POLITECNICO DI TORINO

Department of Electronics and Telecommunication (DET)
Master Degree Program in Engineering

Communication and Computer Networks

Master Degree Thesis

Modern architecture for testing mobile applications

Supervisor

Prof. MALNATI GIOVANNI (DAUIN)

Candidate

Saeed Farkhondeh

July 2020

ACKNOWLEDGMENTS

First, I greatly appreciate all efforts of my dear professor Giovanni Malnati that his effective guide conducts all my activities in the right direction.

Also, I would like to express my sincere gratitude to all friends who working in Synesthesia Company because of their useful and technical help in terms of testing applications. Their advice and tips significantly gained my knowledge in this field. Furthermore, I thank all managers of this company who provided a friendly atmosphere and needed tools for doing my project.

Moreover, I am grateful for friends who shared their valuable experience with doing a thesis which it was useful to do my thesis appropriately.

Finally, I am so much thankful for my parents' support which encouraged me to do my best for this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

Figure **PAGE**

Summary:

This thesis topic describes one general approach for testing android applications which its foundation is based on one of the android and iOS architectures.

In other words, this thesis topic targets this goal to figure out which application is compatible and match with this architecture. In order to reach this aim, we will check architecture and the functionality of each layer in this architecture and the rule of interactions between different layers. All these things will be done by using some tools and implement some methods for testing which describe in several steps and mentioned in below lines:

- We will test the architecture to confirm it is based on the architecture that we considered for developing our app.
- We will demonstrate why, what, how, and when we are going to test.
- During this research, we will investigate the methods of testing including unit, UI, and integration test.
- Also, we will exploit Android Studio IDE, Gradle, Xcode (with swift language) for our test project.
- Moreover, we will clarify the Clean Architecture and its layers.
- For implementing different kinds of methods for testing, we will use such tools like Mockito or espresso. The main result of this thesis is about how we can use this kind of tool for an application architecture such as Clean Architecture.

Chapter 1

Introduction

In mobile application designing, the testing part helps to avoid lots of manual testing. The more comprehensive testing code, the higher chance for discovering hidden bugs.

Besides, writing tests for applications will give us a better notion to estimate the requirements and also for bug detection. Furthermore, it is not possible to write a test for a part of the code without knowing about its functionality.

Moreover, for testing applications, it is better to have an automated test. So, we can run a part of the test or all part of the test again to be sure that after every change in code, we have the same condition as before and this new code will satisfy all the tests again. After that, we can introduce Continuous Integration as part of the development process.

Generally, we can test all the parts of code, but instead of testing all parts of code, it is feasible to reduce testing into a few parts of code and just consider some key methods and functions for testing. In this thesis, we just focused on testing the functionality of each layer and correlated methods.

We chose this topic because it is a critical part of designing applications and it will help us to enhance and maintain the android applications. In other words, the benefits of testing applications are to decrease the maintenance coast of software and gain productivity.

1.1 Architecture of an app and modularity feature

On the other hand, through a testing approach, we can check the modularity of an application. Why do we need to do that?

Well, the answer to this question refers to this fact that why we would need to design an architecture for our app and make it modular.

One of the main and important points about designing mobile applications is that the designers have to launch the app components individually and out of order and the operating system or user can destroy them at any time. Because the events related to ending the lifecycle of these components are not under our control, and we shouldn't store any app data or state in our app components, and our app components shouldn't depend on each other.

Now, a critical question arises. If it is not good to use app components to save app data and state, how should we design our app? The solution related to the "Separation of concerns" (SoC) concept in computer science which leads to the design principle for dividing a computer program into distinct sections.

Generally, it is not correct to write all our codes in an activity, and the user interface classes should only contain logic which has a task to represent UI and also the interactions with the operating system. Therefore, by presenting these classes in short form, we could avoid many lifecycle problems. Moreover, to provide user satisfaction and create a manageable application, it is better to reduce our dependency on Activities and Fragment classes.

Overall, by designing our app base on a model of classes with the well-defined functionality of managing the data, our app will be more modular and consequently, it can be more testable and consistent.

1.2 Tools needed for testing

Also, in this chapter, we will mention the requirements needed for testing applications and the efficient tools for this aim. Moreover, we will justify our approach for the testing base on a well-known architecture called "clean architecture".

- Bash script: a part of this thesis will be done by writing a bash file. In order to test the architecture design of projects.

- Mockito: one of the sophisticated tools for testing applications in terms of unit test is the Mockito framework. Through this tool, we can create a mock object for an interface or a class, and also we will define the expected output value to be compared with the output of functions.

- Espresso: this framework is a powerful tool in android for user interface testing. Google designed Espresso framework testing in Oct.2013 and it is a part of the Android Support Repository.

- XCTest: it is a testing framework that allows us to generate and execute unit and UI tests for our Xcode project. More likely to Espresso and Mockito, it demonstrates that if specific conditions are satisfied during code execution, and show us test failures if those conditions are not satisfied.

- Rxjava: in the company projects, they exploit widely Rxjava libraries for composing asynchronous programs by using observable sequences.

- Clean architecture: mobile applications in a company designed based on an architecture called Skeleton (clean architecture). This architecture composed of 5 different layers with different tasks including entities layer, use case layer, data layer, frameworks layer, and presentation layer.

1.3 General technique for testing

Generally, the testing techniques that imply different test inputs are categorized in 4 different groups including 1- model-based, 2- symbolic execution, 3- combinational testing, 4- random, and pseudo-random testing.

The most relevant findings for testing approaches refer to the improvement of the model-base technique.

1.4 General form of testing in practice

In the general form of testing, we will give the fake data as an input to a class that has specific functionality, and then we will get output. This output would be compared with the expected value that we expect of this class. If the expected value and output match together, so it is reasonable to conclude that our class passed the test successfully, otherwise, we should say there is a problem or a bug. Finding the bug or problem is related to another process which is debugging or troubleshooting of app.

1.5 resources and references

In order to get the result, we exploited the company documents and resources and also the experience of experts working in this company. Moreover, some provided documents and research in this field provided by google and Github were useful.

Chapter 2

Related literature and theoretical focus

In this chapter, we will go through different literature which refers to the main concepts needed for explaining the testing approaches.

(i) Agile and waterfall method for project management
(ii) BDD approach for developing
(iii) TDD approach for developing
(iv) Clean architecture
    (a) Dependency rule
    (b) Dependency inversion principle
    (c) Entities layer
    (d) Use case layer
    (e) Data layer
    (f) Frameworks and Drivers layer
    (g) Presentation layer

(v) Instrumentation tests
(vi) Android Unit test
(vii) Test Components
(viii) Mocking

2.1 Agile and waterfall method for project management

The agile approach is one of the most effective methods for cooperation between mobile developers and continuous communication between members of the team and customers in a project.

In this method, the whole process will break into many sub-tasks which each of these sub-tasks will be considered as a mini-project for the development team.

On the other hand, the waterfall approach is a sequential design process. By that means, we have eight stages for development (conception, initiation, analysis, design, construction, testing, implementation, and maintenance) and when each of these stages completed, the developers can start another stage.

One of the differences between the two methods is their approach to quality and testing. The "Testing" phase will be performed after the "Built" phase in the waterfall method, but, in the Agile method, the testing part will be done concurrently with the programming part.

2.2 BDD approach for developing

The BDD (Behavior Driven Development) approach consists of three steps including Given-When-Then. The Given step refers to a specified scenario and the When step declares an action that takes place. In the last step ("Then" step), we ensure that the new state of the system is correct or we will check some behavior as a result of the system. Also, in this approach, we are no longer defining 'test', but we are defining 'behavior'. Furthermore, there is another advantage of this method which is better communication between developers, testers, and product owners.

2.3 TDD approach for developing

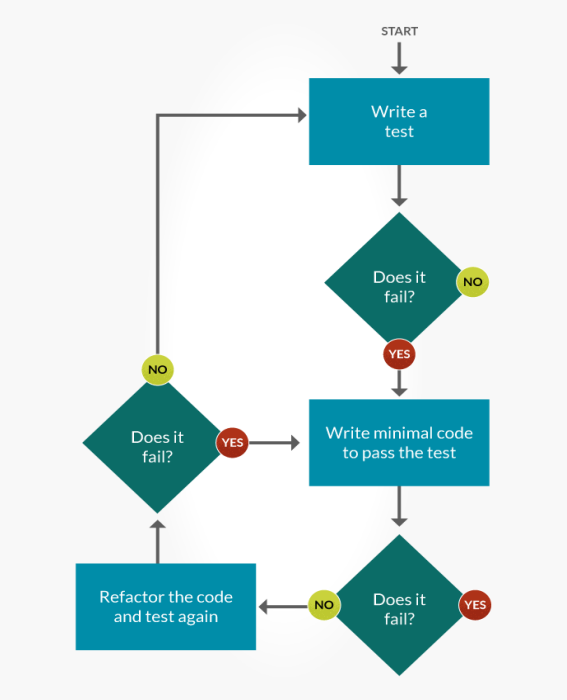


Figure 2.3: TDD algorithm.

ln TDD (Test Driven Development) approach, the coding will be done before testing compared with the common development approach. With the TDD method, we will write tests upfront for functions that don't yet exist. Also, we know that there is a high possibility of failing the test at the beginning of the testing process, but by coding more at each stage, we will make sure that all classes and methods will pass the test at some points.

2.4 Clean architecture

Clean architecture code is a software designing approach which separates the elements of design into ring levels. In this architecture the outer circles include mechanisms and the inner circles are about the policies. More similar to other

software design philosophies, clean architecture has this aim to provide a structure for coding in order to make it easier for developing.

The clean architecture is proper for testing because of regularity for functions inside each layer. Therefore, we can write a test for functions in each layer base on its task.

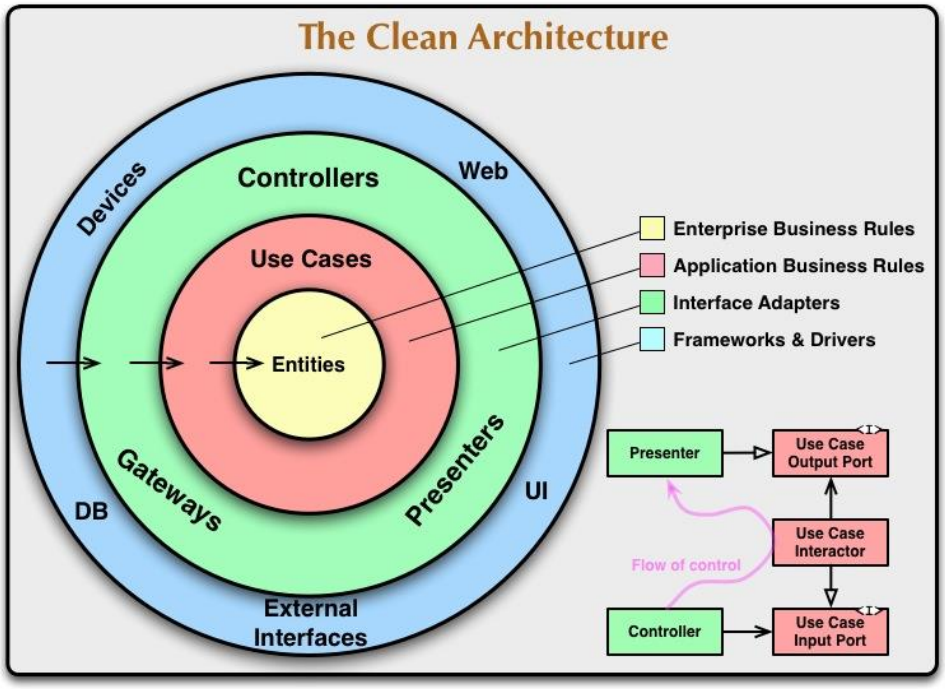


Figure 2.4: Clean architecture

2.4.1 Dependency rule

One of the main rules of clean architecture is code dependencies which imply that source code can only point to the inner circle of this architecture. By that means, all the things inside the inner circle are not allowed to know anything at all about something inside an outer circle. This implies that if something which defined in an outer layer must not be used by the code in the inner layer. That includes functions, classes, variables, or any other software entity.

Another important thing about the dependency rule is that as we move towards inner layers the level of abstraction will increase, so it is reasonable to expect low-level concrete details for outermost circles in this architecture.

2.4.2 Dependency inversion principle

The main question about clean architecture and its dependency rule is that how an inner layer (for example Data layer) can communicate with an external layer?

This is possible through the dependency inversion principle which includes two general rule:

A. High-level modules should not rely on low-level modules. Both should depend on the abstraction layer.
B. Abstractions should not depend on details, instead, details should depend on abstractions.

Usually, for Dependency Rule, we inject the dependency (we can use Dagger or kodein for this aim) or add the dependency inside the Gradle. But for Dependency Inversion, we should define an interface (refers to case B which mentioned to "depend on an abstraction")

2.4.3 Entities layer

In this architecture, an entity can be an object with methods, or it can be a set of data structures and functions. Entities are including the most general rules. For instance, in one of the company projects, they used the Movie class and the Subtitle class for showing the list of movies. Ideally, it should be the biggest layer,

though it's correct to say that Android Apps usually tend to just use an API in the screen of a phone, so a great portion of core logic will just compose of requesting and persisting data.

2.4.4 Use case layer

Usually, this layer called interactors because it describes mainly the actions that the user can cause. This kind of action categorizes into two groups that can be active actions (the user clicks on a button) or implicit actions (the App navigates to a screen).

2.4.5 Data layer

A set of adapters defined in this layer that convert data from a suitable for the use cases and entities to the format proper for some external components such as the Database or the Web. The Controllers and presenters all defined here. The models are passed from the controllers to the use cases and then return from the use cases to the views and presenters.

2.4.6 Frameworks and Drivers layer

This layer **(**Driver layer**)** encapsulates the interaction with the framework so that the rest of the code can be reusable in case we want to exploit the same App on another platform. With the framework layer, we are not only referring to the

Android framework, but to any external libraries that we want to deform in the future.

2.4.7 Presentation layer

This is the layer that communicates data with the UI (Fragments & Activities) for display data. For example, in this layer, that will wholly include the Model-View-Controller pattern of a GUI.

2.5 Instrumentation tests

This kind of test will run on Android devices and emulators instead of running on JVM. Also, these kinds of tests have access to the mobile phone and its resources and are useful to unit test functionality which it is not possible to be mocked by mocking frameworks.

In this approach, the foundation for the test is InstrumentationTestRunner which will initial and load other test methods. It can interact with the Android system through the instrumentation API.

2.6 Android Unit test

In this type of test, we verify that the logic of individual units is correct. Usually, unit testing makes use of object mocking. Mock objects are created and configured to present a certain behavior during testing. In fact, in the unit tests, we will create a mock object to separate each unit of code from its dependency in order to have different parts for test and repeat the test any number of times. Also, Junit is a standard tool for unit tests on Android.

Typically, we are using some methods for the unit test including the setUp() method, tearDown() method, and test method.

Inside the setUp() method, we are going to initialize the test and the related code state.

Inside the tearDown() method, we can release resources that we used for test and it will be invoked "after" every test.

All the methods in which their names start with the “test” keyword will be considered as a test method. When the test method executes the other methods inside itself, it will return some values that should be compared with the expected value. JUnit provides a set of methods (“assert”) for this comparison and it will issue an exception if the conditions are not met.

2.7 Test Components

This kind of test includes several parts for different components like test “activity” (usually use Espresso), test “service” component, and test “content provider” component.

For the testing service component, we use the ServiceTestRule class provided by the Android Testing Support Library. This rule provides a simplified mechanism to trigger and shut down your service before and after your test.

2.8 Mocking

Unit testing also makes use of object mocking. In this case, the real object is exchanged by a replacement which has a predefined behavior for the test. Mock objects are configured to perform a certain behavior during a test.

Chapter 3

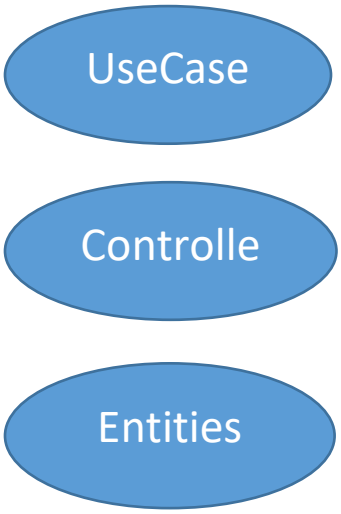
Testing approach for architecture

We will implement our method for testing through using one of the demo projects which is based on clean architecture. This application has a task to retrieve movie data from a remote server and present this information on the screen. There are 4 different layers defined in this project, as we expected for clean architecture.

The first step of our testing is related to checking the architecture of the application. In order, to reach this goal, we run a program that can be implemented for both IOS and android. The programming language used for this aim is the bash script. The shell file will be executed by the Gradle in android studio and also it runs by building phases in Xcode.

This bash script program will target two main features of clean architecture for testing including the number of layers and the dependency rule between these layers.

For checking the number of layers, we will query base on the name of layers in the group of classes name and for dependency rule. The key point for checking the dependency rule is to extract the name of classes in a higher layer and search in the lower layer classes for an object with the same name.



- STEP 1: determine how many layers do we have based on the architecture that we are going to use in our project?!
- STEP 2: Check the rules between layers.

3.1 IOS layers in the architecture

For example, for the IOS layer, we can check the group of class's base on the name of these groups, and then we can count the number of layers.

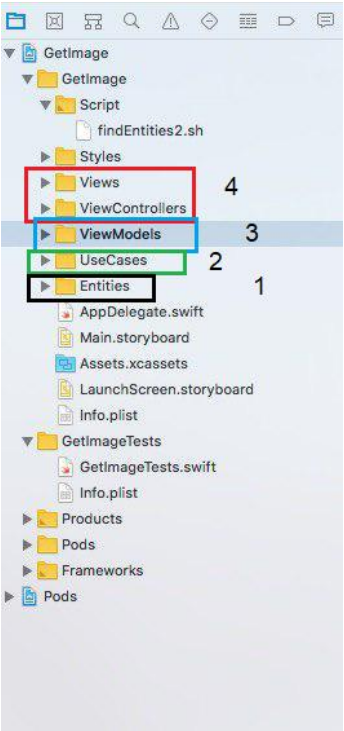


Figure 3.2: Layers of architecture in IOS app

As we can see, in the above hierarchy of groups of classes Views and ViewController, UseCases, Entities are 4 different layers of clean architecture that considered for an application. However, these names could change for every layer depend on the programmer who wants to choose a name, but generally, in a company which works with an agile approach, they would choose the same names for layers in their project. Also, the script for checking the number of layers is changeable and easily could be adapted for different names for layers.

3.2 Android layers in the architecture

Also, for android, we have a group of java or Kotlin classes and we can use their names for checking the architecture.

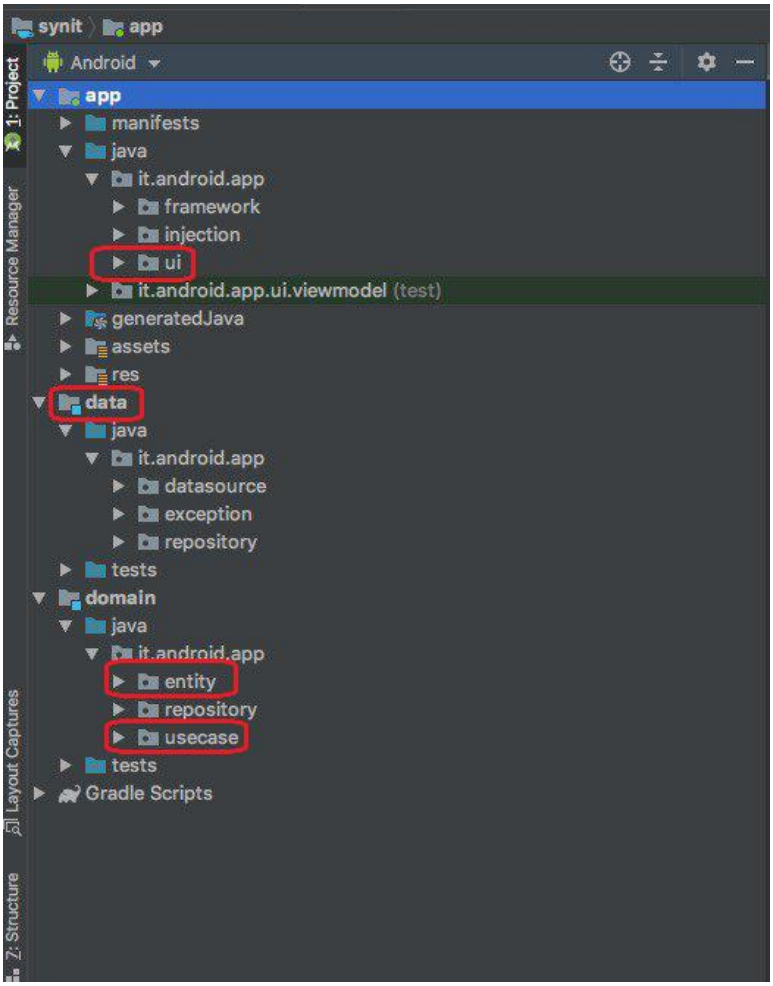


Figure 3.2: Layers of architecture in Android app

3.3 Execute a bash file by android studio SDK and Xcode

For executing the bash file by Xcode, we will add the bash file in the TEST SCRIPT part of the project.

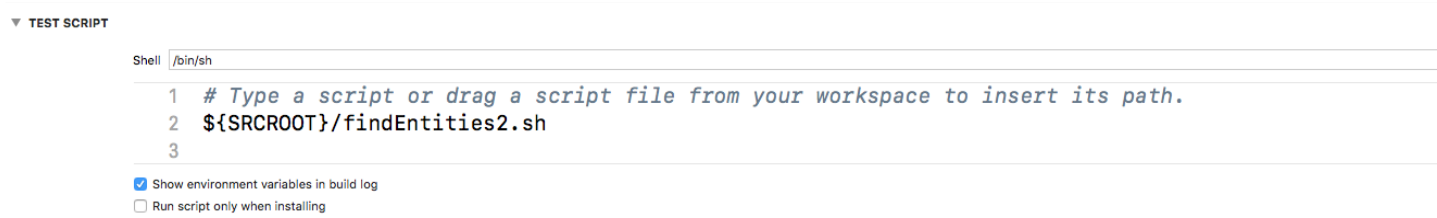


Figure 3.3.1: Running a script in Xcode

Moreover, for executing the bash file in Android studio, we can add it in Gradle that we called it here "testArchitectureAndroid.sh"

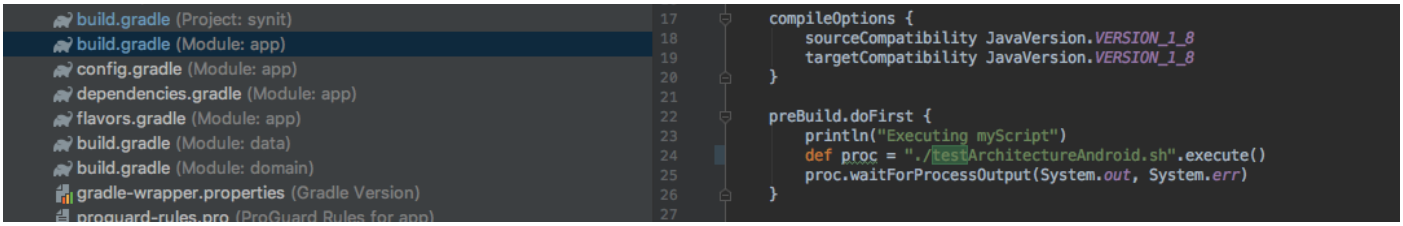


Figure 3.3.2: Running a script in Android

3.4 Test number of layers in the architecture

Now, the question is, how this piece of code would check the number of layers?! Well, simply when the bash code is running by Gradle or Xcode, it will start searching between groups of classes for common names that usually developers are going to use in the structure of code. It is a list of possible names for each layer and our script is going to search in all subfolders these names. If it success to find the names, then it will confirm that the layer exists. If the number of layers is not matched with the expected number of layers in our architecture, the test would fail and print a message in the log bar.

3.4.1 Test number of layers for Android

```
8   array_for_variables_name=(var val let)
9   array_for_layer1=(Entity entity Entities entities)   ## use an array to create a list of possible names for layer 1
10  array_for_layer2=(usecase Usecase UseCase)
11  array_for_layer3=(data Data Controllers controllers controller Controller Presenters pre
12  array_for_layer4=(UI ui userInterface UserInterface Main.storyboard)
13
14  ############################### Step 1: find the layer 1 based on its common name ######
15
16  for layer1_names in "${array_for_layer1[@]}"
17  do :
18
19  if [ "$flag" -eq "1" ]
20  then
21  break
22  fi
23
24  array1=(`find $(pwd) -type d -name ${layer1_names}`)
25
26  for i in "${array1[@]}"
27  do :
28     if test -d "$i"; then
29     flag=$((flag+1))
30       echo "Layer Entities exist"
```

Figure 3.4.1: Checking the number of layers in Android

In the above code, for android, we checked the number of layers. For example, the number of common names for using in layer 1 highlighted.

```
103         echo "Layer User Interface exist"
104         break
105       else
106         echo "Layer UserIInterface is missing"
107       fi
108     done
109     done
110
111     ###############################################################################
112     if [ "$flag" -lt "4" ]
113     then
114         echo -e "**** Warning: THIS APPLICATION IS NOT MATCH WITH THE SKELETON ARCHITECTURE
115     else
116         echo "**** NOTE: APPLICATION IS MATCH WITH THE SKELETON ARCHITECTURE ****"
117     fi
118
119     ############################## Dependency rule for android LAYER 2 ######################
120
121     ## The public variables and module related to their classes
122     for i in "${array2[@]}"
123       do :
124           if test -d "$i"; then
125
126       nameOfClasses2=(`find "$i" -type f -name "*.kt" | awk -v FS="" '{print $1}' | awk -
```

Figure 3.4.2: Print the result of the test in the script

In the previous part of the script, we will issue a message as a warning or a note to print the result for testing.

3.4.2 Test number of layers for IOS

If we check the below code, obviously we can see, the peace of cod for the IOS part is slightly different from the Android part.

```
#!/bin/bash

flag=0
flagMatch2=0
flagMatch3=0
flagMatch4=0
prefix="$PWD"
array_for_variables_name=(var val let)
array_for_layer1=(Entity entity Entities entities)
array_for_layer2=(usecase UseCases)
array_for_layer3=(data Data ViewModels)
array_for_layer4=(ViewControllers Views)

############################### Step 1: find the layer 1 based on its common name
    ##############################################

for layer1_names in "${array_for_layer1[@]}"
do :

if [ "$flag" -eq "1" ]
then
break
fi

array1=(`find $(pwd) -type d -name ${layer1_names}`)

for i in "${array1[@]}"
do :
   if test -d "$i"; then
   flag=$((flag+1))
     echo "Layer Entities exist"
     break
    else
     echo "Layer Entities is missing"
  fi
done
done
```

Figure 3.4.3: Checking the number of layers in IOS

3.5 Test dependency rule between layers

Now, we are ready to check the dependency rule between all these layers. So, based on the dependency rule, we will search inside the inner layer classes the objects related to the outer layer classes. If the result of the search is positive, then testing for the dependency rule is failed. Otherwise, if there is not any object related to outer layer classes inside the inner layer, thus the dependency rule testing is successful.

Moreover, we don't need to check the dependency rule for the first inner layer.

3.5.1 Dependency rule for Android

```
118
119     ############################## Dependency rule for android LAYER 2 ######################
120
121     ## The public variables and module related to their classes
122     for i in "${array2[@]}"
123       do :
124            if test -d "$i"; then
125
126          ## we extract the name of classes here in layer 2.
127          nameOfClasses2=(`find "$i" -type f -name "*.kt" | awk -v FS="." '{print $1}' | awk -
128
129              fi
130          done
131
132               ##### query in layer 1 #####
133          for lp in "${array1[@]}"  ## name of directories which their names refers to the lay
134                   do :
135
136             for className in "${nameOfClasses2[@]}"
137             do :
138                   if grep -r -q "$className" "$lp"   ## query in all files in that directory
139                then
140               flagMatch2=1
```

Figure 3.5.1: Checking dependency rule for Android

As we can see, our testing for dependency rule starts from layer 2 and first extract the name of classes in this layer. Then, in the next step, we will search any object based on the names of these classes only in layer 1.

```
134             do :
135
136           for className in "${nameOfClasses2[@]}"
137           do :
138                 if grep -r -q "$className" "$lp"   ## query in all files in that directory
139                then
140               flagMatch2=1
141               fi
142         done
143          done
144
145       if [ "$flagMatch2" -eq 1 ]
146          then
147          echo "WARNING: Dipendency rules is not satisfied for LAYER 2"
148       else
149           echo "Dipendency rules is OK for LAYER 2"
150       fi
151    ############################## Dependency rule for android LAYER 3 ######################
152
153    ## Determine the module of layer 3. Is it "data", "app" or "domain"
154    ## determine the module of other layers
155    ## Find words of the file in different layers and then compare the modules of that layer
156
157    for i in "${array3[@]}"
```

Figure 3.5.2: print dependency rule for Android

As we can observe in the previous code, the value will be set to 1 for “flagMatch2”, if there will be any object inside layer 1 (inner layer) related to layer 2 (outer layer). Finally, the code will issue a message based on the value of this flag.

There is also another point, which is about the pattern for objects defining by different programming languages. For example, in Kotlin language we have these prefixes for creating a new object of classes such as “.kt”. This could be also modified for different languages.

3.5.2 Dependency rule for IOS

Also, for IOS, by looking at the code for dependency rule between layers, we figure out that the implemented algorithm is as same as the Android part.

```
119 ############################## Dependency role for android LAYER 2 ###################################
120
121 ## The public variables and module related top their classes
122 for i in "${array2[@]}"
123   do :
124         if test -d "$i"; then
125
126         nameOfClasses2=(`find "$i" -type f -name "*.swift" | awk -v FS="." '{print $1}' | awk -v FS="/"
              '{print $NF}'`)
127
128         fi
129     done
130
131          ##### query in layer 1 #####
132     for lp in "${array1[@]}"  ## name of directories which their names refers to the layer name
133               do :
134
135         for className in "${nameOfClasses2[@]}"
136        do :
137               if grep -r -q "$className" "$lp"   ## query in all files in that directory for the same name
                      of variables and the specific pattern
138            then
139           flagMatch2=1
140          fi
141       done
142       done
143
144    if [ "$flagMatch2" -eq 1 ]
145       then
146       echo "WARNING: Dipendency rules is not satisfied for LAYER 2"
147    else
148        echo "Dipendency rules is OK for LAYER 2"
149    fi
```

Figure 3.5.3: Checking dependency rule for IOS

In the above figure for testing dependency rule between layers in IOS, we tried to find the classes with suffixes ".swift", then extract the name of the class for search in other classes as an object.

Chapter 4

Unit and UI test for each layer of architecture

In this chapter, mainly we will work on the “UI” and unit “test”. Moreover, we will employ our methods for testing by using one of the projects designed based on clean architecture. More likely to chapter 3, this application retrieves data for movies from a remote server and shows this information. But before going through codes and analyze each layer, we need to know about Rxjava and its functionality in this project.

4.1 Implementation of Rxjava

Rxjava allows us to form the reactive components in Android applications. By that means, it provides a scheduler that schedules on the main thread or any given looper for asynchronous tasks.

In the framework and driver layers of this project, we defined a class "RemoteDataSourceImpl" to retrieve data from the remote source and display it on the UI. There are some descriptions about the classes that we used and later, we will mock them in the testing part.
One of the functions used in this class is “Observable” which has a task to observe the outcome on the main thread.

For example, in a class:

```
Observable.just("one", "two", "three", "four", "five")
        .subscribeOn(Schedulers.newThread())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(/* an Observer */);
```

This will run the Observable on a new thread, and send outcomes via onNext on the main thread.

Another function is flatMapIterable which maps each item receiving from the server-side into a list of items.

Also, “compose (traktShowToWatchableEntity())” transforms an observable source by applying a particular transformer function to it.

Generally, the first function in this class is used to get a single pattern from a remote source through an Id (getshow()) and then convert it to an observable object (in order to be showed elements of this pattern via main thread). In the next step of this function, we will transform (compose) these observable sources. Finally, we will emit (singleOrError) them one by one.

4.2 Mockito implementation for test

Now, we are going to write a test for this function. In the Test class, we first Mock the two interfaces to generate the fake response from the server side later in the “RemoteDataSourceImpl” class. This will be done by using @Mock annotation for these interfaces.

```
@Mock
lateinit var traktApi: TraktApi
@Mock
lateinit var tmdbApi: TmdbApi
```

Figure 4.2.1: Mocking an interface

In the next step, before running any test case (after @Before annotation), we will populate the annotated fields via MockitoAnnotations.initMocks and create an object of a class that we want to test (here RemoteDataSourceImpl ()).

```
@Before
fun setUp() {

// If you use the @Mock annotation, you must trigger the creation of annotated objects.
// The MockitoRule allows this. It invokes the static method MockitoAnnotations.
// initMocks(this) to populate the annotated fields.
    MockitoAnnotations.initMocks( testClass: this)
    // create an object of class that we want to test
    remoteDataSourceImpl = RemoteDataSourceImpl(traktApi, tmdbApi,
            watchableMapper, showMapper)
}
```

Figure 4.2.2: Initial a mocking object

Afterward, we will test the function through @Test annotation based on the BDD approach which has 3 main steps Given-When-Then.

```
@Test
fun getWatchableTest() {
    // TRAKT
    // Create a fake response from server side
    // this is the "GIVEN" part in BDD approach which create a scenario
    val traktShowResponse = Show(
            title = "MyTitle",
            year = 2019,
            ids = Ids(
                    trakt = 1,
                    tmdb = 2),
            overview = "MyOverview")
```

Figure 4.2.3: “Given” part of a test

In the When and Given part of test approach, we have

```
// This is the "WHEN" and "THEN" part in BDD approach (an action takes place)
// & (this should be the outcome) The when(….).thenReturn(….) method chain
// is used to specify a return value for a method call with pre-defined parameters.
whenever(traktApi.getShow(any())).thenReturn(Single.just(traktShowResponse))
```

Figure 4.2.4: “When” part of the test

Moreover, we will generate the expected value to compare with the returned values from class.

```
//EXPECTED MAPPED ENTITY (the things that we expect to watch)
val expectedEntity = WatchableEntity(
        id = 1,
        title = "MyTitle",
        year = 2019,
        overview = "MyOverview"
)
```

Figure 4.2.5: Expected values for comparison

Finally, we will compare the returned value and the expected one.

```
testObserverWatchable
        .assertNoErrors()
        .assertValue(expectedEntity)
```

Figure 4.2.6: Compare the expected with the result

The result of this simple test if the expected value will be matched with the returned one, we will get a match mark. Otherwise, the testing will fail and it will return exception error with code 1.

4.3 Using Espresso for UI test

In terms of testing UI with espresso, there is a simple example of the project for login.

In this class, we have two edit text fields for email and password. In the testing class, we will set the value for them and then check if the button for login is enabled or not.

```
    @Test
    fun onCreate() {

        // input some text in the edit text for Email and password
          Espresso.onView(withId(R.id.et_email)).perform(typeText(username))
          Espresso.onView(withId(R.id.et_password)).perform(typeText(password))

        // close soft keyboard
        Espresso.closeSoftKeyboard()

        // check the button is enabled or not
          Espresso.onView(withId(R.id.btn_login)).check(matches(not(isEnabled())))
    }
}
```

Figure 4.3.1: Espresso for UI test

4.4 UseCase layer testing

In another case, by testing a class in use case layer, we can check the fidelity of its functionality. Furthermore, this class in the usecase layer has a task to search a keyword for a specific title or description related to the list of movies.

Like the previous procedure, at first, we mock the class that we want to use in our test.

```
class GetTrendingShowsUseCaseTest {

    @Mock
    lateinit var entertainmentRepository: EntertainmentRepository

    val schedulerProvider = TestSchedulerProvider()

    private val testGetTrendingObservable =
            TestObserver<List<EntertainmentEntity.ShowEntity>>()

    lateinit var getTerendingShowCase: GetTrendingShowsUseCase
```

Figure 4.4.1: using Mockito for testing “UseCase” layer

Then, we will initiate the values in setUp() and after Test annotation, we will define the expected value and the fake value as an input.

```
@Before
fun setUp() {

    MockitoAnnotations.initMocks( testClass: this)

    getTerendingShowCase = GetTrendingShowsUseCase(
            schedulerProvider, entertainmentRepository)
}
```

Figure 4.4.2: Initial a mock object for UseCase layer

```
@Test
fun filter() {


    val myList : List<EntertainmentEntity.ShowEntity> = listOf(EntertainmentEntity.ShowEntity(
            id = "1",
            title = "ForCheck A",
            backdropImageFileName = null,
            posterImageFileName = null,
            year = 2009,
            description = "this is a description for test",
            seasons = null
    ),
            EntertainmentEntity.ShowEntity(
                    id = "2",
                    title = "ForCheck B",
                    backdropImageFileName = null,
                    posterImageFileName = null,
                    year = 2009,
                    description = "this is a description for test",
                    seasons = null))
```

Figure 4.4.3: “Given” part of the test for UseCase layer

Finally, we will compare the returned value and the expected one.

```
whenever(entertainmentRepository.getShowsTrendingPaginated(any()))
        .thenReturn(Single.just(myList))


//EXPECTED MAPPED ENTITY (the things that we expect to watch)
val filteredListExpected : List<EntertainmentEntity.ShowEntity> = listOf(
        EntertainmentEntity.ShowEntity(
        id = "1",
        title = "ForCheck A",
        backdropImageFileName = null,
        posterImageFileName = null,
        year = 2009,
        description = "this is a description for test",
        seasons = null
))

//test without filter params and check if the list that you receive is the entire
val paramsWithoutFilter = GetTrendingShowsUseCase.Params( page: 1)
getTerendingShowCase.execute(paramsWithoutFilter)
        .subscribe(testGetTrendingObservable)
testGetTrendingObservable.assertValue (myList)
```

Figure 4.4.5: “When” and “Then” part of the test for UseCase layer

4.5 Data layer testing

In another case for testing different layers, inside the data layer of this application, we have a class that its functionality is to receive data from the cache, and then if any error happens, it will try to retrieve data from the remote data server. In this case, the testing part only tests the calling functions for 2 different states of the cache. In the first state, we get a Cache exception and data (list of single items) retrieved from the remote data source. In this state, we will check if the function related to the remote data source called.

```
@Test
fun list_notCached() {

    whenever(remoteDataSource.getMovieTrendingPaginated( page: 4))
            .thenReturn(Single.just(testSampleList.map { it.id }))

    whenever(localDataSource.getMovie(any()))
            .thenReturn(Single.error(CacheException.NotFound("Cache not implemented")))

    whenever(remoteDataSource.getMovie(any()))
            .thenReturn(Single.just(testSampleList[0]))

    whenever(localDataSource.cache(any()))
            .thenReturn(Single.just( item: true))


    repository.getMovieTrendingPaginated( page: 4)
            .doOnError { it.printStackTrace() }
            .subscribe(testObserver)

    testObserver.assertValueCount(1)
}
```

Figure 4.5.1: Test for Data layer – getting cache exception

In the second state, we are not getting a cache exception and also we would never call the function related to the remote data source.

```
@Test
fun list_cached() {

    whenever(remoteDataSource.getMovieTrendingPaginated( page: 4)).
            thenReturn(Single.just(testSampleList.map { it.id }))

    whenever(localDataSource.getMovie(any())).thenReturn(Single.just(testSampleList[0]))
    whenever(remoteDataSource.getMovie(any())).thenReturn(Single.just(testSampleList[0]))
    whenever(localDataSource.cache(any())).thenReturn(Single.just( item: true))

    repository.getMovieTrendingPaginated( page: 4)
            .doOnSuccess { entity : List<EntertainmentEntity.MovieEntity>! ->
            }.subscribe(testObserver)

    remoteDataSource.inOrder { verify(remoteDataSource, never()).getMovie(any())    }

    testObserver.assertValueCount(1)
}
```

Figure 4.5.2: Test for Data layer – without a cache exception

4.6 Unit test for IOS

Now, we are going to do a "Unit test" in the IOS app. This "Unit test" will be performed by XCTest for asynchronous cod (specifically, network operations and using a URL for retrieving data).

For this aim, we will use the XCTestExpectation class which performs long-running tasks or background tasks, then wait for these tasks to satisfy expected conditions.

After adding a new "Unit Test Case Class" to our project which called "GetImageTests", we will import the GetImage app below the "import XCTest".

```
 9  import XCTest
10  @testable import GetImage
```

Figure 4.6.1: Implement XCTest framework in IOS for testing

Moreover, we are going to define an object named "urlUnderTest" inside setup() function, in order to send a request to a server that contains information about movies and series. Later, we will release this object in the teardown() function.

```
class GetImageTests: XCTestCase {

    var urlUnderTest: URLSession!

    override func setUp() {
        // Put setup code here. This method is called before the
            invocation of each test method in the class.
        urlUnderTest = URLSession(configuration:
            URLSessionConfiguration.default)

    }

    override func tearDown() {
        // Put teardown code here. This method is called after the
            invocation of each test method in the class.

        urlUnderTest = nil
    }
```

Figure 4.6.2: Define an URL object for testing

Now, we can add an asynchronous part in our test code.

```
    // Our Asynchronous test function
    func tvServerCalling() {
        // given
        let url = URL(string: "https://api.tvmaze.com/")
        // 1
        let weExpect = expectation(description: "Status code: 200")

        // when
        let dataTask = urlUnderTest.dataTask(with: url!) { data, response, error in
            // then
            if let error = error {
                XCTFail("Error: \(error.localizedDescription)")
                return
            } else if let statusCode = (response as? HTTPURLResponse)?.statusCode {
                if statusCode == 200 {
                    // 2
                    weExpect.fulfill()
                } else {
                    XCTFail("Status code: \(statusCode)")
                }
            }
        }
        dataTask.resume()
        // 3
        waitForExpectations(timeout: 5, handler: nil)
    }
```

Figure 4.6.3: Implement XCTestExpectation for testing

In the above code, we check if sending a simple request to the “tvmaze” server return with status 200 then our expectation will satisfy (fulfill) and our test will be successful. Otherwise, with another status code, the test will be failed. However, if get a local error from the server, it will be considered also.

Chapter 5

Conclusion and Bibliography

5.1 Conclusion

In this thesis, we tried to figure out how much our method for testing can affect development team performance for designing applications in both agile and waterfall approaches.

Well, after applying this method for testing and first checking the architecture of an application in advance, the development part and also putting different layers of the application under test would be less time consuming for all team members. Moreover, project managers have a better chance to conduct team efforts and monitor the development process. Later, after the delivery of products, it is completely obvious what is the structure and rules between different classes (since we test the application architecture).

However, this approach has some drawbacks also, such as increasing the complexity of code for testing architecture when the number of languages using in development increases.

Overall, using architecture for the development and follow a procedure for development would lead to a better quality for the product, faster debugging, more objective testing, and maintenance.

5.2 Bibliography

[1] Paul Blundell, Diego Torres Milano “Learning Android Application Testing”

[2] Synesthesia documents, “International Software Testing Qualification Board”

[3] Synesthesia documents, “Mobile Architecture” [1] Paul Blundell, Diego Torres Milano “Learning Android Application Testing”

[4] https://developer.android.com/topic/libraries/architecture/

[5] https://github.com/synesthesia-it/Murray

[6] https://developer.apple.com/

[7] https://www.raywenderlich.com/960290-ios-unit-testing-and-ui-testing-tutorial