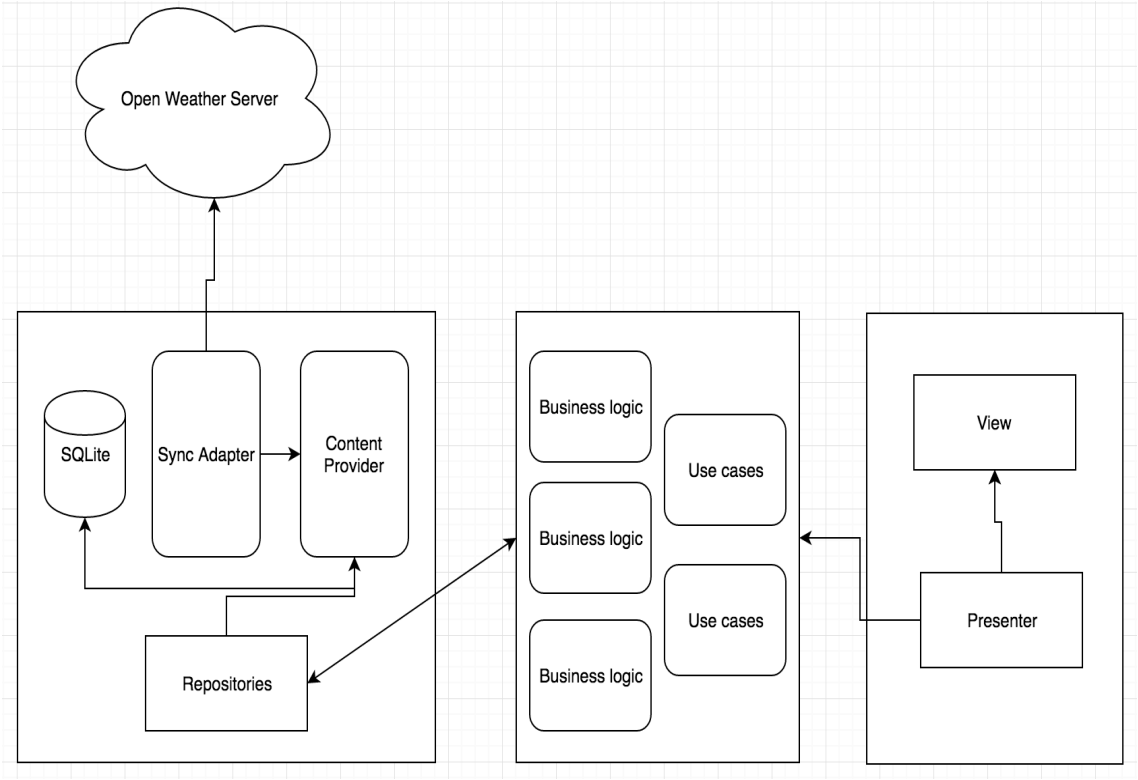

Figure 19. Clean architecture in Sunshine

Before refactoring Sunshine application using clean architecture, C63 Studio development team needs to decide which application folder structure should be used for Sunshine.

6.2 Application folder structure

As shown by figure 20, the old Sunshine application does not follow any folder structure. It makes application difficult to navigate as well as naming class when the application grows.

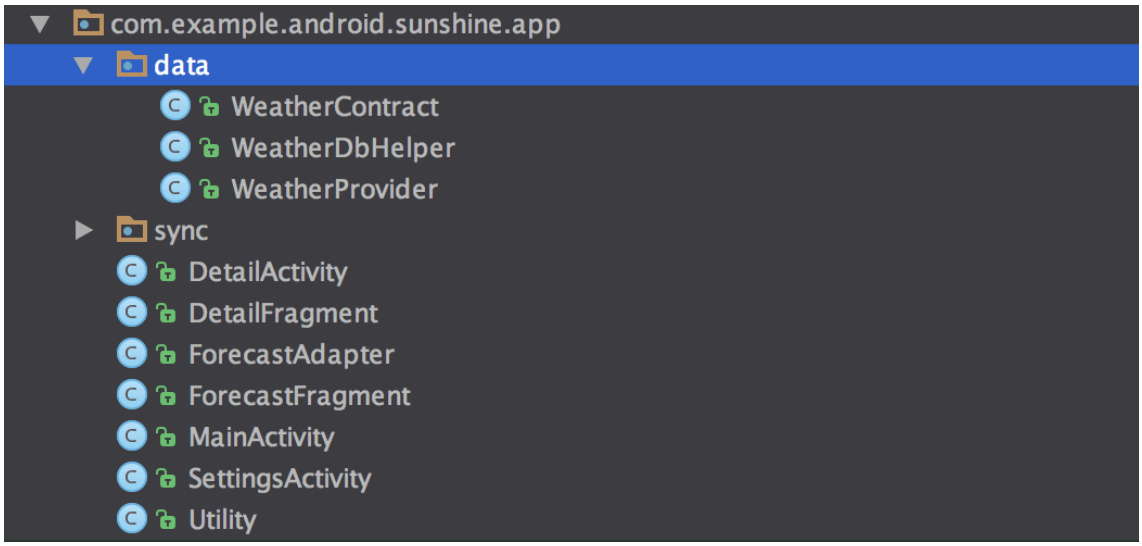

Figure 20. Old sunshine application folder structure

Normally developers have two ways of structuring an application: horizontal structure and vertical structure.

Based on the functional role, the horizontal structure is very suitable for small projects without the need to scale later. However, if the application becomes larger with more functionality, the structure will soon be very difficult to navigate.

Instead of based on the functional role, the vertical structure utilizes features which makes it easier to scale the application. It reduces the efforts to edit and add new features easily, and makes it easy for new developers who join the project to navigate the source code and has an overview of what functionality the application provides.

6.3 Clean Architecture on Sunshine home page

The current implementation of ForecastFragment (illustrated by figure 21) is very difficult to change and maintain. The developers need to know detailed implementation of the database if they want to show data to the UI. The developers also need to implement new forecast fragment and forecast adapter for every change in database schema.

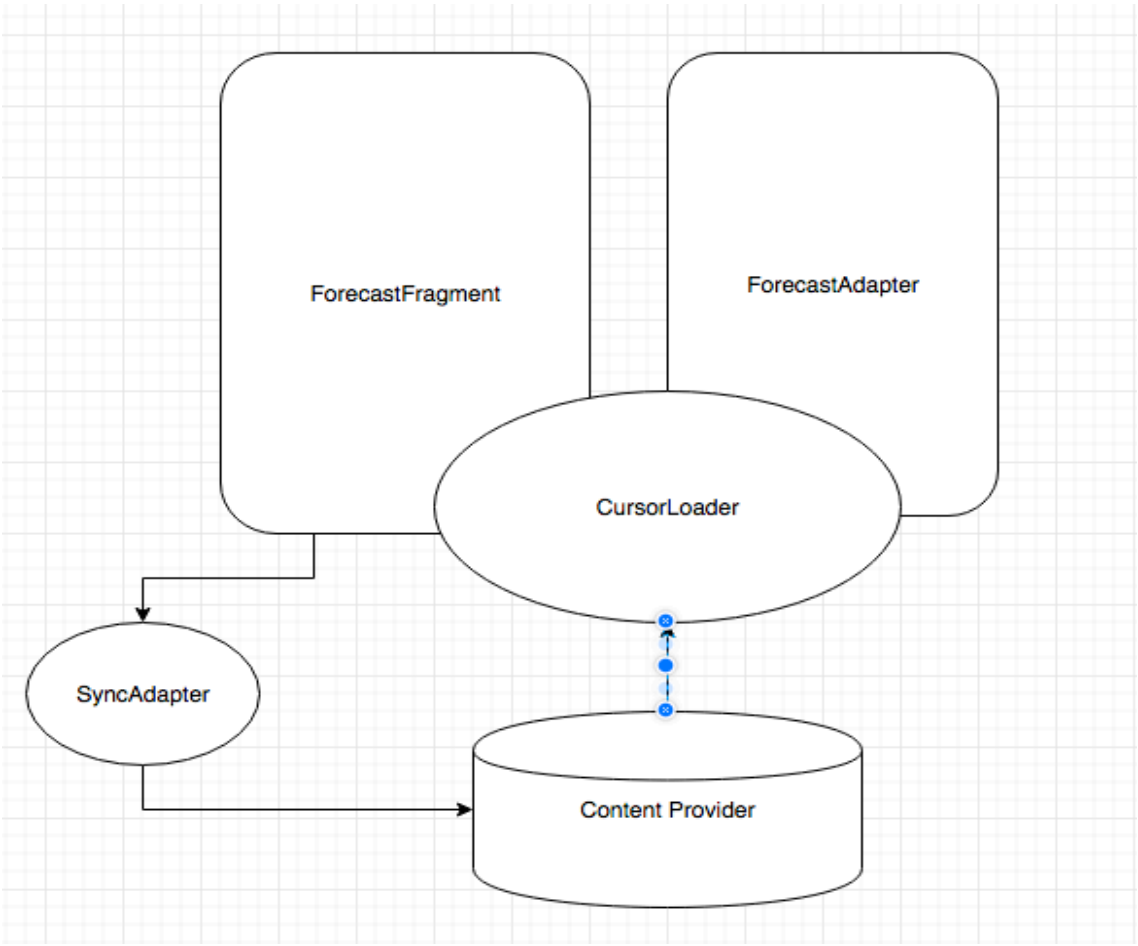

Figure 21. Tight coupling implementation of Sunshine

This implementation is not only inflexible but also difficult to test. It is nearly impossible to provide any interaction test between the sync adapter, content provider and forecast fragment. By introducing the domain layer and data layer, developers can avoid tight coupling in this scenario.

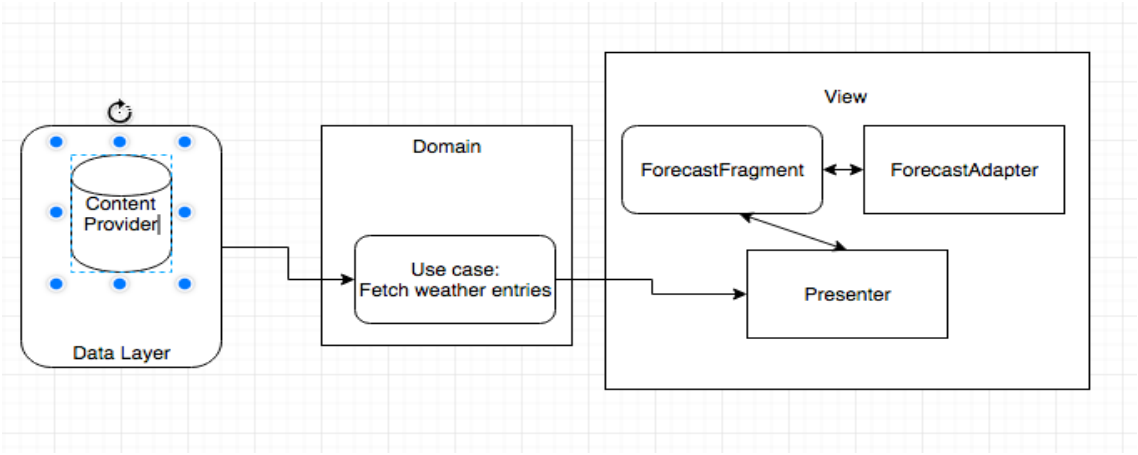

Figure 22. Clean architecture in Sunshine home screen

As illustrated in figure 22, it is now very easy to change how data is stored and accessed in the application. Developers just need to provide different implementations which produce the same data required by the use case when they want to make any changes in the data layer. The forecast fragment and forecast adapter can stay the same. Using the domain layer and data layer is not only flexible but also testable. It is now very easy to mock data for interaction testing between the domain layer and view layer, without worrying about detailed implementation of the data layer.

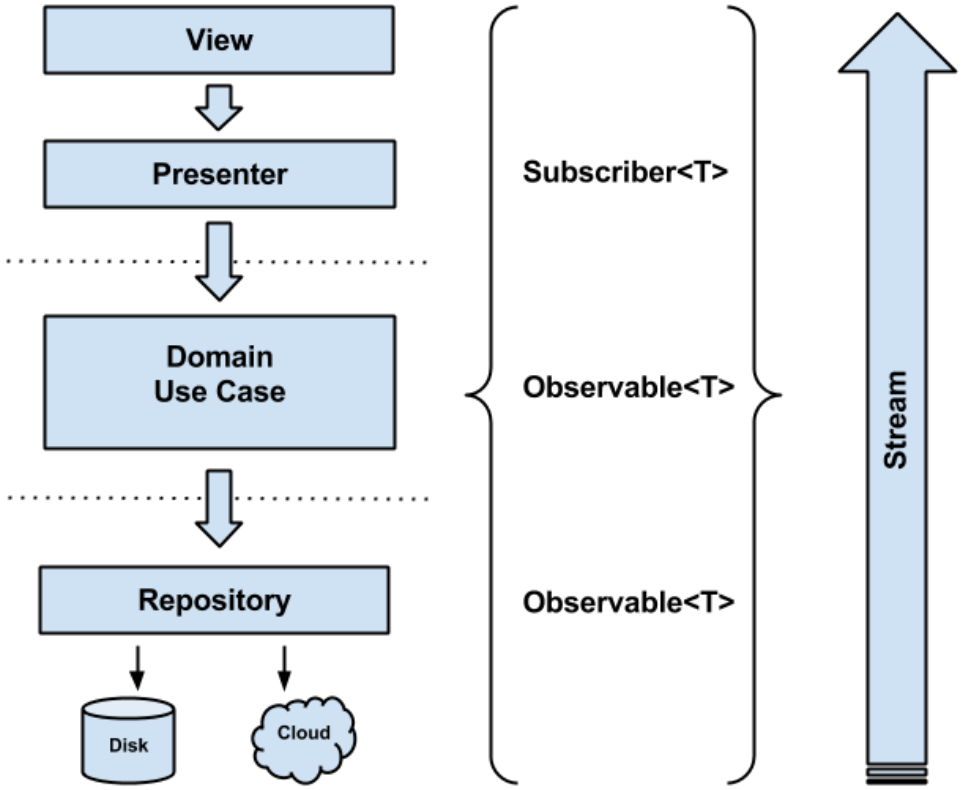

Figure 23. Reactive Clean architecture for Sunshine

Figure 23 describes how applying reactive programming technique in clean architecture can provide a way of crossing boundaries without the dependency inversion needed. It has also a higher level of abstraction for communicating between layers. It is also easier to mock the observable and subscriber to test interaction between layers.

6.4 Dependency problems in Sunshine

Listing 14 shows an example of code which is hard to test. There is no out of the box solution for writing unit tests for these two methods shown in listing 14. The SharedPreferences should be passed as a parameter to the function.

```
public class Utility {
    public static String getPreferredLocation(Context context) {
        SharedPreferences prefs = PreferenceManager.getDefaultSharedPreferences(context);
        return prefs.getString("location",
                "94043");
    }

    public static boolean isMetric(Context context) {
        SharedPreferences prefs = PreferenceManager.getDefaultSharedPreferences(context);
        return prefs.getString("units",
                "metric")
                .equals("metric");
    }
```

Listing 14. Example of hard-to-test code

However, passing SharedPreferences to the function leads to another problem. The developers need to specify exactly what SharedPreferences they will use to call the function. It is a difficult task for new developer to know what SharedPreferences should they use if the code base is large and complex. This problem can be solved by using Dagger 2 to inject the shared preferences.

6.5 Implementation of new home screen

The very first task of refactoring the Sunshine application is to build the core of dependency system using Dagger 2.

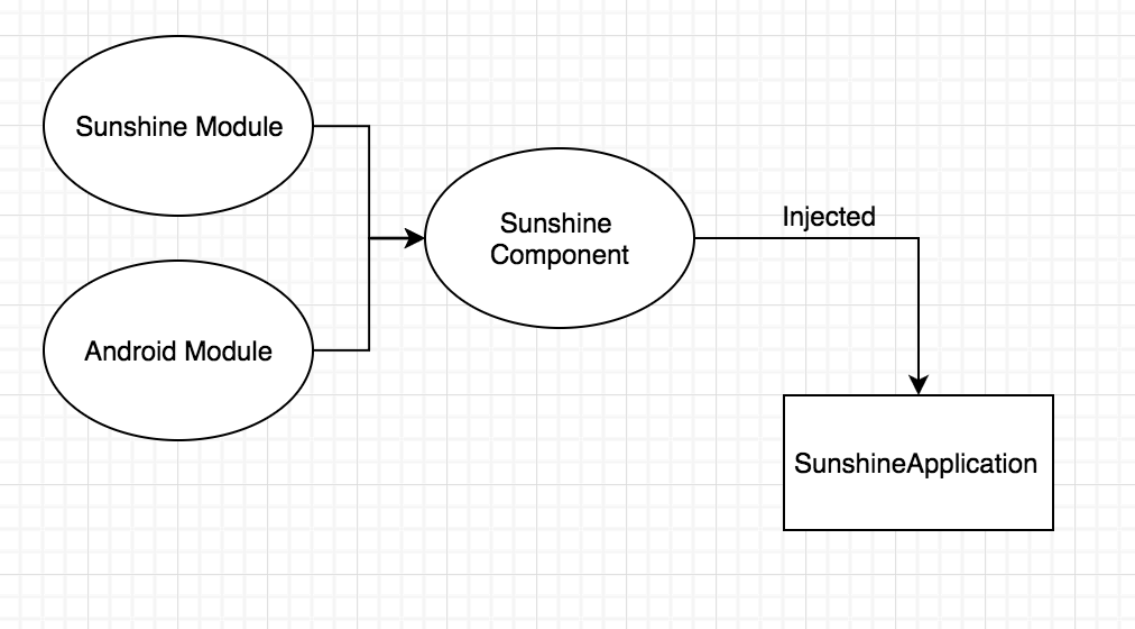

Figure 24. Overview of core module and component in Sunshine Application

Figure 24 illustrates overview of dependency graph in Sunshine application. The Sunshine Module provides a singleton http client and managers which provide access to the data layer while the Android module provides Android-specific objects like SharedPreferences, Context and Application. All these are provided to the Sunshin application through the Sunshine Component. Later, new feature will define other components or sub-components which depend on Sunshine component.

The Data layer includes three managers: LocationManager, WeatherManager and SettingManager. The location manager is responsible for adding and getting a location, the weather manager is responsible for adding and retrieving weather information while the setting manager is responsible for storing application setting.

```
refreshSubject
        .flatMap(aLong -> settingManager.locationObservable())
        .flatMap(s -> {
            String units = settingManager.isCurrentUnitIsMetric() ? "metric" : "imperial";
            return weatherService.getDailyWeatherList(s, "json", units, 14, BuildConfig.OPEN_WEATHER_MAP_API_KEY);
        })
        .subscribe(openWeatherAPIResponse ->
                getWeatherDataFromJson(openWeatherAPIResponse, settingManager.getCurrentLocation()));
updateWeather();
```

Listing 15. Reactive programming in Sunshine

Reactive programming is powerful in this scenario. As illustrated by listing 15, the setting manager will emit a new item to the stream for every change in the application setting. In the weather manager, there is an observer that observes this stream and will try to get new weather information. The Sunshine application is up-to-date without explicit calls whenever setting changes.

There are two use cases for the WeatherList screen: get Weather and set display unit. The get weather use case should be executed when the location setting changes, and the set display use case should be executed when the unit setting changes (metric to imperial or vice versa).

```
private void buildDisposable() {
    disposable = getWeatherUseCase.getWeatherObservable()
            .mergeWith(updateUnitDisplayUseCase.getWeatherObservable())
            .doOnSubscribe(disposable1 -> {
                if (view != null) view.onWeatherStartLoading();
            })
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(entries -> {
                if (view != null) view.onWeatherLoadSuccessfully(entries);
            }, throwable -> {
                if (view != null) view.onWeatherLoadFailed(throwable);
            });
}
```

Listing 16. Usage of two use cases in ForecastListPresenter

As illustrated by listing 16, now the presenter does not hold any business logic. It only observes changes from stream which is merged from two stream emitted by the get weather use case and update unit display use case. The whole view layer (Activities, Fragment and Presenter) does not know anything about how and where the data is fetched and stored. Its only task now is to display what is given by these use cases.

Testing is a major selling point of clean architecture and dependency injection. By using the interface to abstract communication between the domain layer and view layer, the developers can easily mock and test interaction between the presenter and view. It is also very easy to provide unit tests for each use case when there are hardly any dependencies between layers.

7 Conclusion

After refactoring the Sunshine application, the team realized some advantages and disadvantages of applying clean architecture, dependency injection and reactive programming.

Clean architecture and dependency injections provide possibilities to change external dependencies or implementation details of the data layer or view layer without affecting the business logic. It also makes the application easy to test as well as easy to add new features if the new features' implementation is followed the dependency rules. The downside is it has a steep learning curve as well as it requires more time to set up the skeleton and data flow of the application and use cases for each model (This may also be considered as a good practice). Clean architecture and dependency injections also introduce more boilerplate code into the application to make things work. The implementation process is painful in the beginning, but it will reduce a lot of pain for later development.

Using RxJava provides a high level of abstraction to the application. The developers can focus on the business logic of interdependent events instead of focusing on detailed implementation how these events are communicated and coordinated. It also makes the Sunshine application becomes more reactive: every change in the data layer will immediately trigger a change in the view layer and vice versa.

Refactoring the sunshine application took a lot of time to refactor even though it is just a simple application. But the result is positive. The team is confident about adding test and new features while providing good test coverage. The team also successfully applied this architecture to another Android application. The team decides to use this architecture for later Android projects.

Software architecture plays a major role in the success of every software project. It may take time in the beginning to design good architecture, but the effort will be paid off later. It is always easier to do the right thing at the beginning of the project than to try to refactor poor quality code with bad architecture later.

The team will keep researching good architecture design patterns, because software complexity is increasing every single day, and the team needs to keep up-to-date with latest trends in software development. However, for now the team has decided to use clean architecture for later projects as well as refactor all legacy projects by applying this architecture.

8 References

1 Mark Richards. Software Architecture Patterns: Understanding common architecture patterns and When to Use Them. O'Reilly; 2015

2 Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. Pattern-Oriented Software Architecture - A system of architecture, Volume 1. Willey; 2001.

3 Tin Megali. An Introduction to Model View Presenter on Android [online]. 30 March 2016.
URL: https://code.tutsplus.com/tutorials/an-introduction-to-model-view-presenter-on-android--cms-26162 Accessed 14 November 2016.

4 Martin Fowler. Separated Presentation [online]. 29 June 2016.
URL: https://martinfowler.com/eaaDev/SeparatedPresentation.html Accessed 24 October 2016.

5 Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley; 1994.

6 Martin Fowler: Inversion of Control Containers and the Dependency Injection pattern [online]. 23 January 2004.
URL: http://www.martinfowler.com/articles/injection.html Accessed 15 November 2016.

7 Robert Cecil Martin. The Clean Architecture [online]. 13 August 2012.
URL: https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html Accessed 16 November 2016.

8 Google Sample. Android Architecture Blueprints – MVP + Clean Architecture [online, Github repository]. 26 September 2016.
URL: https://github.com/googlesamples/android-architecture/tree/todo-mvp-clean/ Accessed 16 November 2016.

9 Android Developers. Keeping your app responsive [online].
URL: https://developer.android.com/training/articles/perf-anr.html#Reinforcing Accessed 17 November 2016.

10 ReactiveX. Observable [online].
URL: http://reactivex.io/documentation/observable.html Accessed 17 November 2016

11 ReactiveX. Operators [online].
URL: http://reactivex.io/documentation/operators/ Accessed 18 November 2016.

12 Mario Fusco. Reactive Programming for a demanding world: building event-driven and responsive application with RxJava [online]. 15 Mar 2015.
URL: http://www.slideshare.net/mariofusco/reactive-programming-for-a-demanding-world-building-eventdriven-and-responsive-applications-with-rxjava Accessed 18 November 2016

13 Compiling with Jack [online].
URL: https://source.android.com/source/jack.html Accessed 22 November 2016

14 Android Developer Guide. Use Java 8 Language Features [online].
URL: https://developer.android.com/guide/platform/j8-jack.html Accessed 22 November 2016

15 CodePath. Dependency Injection with Dagger 2 [online].
URL: https://github.com/codepath/android_guides/wiki/Dependency-Injection-with-Dagger-2 Accessed 22 November 2016

16 The Open Group. Architecture pattern [online].
URL: http://pubs.opengroup.org/architecture/togaf8-doc/arch/chap28.html

17 RxJava [online]:
URL: https://github.com/ReactiveX/RxJava Access 30 March 2017

18 RxAndroid [online]
URL: https://github.com/ReactiveX/RxAndroid Accessed 30 March 2017

19 Steve Freeman. Growing object-oriented software, guided by test. Addison Wesley; 2009

20 Robert Cecil Martin. Screaming architecture [online]
URL: https://8thlight.com/blog/uncle-bob/2011/09/30/Screaming-Architecture.html Accessed 10 April 2017

21 Jeffrey Palermo. The onion architecture [online]
URL: http://jeffreypalermo.com/blog/the-onion-architecture-part-1/ Accessed 10 April 2017

22 Brett L. Schuchert. DIP in the Wild [online]
URL: https://martinfowler.com/articles/dipInTheWild.html Accessed 10 April 2017

23 Stephen Blackheath, Anthony Jones. Functional Reactive Programming. Manning Publications; July 2016.

24 Concurrency. Oracle Java Documentation [online].
URL: https://docs.oracle.com/javase/tutorial/essential/concurrency/ Accessed 11 April 2017

25 Automating User Interface Test [online]
URL: https://developer.android.com/training/testing/ui-testing/index.html Accessed 11 April 20