Expertise
and insight
for the future

Hung Nguyen

Biometertest – Android application for improving test automation coverage of Contour biometer driver

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

14 April 2021

Author Title Number of Pages Date	Hung Nguyen Biometertest – Android application for improving test automation coverage of Contour biometer driver 47 pages 14 March 2021
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Mobile Solutions
Instructors	Kari Salo, Principal Lecturer Rami Sirenius, Team Manager R&D

Improving test automation coverage is one of the most important practices in a technology company. To develop a good quality product, automated testing should be a crucial factor in the big picture of any software development process. This thesis mainly focuses on the Biometertest application, which is an application utilizing the latest Android technologies to improve the testing work in Signant Health company.

In the beginning, there was a need to test the Contour Next One biometer via its driver interface methods in a way that reduces the test execution time, enhances repeatability, saves testing effort, and introduces a better way to develop and debug the Contour jar driver. Therefore, the Biometertest application was developed to aim at solving the above issues.

Although few possible improvements for the application in the future still exist and many knowledge transfer sessions have to be conducted to increase the application's usability on the company level, there is a strong belief that this solution will be widely used by a large majority of company's employees based on its outweighed strengths and advantages.

In the end, the application was successfully developed and used by the company's internal personnel. With the assistance of the Biometertest, there is a noticeable reduction in testing time for the specific Contour meter executed by validation specialists. This has contributed greatly to the company's effort in cutting down on manual testing and extending testing tools to cover several third party devices.

Keywords	Test automation, Android development, JetPack, Kotlin

Contents

List of Abbreviations

1 Introduction 1

2 Software testing 2

2.1 Manual testing 2

2.2 Automated testing 2

2.3 Benefits and risks when moving from manual to automated testing 2

3 Development environment and prerequisites 4

3.1 Android development with Kotlin 4

3.2 Android Jetpack libraries 5

3.3 Overview of Contour Next One 5

4 Biometertest application 7

4.1 Application overview 7

4.1.1 Usage 7

4.1.2 Functionalities 9

4.2 Application architecture 14

4.2.1 MVVM architecture 15

4.2.1.1 Model 17

4.2.1.1.1 Room persistence library 17

4.2.1.1.2 Dependency injection with Koin 21

4.2.1.2 View 23

4.2.1.2.1 Single activity application 23

4.2.1.2.2 UI implementation with live data observation 25

4.2.1.2.3 Data binding 27

4.2.1.3 View Model 30

4.2.1.3.1 BaseViewModel 30

4.2.1.3.2 Launching tests with coroutines 31

4.2.2 Common library for testing abstraction 33

4.2.3 Contour Next driver as a library 34

4.2.4 Other external dependencies 37

5 Application practicalities 40

5.1 Comparison to other test automation tools 40

5.2 First step in increasing test automation coverage 41

5.3 Future improvement and possible features 42

6 Conclusion 44

References 45

List of Abbreviations

MVVM	Model-View-ViewModel. Android design pattern that separate components into three specified categories: graphic user interface, business logic, and value converter for exposing data objects from the model
ErrorCode	An enumeration that holds all the supported error codes that could be returned from the driver
tcbdErrNone	Error code that informs there is no error returned from the driver
tcbdErrLibAlreadyOpen	Error code that informs there is an already open session
POJO	Plain old Java object
Dao	Database access object
HTML	HyperText markup language
DOM	Document object model

1 Introduction

In software development, quality control has always been a top priority for technology corporates when delivering products to customers. In this process, software applications have to be tested against malfunctions and “bugs” to guarantee the best standard before releasing in production environment and real usage. Therefore, testing would be a heavily invested budget in software development and even takes up to 50 percent or more of the whole project cost [1, p. 85]. Traditionally, testers have relied on manual testing and usually spend much time and effort to complete the work. However, due to the rapid growth of international competition, tight delivery time as well as the pressure to optimize cost efficiency in software corporates, test automation has been introduced as an answer to all of the above demands. It has removed the simple and basic monotonous tasks from software testers and accelerated product’s time-to-market [2, p. 494].

Android is an open-source operating system developed by Google and mainly used for mobile phones and tablets, ... Currently, Android has been the most favoured mobile platform compared to others such as iOS, Windows, or Blackberry, ... and leads the OS market with over 70% share in July 2020 [3]. In 2018, Google announced the release of Android Jetpack, which is a suite of useful libraries to provide developers better coding convention, boilerplates reduction, and backward compatibility, and this “next generation of Android APIs” has greatly contributed to speeding up the development of mobile applications [4].

This thesis mainly focuses on explaining the Biometertest, which is an Android application built in Kotlin programming language with the purpose of reducing the testing time for Contour biometer driver and making the driver development faster and easier. Using the latest Android Jetpack libraries, the application lays the initial foundation of increasing test automation coverage for drivers through testing the biometer interface methods. Furthermore, it provides a useful tool for debugging and developing the Contour biometer driver since this driver is added to the Biometertest application as an external library. This application is a project of Signant Health company and plays an important role in the process of switching from manual testing to automated testing and enhancing validation for third party devices.

2 Software testing

Software testing is an activity to guarantee there is no error or defect in application code. Instead of proving that the software functions correctly, testing activities should show the opposite intention, finding errors [5, p. 10-11.] Choosing well-planned test cases and correct testing approach in the beginning phase of development cycle could help to avoid possible future flaws as well as save human effort in debugging and fixing these defects. There are currently two main approaches for software testing, manual and automated. While manual testing relies on the experience of testers to tackle the error-prone part of an application, automated testing can execute a large number of test cases in a short amount of time and requires less user interaction. [6, p. 1.]

2.1 Manual testing

Manual testing involves test execution done physically by a quality assurance specialist in order to discover software defects or feature problems before releasing products to market. Usually, it requires an experienced tester to proceed with all the test cases and produce test results without any help from automation tools.

2.2 Automated testing

Automated testing uses code, test scripts, and tools, which create test cases with expected results, to eliminate the boredom of the manual process. Taken execution time into consideration, automated tests always take a shorter amount of time and more efficient than manual tests.

2.3 Benefits and risks when moving from manual to automated testing

Although automated testing seems to outweigh manual testing in testing efficiency and companies usually want to increase more test automation coverage, pros and cons need to be considered when replacing manual with automated testing.

Pros:

- Automated testing is generated and executed with computing power rather than human force (testers), thus makes more testing in less time.

- Automation requires no human interaction, so it greatly reduces inaccuracy and inefficiency factors from testers

- Automated testing allows scaling, repetition in test run and ensures better test coverage.

Cons:

- With manual work, testing can discover extensive and exception cases where automated tools cannot. When handling situations with complex and in-depth test cases, manual testing would be a preferable choice.

- There is a big cost in implementing, maintaining, and training for test automation process. Testers, especially manual ones, have to switch from their daily working practices to familiarize themselves with automated system. [7, p. 123.]

Nowadays, manual testing discovers a large percentage of bugs and defects, and test automation should be seen as a tool to clear up more time for sophisticated manual testing, not replace it completely [2, p. 494]. When considering moving forward with test automation, companies should first analyze if their testing process is ready to be automated yet, what tools should be used, and how much coverage it should be. Therefore, automated and manual testing should be used in harmony in order to bring the most benefits and positive effects.

3 Development environment and prerequisites

3.1 Android development with Kotlin

At Google I/O 2017, Kotlin was announced to be the official programming language for Android. Since then, Kotlin has demonstrated to be superior to the previous language: Java when compared in many aspects such as code boilerplate reduction, null reference, lambda expressions, inline and extension functions support, singleton and companion object, ... [8]. These advantages have been the reason for Kotlin to become more and more interesting towards developers. Figure 1 below showcases the result of a recent survey conducted by Stackoverflow in 2020.

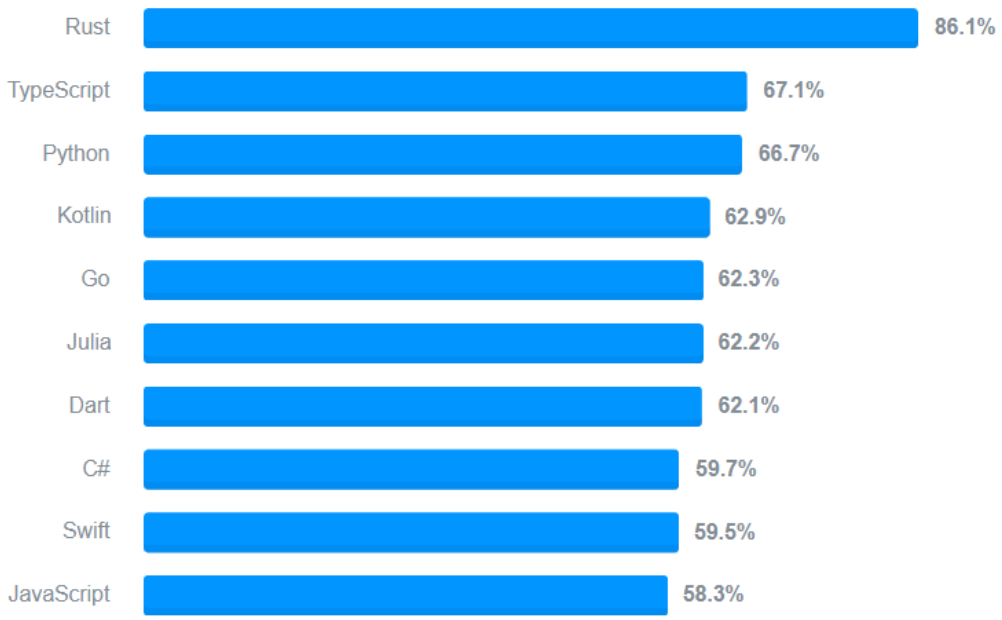


Figure 1 Percentage of developers who are using the programming language and express their willingness to continue with it. Copied from Stackoverflow (2020) [9].

Figure 1 above has shown Kotlin's popularity via over 60% of the surveyed developers who are using the programming language express their enthusiasm for keeping utilizing it. Nowadays, Kotlin is a vital and indispensable component of Android application development.

3.2 Android Jetpack libraries

Developed based on Support Library, Android Jetpack has gained its attraction since the official release in 2018. With the goal to make Android application development more reliable, faster, and simpler, Android Jetpack is a combination of useful instruments, supportive libraries, and architectural recommendations [10], which are categorized into four main groups depicted in the following figure 2.

Figure 2 Android Jetpack categories. Copied from Google Android Developers Blog (2020) [11].

Figure 2 showcases four main components of Android Jetpack: Architecture, UI, Foundation, and Behavior. Each of the components is an ”unbundled” package and can be utilized separately without causing any backward incompatibility. [11.]

3.3 Overview of Contour Next One

The Contour Next device is mainly used for controlling the blood glucose level in patients who are treated with or without insulin for diabetes symptoms. Persons with diabetes themselves and healthcare personnel can measure the glucose level by extracting a whole blood drop with the test strip (from the patient’s fingertip or palm) and the test is for single-patient use at most. [12, p. i.]. Below figure 3 demonstrates how a Contour Next One meter looks like.

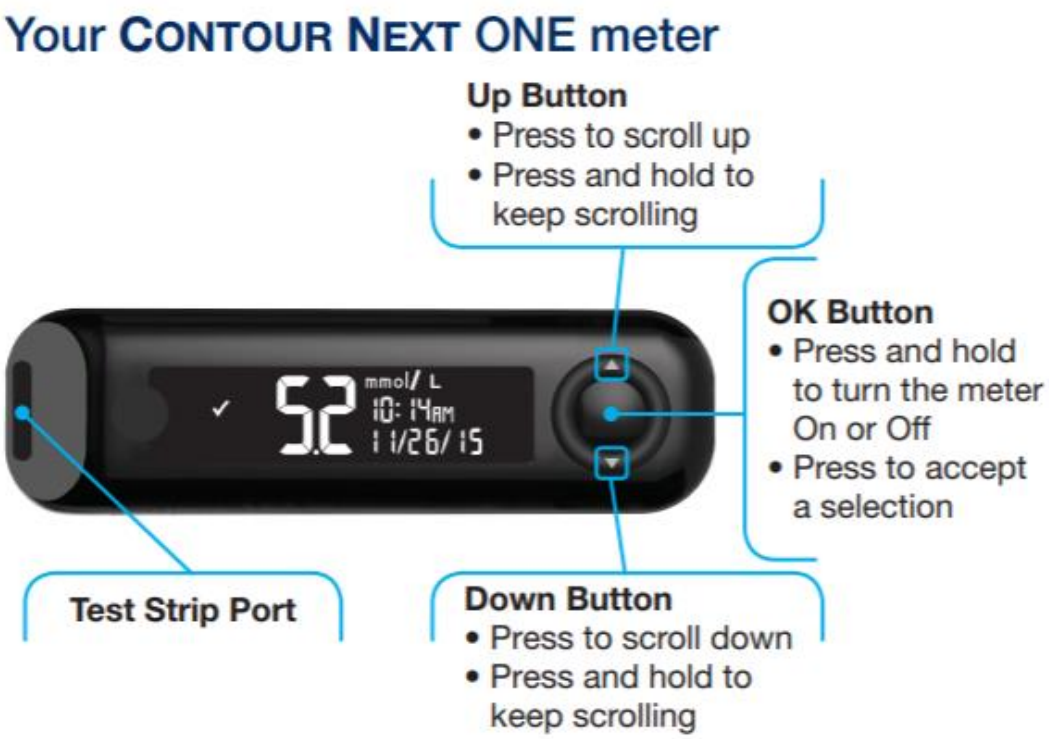


Figure 3 Contour Next One Meter overview. Copy from Contour®Next One User Guide (2020) [12, p. 3].

Figure 3 shows a brief user guide for Contour Next One device. With the physical Contour meter, any mobile device (Biometertest application in this case) can be paired via Bluetooth connection. By pressing the OK button and holding, the flashing blue light at the end of the meter (test strip port) comes up, blinks a couple of times, after that the meter is in pairing mode and ready for any Bluetooth connectivity [12, p. 46]. However, two dependencies have to be taken into consideration before investigating more how Biometertest application interacts with the Contour meter and delivers test automation practices. The first dependency is TrialCollector, which is a Signant Health application to provide diaries for collecting data from patient clinical trials. TrialCollector will interact with the Contour meter through the second dependency, which is the Contour Next One driver. This driver is an implementation of low-level Bluetooth interaction with the Contour Next One device and provisions functioning interfaces for TrialCollector to communicate with the meter. [13, p. 6.] These interfaces, which are extended by the Biometer interface library including methods and utility objects that can communicate with biometers [14, p. 5], have to be ensured that they will operate properly and correctly and that leads to the need for automating testing practice to help the company's specialists speed up the

validation process. Therefore, the Biometertest application is developed as a useful tool to solve this need.

4 Biometertest application

4.1 Application overview

4.1.1 Usage

To start using the Biometertest application, there are two requirements needed to be fulfilled. First, the application .apk file has to be installed properly on an Android device (iOS is not supported at the moment). The .apk file can be found via Signant Health's public drive at the address O:\Public\HEL\harrison\Biometertest\apk. After the normal installation, when the application is opened, device location and Bluetooth connection permission have to be granted so that Biometertest can start pairing with the Contour meter [15]. Figures 4 and 5 below depict how location and Bluetooth rights are enabled from the Biometertest application.

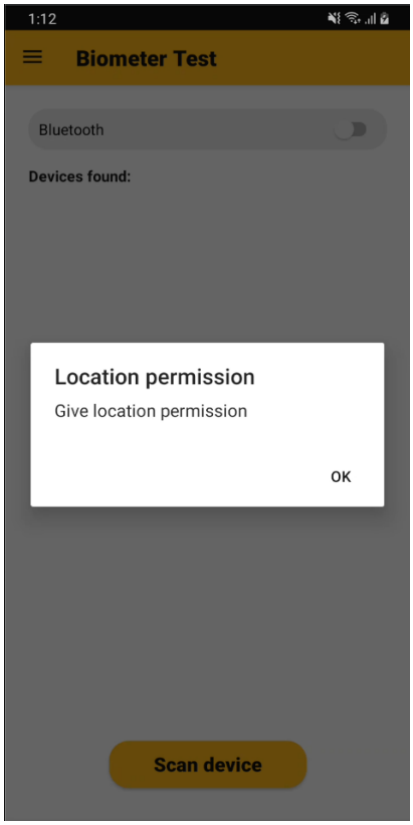


Figure 4 Device location popup.

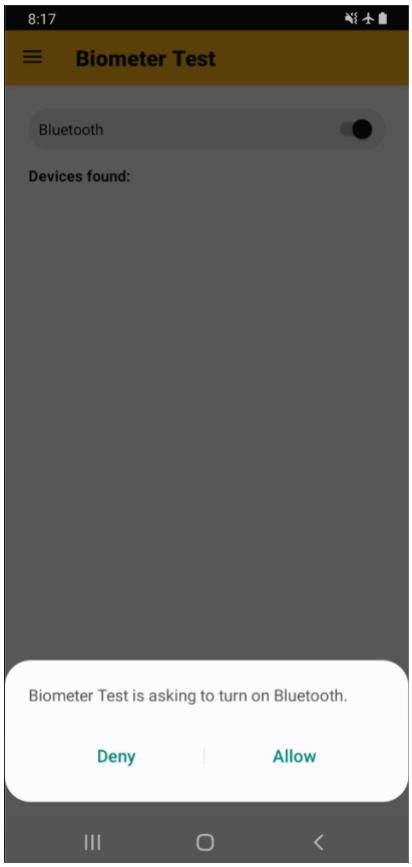


Figure 5 Device Bluetooth connection popup.

According to figures 4 and 5, device location can be turned on by selecting “OK” in the location popup and Bluetooth connection can be enabled via the Bluetooth toggle at the top of the screen. After completing appropriate setup for the application, the second requirement would be establishing the connection between a Contour Next One meter and current device. The same process must be followed as section 2.3.1 above, including pressing the OK button of the Contour meter until there is a constant blue light and afterward, tap the “Scan device” button from the Biometertest scanning screen to start pairing with the available meter. An example of the scanning screen is illustrated in figure 6:

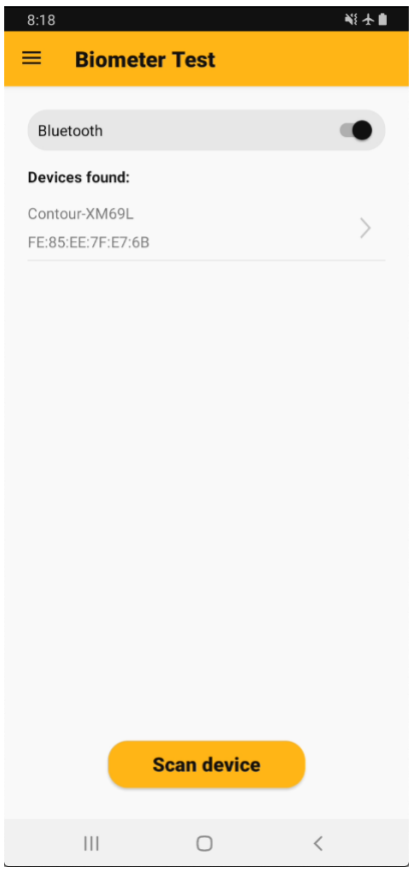


Figure 6 Scanning screen.

As shown in figure 6, if a connection between Biometertest application and Contour device is established, there should be a list of Contour meter items displayed with biometer's name and series number (Contour-XM69L and 830-348-8100 in this case). Remember to check this information carefully at the back of the Contour meter and then select the correct Contour device among the list items. After this point, a user is ready to use the application to perform testing practices.

4.1.2 Functionalities

With a Contour Next One meter pairing correctly with a properly setup Biometertest, testing practices can be carried out by users via many application features. First of all, the application is able to present a list of testing categories and from that list, user can select all or certain groups of categories to perform testing activities. To be easier to comprehend, feature 7 below will demonstrate how this functionality looks like.

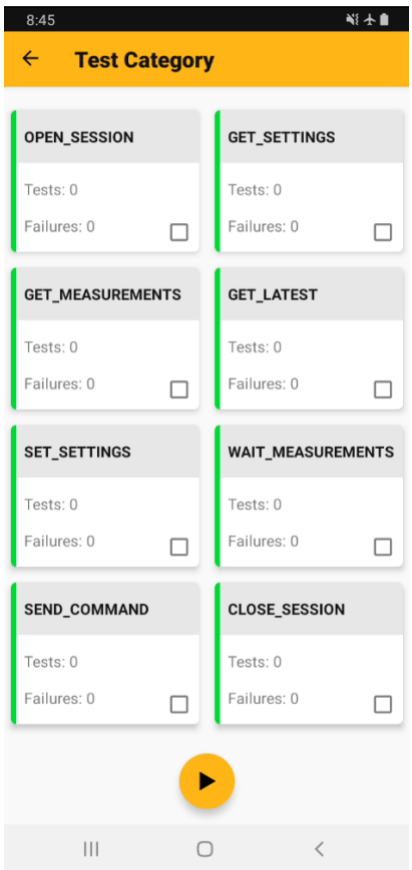


Figure 7 Category screen.

In figure 7, a list of categories is displayed with “OPEN_SESSION”, “GET_SETTINGS”, “GET_MEASUREMENTS”, … items. These categories represent every biometer interface method of IBiometerDriver class, which need to be tested when these methods interact with a meter device (Contour Next One in this case). User can then select different categories (or all of them) by tapping the checkbox at the right end of every category item and run the tests by pressing the run button at the bottom of the screen. After all tests finish running, there will be a brief summary for each of the selected categories, stating how many tests have been performed and how many tests have failed for each of the category. Moreover, there is a color bar indicator at the left hand-side of the category item to determine if that category has passed all the tests (green bar) or failed (red bar). This process is illustrated by figure 8 below.

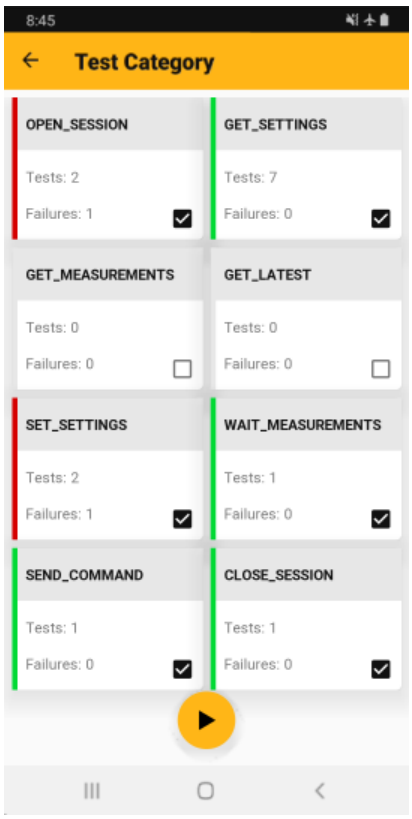


Figure 8 Category screen with performed tests.

Figure 8 demonstrates a testing result screen with OPEN_SESSION category fails (2 tests run and 1 failure), GET_SETTINGS category passes (all 7 tests), SET_SETTINGS category fails (2 tests run and 1 failure), WAIT_MEASUREMENTS category passes (1 test), SEND_COMMAND category passes (1 test) and CLOSE_SESSION category passes (1 test). The other categories (GET_MEASUREMENTS and GET_LATEST) were not selected to be executed.

Secondly, Biometertest can display a list of performed tests based on a specific selected category and show testing results – a summary of each test (name, fail/pass, test description). Figure 9 below demonstrates a list of tests that have been executed for OPEN_SESSION category.

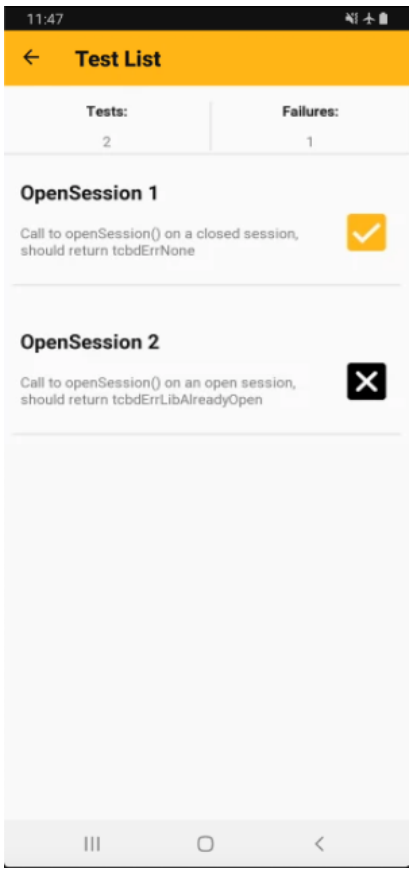


Figure 9 List of OpenSession tests.

After selecting the first executed category in figure 8 above, a list of test cases for that category is shown in figure 9. To be more specific, OPEN_SESSION category consists of two test cases: “OpenSession 1” and “OpenSession 2”. The former passed and has “Call to openSession() on a closed session, should return tcbdErrNone” as a description of what the test is used for, and the latter failed with slightly different explanation “Call to openSession() on an open session, should return tcbdErrNone”.

Thirdly, each test after being performed can be read with more detailed information, such as test name, test status, test execution time, test logs with meter device’s name (Contour device currently supported), driver’s interface method, and error message. This functionality is demonstrated in figure 10 below.

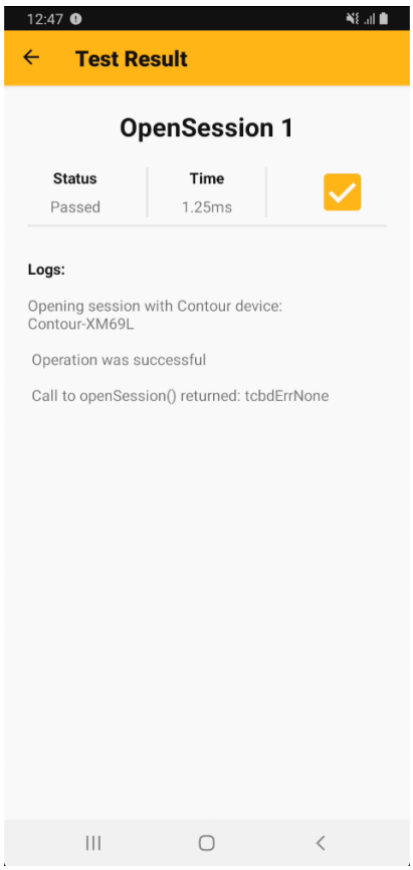


Figure 10 OpenSession 1 detailed screen.

When a test case of the test list (figure 9) is selected, a detailed screen of that test will be displayed and in figure 9, “OpenSession 1” detail screen is shown as an example. Detailed information can be analyzed such as test name “OpenSession 1”, test status “Passed”, test execution time “1.25 ms”, and test logs such as Contour-XM69L was the name of the device the test run upon, “openSession()” function was tested and there was no error returned (tcbdErrNone). This information will be greatly helpful for determining if a driver interface method (openSession() in this case) interacts correctly with the Contour meter device so that TrialCollector can fully operate without any defect.

Lastly, after running all of the needed tests, Biometertest can automatically generate a test report that summarizes every testing attempt, which helps to keep track of the testing activities and provide enough evidence for spotting testing defects. This useful feature will be explained later with more details in the section “Other external dependencies”.

4.2 Application architecture

Biometertest application follows MVVM Architectural Pattern for Android application. Besides, countournextdriver and commonlibrary are also added to the project structure as Android library modules with the purpose of abstracting biometer dependencies. Figure 11 below describes an overview of the whole project structure.

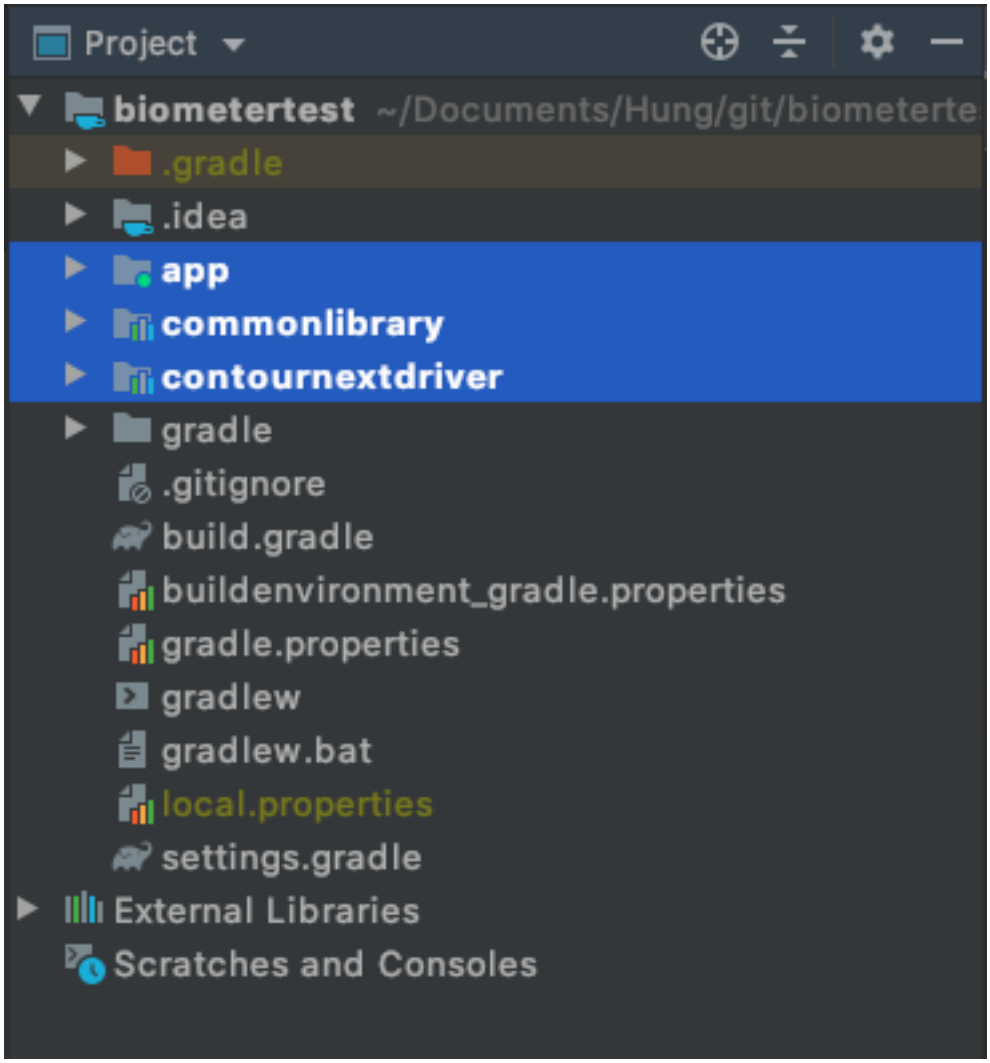


Figure 11 Application structure with two dependency modules.

To add commonlibrary and contournextdriver as separate modules in Android Studio as in figure 11, the steps below should be followed:

- Go to File -> Project Structure…

- Select “Dependencies” tab -> New module (“+” button)

- Select “Android library” -> Fill in Module name -> Click “Finish”

After that in settings.gradle file, there will be two newly-created subprojects. The content of setting.gradle file is described in listing 1 below.

```
include ':app', ':contournextdriver', ':commonlibrary'
```

Listing 1. Setting.gradle file with two subprojects commonlibrary and contournextdriver.

However, to be able to use these new subprojects, app/build.gradle file has to be updated as well. Listing 2 contains two lines of code that have to be added.

```
implementation project(path: ':contournextdriver')
implementation project(path: ':commonlibrary')
```

Listing 2. App/build.gradle implements two new subprojects.

4.2.1 MVVM architecture

MVVM (Model-View-ViewModel) is an architectural pattern that origins from MVC (Model/View/Controller) pattern and consists of three loosely coupling components: Model, View, and ViewModel [16]. This architecture is represented in figure 12 below.

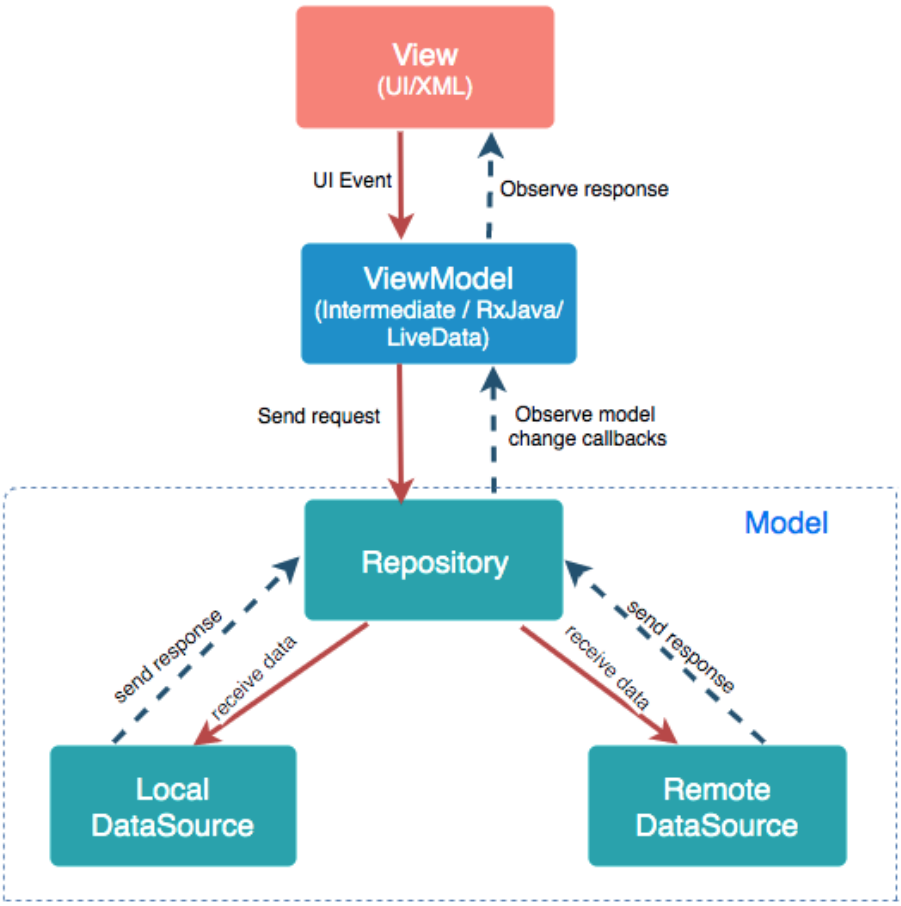


Figure 12 Model-View-ViewModel architecture. Copied from Vandana Srivastava (2021) [14].

According to figure 12, View component is used for displaying the UI part of an application and should be targeted for a general designer rather than an actual developer. UI codes such as Activities, screen Fragments, XML layout files would be good representatives for a View component. In addition, a View component will send user's interaction to the ViewModel but the response will be received through observable variables exposed by the ViewModel using data binding framework.

Then, the Model component will be in charge of the application's data or business logic. This component fetches and updates data, which is retrieved from either a local data source such as Shared Preferences, and Room database or a remote data source such as Firebase and REST API.

Last but not least, ViewModel connects the View with ViewModel. It exposes the observables and sends notifications to any Views that subscribes to these observables. Furthermore, it can emit events to Model component and acts as an intermediary to convert raw ugly model data to user-friendly data in View component.

4.2.1.1 Model

Model is an initial foundation for implementing MVVM architecture. It will handle data from multiple sources, such as shared preferences, local database (Room, SQLiteDatabase), or remote APIs. Most of the time, Model component will expose its data for ViewModel using Observable pattern. In this project, Model component utilizes Room persistent database and uses it as the main approach to save testing results and generate summary reports.

4.2.1.1.1 Room persistence library

Basically, Room library is a higher-level abstraction of SQLite and it allows applications to store offline data in local memory. Because of being a layer on top of SQLite database, Room handles the validation of SQL inquiries at compile-time, gets rid of lengthy boilerplate practice to convert between Java database objects and SQL queries, takes care of database migration, and also supports Live data and RxJava for updating UI automatically when there is data change. [18.]

To start using Room persistent database, dependency adding must be done in application build.gradle file (listing 3 below).

```
dependencies {
   //Room database
   implementation "androidx.room:room-runtime:$roomVersion"
   implementation "androidx.room:room-ktx:$roomVersion"
   kapt "androidx.room:room-compiler:$roomVersion"
}
```

Listing 3 Room dependency in build.gradle file.

In Room, there are three main elements: Database, Entities, and Dao objects. They constitute a Room database and provide the ability to perform operations to that local

database. Regarding an entity, it is basically a POJO (Plain old Java object) which stores database information and needs to be marked with the @Entity annotation so that Room can transform this object into a database table. Because each of the fields of an entity class matches with a column in the database table, entity classes should be small and do not hold any logic [19]. For Biometertest application, there are two entities declared in the common library module, namely Category and TestResult. Listing 4 demonstrates these two entities below.

```
@Entity
data class Category(
    val name: String,
    val tests: Int,
    val failures: Int
) {
    @PrimaryKey(autoGenerate = true)
    var id: Int = 0
}
@Entity(foreignKeys = [ForeignKey(
    entity = Category::class,
    parentColumns = ["id"],
    childColumns = ["categoryId"]
)])
data class TestResult(
    @PrimaryKey
    val testName: String,
    val pass: Boolean,
    val errorCode: String,
    val description: String,
    val logs: String,
    val categoryId: Int,
    val time: Int
)
```

Listing 4 Common entities of Biometertest application – entities.kt file.

As can be seen from listing 4, there are two main database tables in BIometertest database, which are Category and TestResult. Category table will store the current data of the testing categories that need to be run, the number of tests and failures for each category. Regarding TestResult table, all test result entries will be saved with a test

name, a test status, an errorCode. These two tables are common core entities of the whole application which aims to support different kinds of biometer devices in the future such as Contour, Dexcom, TaiDoc, and MyGlucoHealth. Therefore, entities.kt file that contains two entities mentioned above is extracted into commonlibrary module of Biometertest code base.

Dao (database access object) objects are in charge of declaring operating functions to interact with the database. Each DAO is an abstract class with @Dao annotation and consists of one or more database methods such as @Query, @Insert, or @Delete [20.] Listing 5 below is an example of CategoryDao class.

```
@Dao
interface CategoryDao {

    /**
     * Insert a list of categories to the database
     * @param categories: categories to be inserted
     * @return a list of long which is the category's id
     */
    @Insert(onConflict = OnConflictStrategy.ABORT)
    suspend fun insert(categories: List<Category>): List<Long>
    . . .

    /**
     * Return all categories
     * @return a list of categories
     */
    @Query("SELECT * FROM Category")
    fun getAllCategories(): LiveData<List<Category>>
    . . .
}
```

Listing 5 CategoryDao.kt.

In listing 5, CategoryDao has a method which inserts a list of categories returns a list of category ids. Because database access operations are long-lasting tasks and Android does not allow them to be run in the main thread, this insert method also has a *suspend* prefix so that it can be executed in a separate thread with Coroutines. Coroutine is a useful pattern to handle concurrency tasks and will be explained in more details in the

ViewModel section below. Back to the main point, if *getAllCategories* method is taken into consideration, it can be noticed that this function returns a live data of category list. Using live data here helps to respond to any changes in the Category database and update in time for the ViewModel. Furthermore, when combining Livedata with Room, boilerplate code for invoking the method is reduced, especially when using Coroutine to handle the database interaction. This can be shown in listing 6 below.

```
// Not using live data. Must use Coroutine to handle database
// interaction
val categoriesLiveData = MutableLiveData<List<Category>>()
fun fetchCategories() {
    launch {
        categoriesLiveData.value = categoryDao.getAllCategories()
    }
}

// Using live data, avoid using Coroutine
val categoriesLiveData = categoryDao.getAllCategories()
```

Listing 6 Comparison between with and without using Livedata in Room Dao.

Last component of Room persistent library is the database class which owns the application database and acts as a starting endpoint for accessing operation to persistent app's data [18]. To instantiate a database, database class must be annotated with @Database and consists of abstract methods that define instances of Dao classes. Listing 7 depicts how TestResultDatabase class is declared in Biometertest application.

```
@Database(entities = [TestResult::class, Category::class], version= 1)
abstract class TestResultDatabase: RoomDatabase() {
    abstract fun testResultDao(): TestResultDao
    abstract fun categoryDao(): CategoryDao
    companion object {
        @Volatile private var instance: TestResultDatabase? = null
        . . .
        private fun buildDatabase(context: Context) =
        Room.databaseBuilder(
           . . .
        ).addCallback(object: Callback() {
            override fun onCreate(db: SupportSQLiteDatabase) {
```

```
                super.onCreate(db)
                ioThread {
                    val inputStream = context.assets.
                    open("database/prepopulatedata.txt")
                    val query = inputStream.bufferedReader()
                    .use { it.readText() }
                    db.execSQL(query)
                }
            }
        }).build()
    }
}
```

Listing 7 TestResultDatabase.kt.

In listing 7, TestResultDatabase is instantiated through Room's *databaseBuilder* function, besides, prepopulated entries are fetched from database/prepopulatedata.txt file and inserted into a database instance. Therefore, Category table will have eight Categories with zero test and failure value in the beginning when Biometertest database is originated. Content of prepopulatedata.txt file is shown in listing 8 below.

```
INSERT INTO `Category` (`name`, `tests`, `failures`)
VALUES ('OPEN_SESSION', 0, 0),
       ('GET_SETTINGS', 0, 0),
       ('GET_MEASUREMENTS', 0, 0),
       ('GET_LATEST', 0, 0),
       ('SET_SETTINGS', 0, 0),
       ('WAIT_MEASUREMENTS', 0, 0),
       ('SEND_COMMAND', 0, 0),
       ('CLOSE_SESSION', 0, 0)
```

Listing 8 Prepopulatedata.txt.

4.2.1.1.2 Dependency injection with Koin

In Biometertest, there are many cases when CategoryDao and TestResultDao are invoked to interact with the application database. This leads to the need of reducing unnecessary dependencies between a ViewModel class and database abstract class.

Therefore, Koin dependency injection package is taken into use for making these classes less vulnerable to other component dependency changes. To start using this library, Koin must be declared in build.gradle (listing 9).

```
dependencies {
   //Koin AndroidX ViewModel features
   implementation "org.koin:koin-androidx-viewmodel:$koinVersion"
}
```

Listing 9 Koin dependency in build.gradle.

Next, in topmost App.kt file, CategoryDao and TestResultDao are declared as singleton in Koin appModule (listing 10).

```
val appModule = module {
    single {
        TestResultDatabase(get()).categoryDao()
    }
    single {
        TestResultDatabase(get()).testResultDao()
    }
}
class App: Application() {
    override fun onCreate() {
        super.onCreate()
        startKoin {
            androidContext(this@App)
            modules(appModule)
        }
    }
}
```

Listing 10 Module declaration in Koin.

As can be seen in listing 10, App class starts a Koin module call *appModule* which contains two Dao singletons: CategoryDao and TestResultDao. After this step, whenever there is a need to call a Dao to perform database operation, it can be done easily as listing 11 below.

```
class CategoryViewModel: BaseViewModel() {
    private val categoryDao: CategoryDao by inject()
    private val testResultDao: TestResultDao by inject()
}
```

Listing 11 Inject Dao dependencies in CategoryViewModel.kt.

With Koin library, component dependencies have been greatly decreased and therefore, class maintainability also increased due to loosely coupling effect. Furthermore, this feature not only improves the testability of code base but also enhances the readability of simple, separate classes. In conclusion, Koin has been a great support for Biometertest development.

4.2.1.2 View

4.2.1.2.1 Single activity application

Developers used to create multi-screen applications by using intent or fragment transactions and passing data through intent argument from screen to screen. However, this lengthy coding process has been improved by Android Jetpack Navigation. This newly introduced component tackles the complexity of moving from one screen to another and also handles sophisticated patterns such as “Bottom Tab Navigation” or “App Drawer” [21]. Moreover, when using Navigation Component, the application only has one activity with multiple fragments, and this is perfectly visualized through the Navigation graph resource file (figure 13).

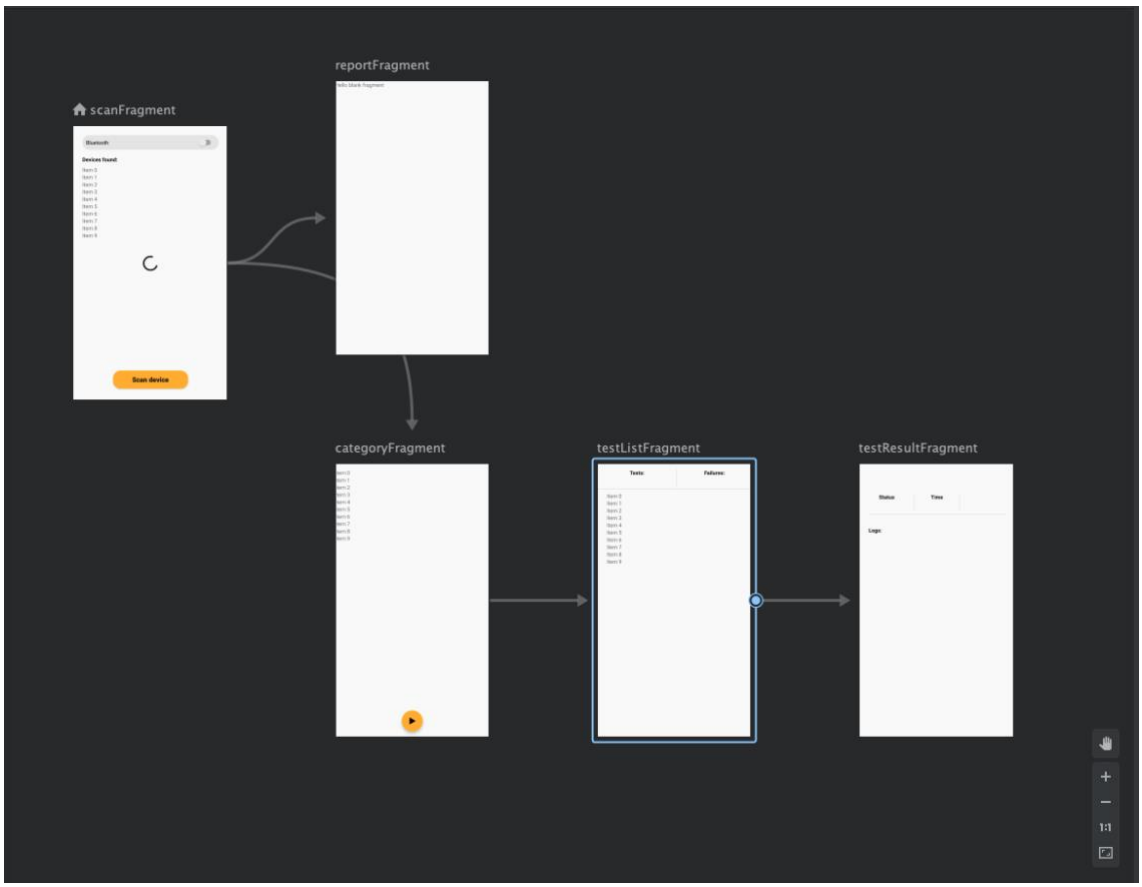


Figure 13 Application navigation: navigation.xml.

As can be seen from figure 13, a brief overview of the application layout is described in the navigation.xml file. Each of the screens represents one fragment and is linked together by arrows (or actions) to navigate to other screens, which are also called *destinations*. Every action contains embedded information of the origin screen‘s and destination screen’s id, arguments passing between these two, animation behaviour, and backstack behaviour. In addition, to create a complete Navigation component, apart from the Navigation graph, NavHost fragment and NavController are also introduced to fully control the behaviour of the application’s navigation. NavHost is basically a fragment and serves as a placeholder for switching screens declared in Navigation graph. Usually, NavHost fragment is included inside activity_main.xml layout, which is the topmost starting point of the application layout (see listing 12).

```
<fragment
        android:id="@+id/fragment"
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
```

```
        app:defaultNavHost="true"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:navGraph="@navigation/navigation" />
```

Listing 12. Application's NavHostFragment inside activity_main.xml.

Listing 12 is a description of a NavHost fragment inside activity_main.xml resource file. It has a property called navGraph which is pointing at the navigation.xml graph (figure 14). Lastly, to actually control the flow of navigation, NavController object needs to be used with an action parameter which specifies the correct navigation between two destinations. Along the way, NavController can also pass on an argument so that it can be used in the destination screen. Listing 13 showcases navigation between Category fragment to Tests fragment accompanied with a categoryId argument.

```
override fun onCategoryClick(view: View, categoryId: Int) {
    if(isClickable) {
        val action = CategoryFragmentDirections.actionCategoryFrag.
mentToTestsFragment()
        action.categoryId = categoryId
        Navigation.findNavController(view).navigate(action)
    }
}
```

Listing 13. Application's NavHostFragment inside activity_main.xml.

With three main elements: Navigation graph, NavHost fragment, and NavController, Android Navigation component not only provides developers a useful way of control the application's flow but also acts as a beacon to follow the MVVM architecture pattern.

4.2.1.2.2 UI implementation with live data observation

To instantiate a screen view, each XML resource file has to be inflated through a screen fragment. Taken fragment_category.xml into consideration, there is a corresponding CategoryFragment.kt file which is in charge of inflating this UI screen. The inflation mechanism can be seen from listing 14.

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    // Inflate the layout for this fragment
    return inflater.inflate(R.layout.fragment_category, container,
false)
}
```

Listing 14. Inflating fragment_category.xml in CategoryFragment.kt.

Noticeable from listing 14, the inflation has to be done inside an override *onCreateView* function, which returns a user interface View, and this function takes three default parameters (*inflater*, *container*, and *savedInstanceState*) but the *inflater* parameter would be mainly used for inflating the screen view.

However, the most important pattern a fragment implements is observing ViewModel live data. As already mentioned in MVVM section above, ViewModel class will provide data for UI views and expose observables to UI controllers, hence acts as a bridge between View and Model components [22]. In this section, observing mechanism would be the main target to be discussed.

LiveData is a class that contains application data, and this class can be observed (or subscribed) based on a specific lifecycle. When observing a live data object, a *LifeCycleOwner* must be included to specify if this owner is in an active state or not (either STARTED or RESUMED state). [23.] To demonstrate this pattern, in *onViewCreated* function of CategoryFragment.kt class, an *observeViewModel* method is introduced to perform the live data subscription (listing 15).

```
private fun observeViewModel() {
    viewModel.categoriesLiveData
      .observe(viewLifecycleOwner, Observer {
         selectedTestCheckBoxes.forEach { checkbox ->
         checkbox.isChecked = false
         }
         selectedTestCheckBoxes.clear()
         categoryListAdapter.update(it)
```

```
    })
    viewModel.testResultsLiveData
      .observe(viewLifecycleOwner, Observer {
        viewModel.createReport(requireContext(), it.toMutableList())
    })
}
```

Listing 15 Observing live data objects in CategoryFragment.kt.

In listing 15, *observeViewModel* function has two live data objects (*categoriesLiveData* and *testResultsLiveData*) being subscribed to the *viewLifecycleOwner* of the current fragment. This owner object helps specify which state of the current fragment lifecycle is, thus, remove the observer if the aforementioned lifecycle changes to DESTROYED state (not in active state), and eliminate a scenario when a live data object is observed twice. Furthermore, if there is any change to the live data value, these two live data objects will notify the observer and update the UI correspondingly. Regarding *categoriesLiveData*, any changes in this live data object would result in *selectedTestCheckBoxes* being unchecked and cleared, and *categoryListAdapter* would be updated. For *testResultsLiveData,* live data value change triggers the report generation process.

4.2.1.2.3 Data binding

Data binding is an Android library that allows developers to attach information or data to application's layouts. It also provides the ability to remove from data configuration in app's functional classes to resource layouts. Therefore, one or more variables can be accessed and updated directly in UI elements. [24.]

To use data binding in an Android application, first data binding feature has to be enabled. Listing 16 is what should be configured in build.gradle file.

```
android {
   ...
   dataBinding {
       enabled = true
   }
}
```

Listing 16 Data binding configuration.

To put data binding into use, data binding variables are declared inside *data* tag which is a child of UI layout parent component [24]. Listing 16 below demonstrates how this can be done in category_item.xml file.

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto">
    <data>
        <variable
                name="category"
                type="com.signanthealth.commonlibrary.Category" />
        <variable
                name="categoryId"
                type="Integer" />
        <variable
                name="listener"
                type="com.signanthealth.biometertest.view
                      .listener.CategoryClickListener" />
        <variable
            name="checkboxListener"
            type="com.signanthealth.biometertest.view.listener
                  .CheckboxCheckListener" />
    </data>
                      . . .
</layout>
```

Listing 17 Initialize data binding variables in category_item.xml.

In listing 17, four variables - *category*, *categoryId*, *listener*, and *checkboxListener* - are declared for data binding. Each variable must have a specific type, for example: integer, bool, data class, or even interface class. Afterward, the variable can be used directly in UI component, as in listing 18.

```
<TextView
        android:id="@+id/categoryTitle"
        style="@style/MyTitle"
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@{category.name}"
        . . . />
```

Listing 18 Using category binding variable in category_item.xml layout.

Since Category screen contains a RecyclerView UI component, a CategoryListAdapter class (extending RecyclerView.Adapter class) is associated with this UI component and inflates each of the category items in the list whose layout is defined in category_item.xml resource file. The list item inflation is done in *onCreateViewHolder* override method (listing 19).

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
CategoryViewHolder {
    val inflater = LayoutInflater.from(parent.context)
    val view = DataBindingUtil.inflate<CategoryItemBinding>(inflater,
               R.layout.category_item, parent, false)
    return CategoryViewHolder(view)
}
```

Listing 19 Inflate category_item.xml inside CategoryListAdapter.kt.

According to listing 19, *DataBindingUtil* will inflate the related UI layout and initialize the binding. Then, it comes to the most important and final step: bind the declared above variables (see listing 20).

```
override fun onBindViewHolder(holder: CategoryViewHolder, position:
Int) {
    holder.view.category = categories[position]
    holder.view.categoryId = position + 1
    holder.view.listener = this
    holder.view.checkboxListener = onCheckboxCheckListener
}
```

Listing 20 Binding declared variables in CategoryListAdapter,kt.

All four variables declared in category_item.xml are bound here in *onBindViewHolder* override method. Each is assigned to an initial value and this value would be used to

render the correct UI element in XML layout. This practice has proven to bring many benefits. First, it helps to shorten coding activity, especially reduces dramatically *findViewById* coding style. Moreover, it can synchronize between UI elements and source data, makes views and data to be clearly set apart, and automatically generates useful binding classes. And last but not least, data binding also fits perfectly with MVVM architecture due to the power of informing all the views when there is a data change in observable objects rather than manually update them. Therefore, using data binding is highly recommended for developing Android applications, especially the Biometertest.

4.2.1.3 View Model

ViewModel is a lifecycle-aware component which is in charge of preparing data for UI components. It is attached to a specific fragment (or an activity) and gets destroyed whenever that fragment/activity is completely finished and discarded. [22.] Usually, a ViewModel would expose its data through live data objects or data binding variables and if there is any change coming from those data, the attached fragment (or activity) would be able to observe and, hence, update the corresponding UI. One main point to be remembered is ViewModel can never hold references to any UI controller, Context, or even a View. This mis-implementation is against the goal of isolating UI components from its data and could cause memory leaking.

4.2.1.3.1 BaseViewModel

In Biometertest application, any ViewModel class should extend from the BaseViewModel abstract class, which is shown in listing 21 below.

```
abstract class BaseViewModel: ViewModel(), KoinComponent,
CoroutineScope {
    private val job = Job()

    override val coroutineContext: CoroutineContext
        get() = job + Dispatchers.Main

    override fun onCleared() {
        super.onCleared()
        job.cancel()
```

```
    }
}
```

Listing 21 BaseViewModel.kt.

According to listing 21, *BaseViewModel* is an abstract class which implements *ViewModel* abstract class, *KoinComponent,* and *CoroutineScope* interfaces. *KoinComponent* interface helps any class that extends *BaseViewModel* be able to use Koin dependency injection – both *CategoryDao* and *TestResultDao* mentioned in section 3.2.1.1.2 above. Besides, using *CoroutineScope* requires a declaration for a coroutine job and a coroutine context, which will be explained in more details in the next section below. After the initial definition of an abstract class (*BaseViewModel*) is defined, launching a coroutine for testing activities and Daos injection would be available for *CategoryViewModel*, *TestListViewModel*, and *TestResultViewModel* classes.

4.2.1.3.2 Launching tests with coroutines

In Android development, blocking main thread with long-lasting operating tasks should be avoided to provide users a smooth and frictionless experience. Therefore, Coroutine is introduced as an Android design pattern to handle concurrency, time-consuming background tasks which can block UI main thread and cause responsiveness issues [25]. This section will explain deeply Coroutine mechanism and how this solution is used for running biometer test cases.

Coroutine is developed by JetBrains and can be considered as an instance of a lightweight thread. It is used for handling long-running operations and can be started with a *launch/await* coroutine builder within a CoroutineScope context. A CoroutineScope will take care of any coroutine created inside it and every coroutine builder would be an extension of the CoroutineScope functionality. [26.] Moreover, when specifying a coroutine scope, a dispatcher can be declared in order to know which thread that coroutine is run on, and there are three main types of dispatchers:

- Dispatcher.Main: used for running Coroutine on main UI thread. This dispatcher should be used with fast operating tasks such as updating Live data objects and invoking suspend functions.

- Dispatcher.IO: this dispatcher can handle blocking IO operations by distributing them to a shared number of threads. Use cases are reading/writing files, network interactions.
- Dispatcher.Default: used for dealing with long-lasting CPU work such as JSON parsing or list sorting.

By using Coroutine, many Room database operations are handled smoothly in ViewModel classes and UI components are updated corresponding based on live data objects. With the same implementation as other ViewModel classes, CategoryViewModel is chosen as an example of using coroutines for database operations (listing 22).

```
class CategoryViewModel: BaseViewModel() {
    val categoriesLiveData = categoryDao.getAllCategories()
    val testResultsLiveData= testResultDao.getAllTestResults()
    val runningTests = MutableLiveData<Boolean>()

    fun runTestTasks(context: Context, deviceType: Drivers, blue.
toothDevice: BluetoothDevice, categories: List<TestCategory>) {
        . . .
        testRunner = TestRunnerFactory.createRunner(context,
deviceType, bluetoothDevice)
        launch {
            runningTests.value = true
            val testResults = withContext(Dispatchers.Default)
{testRunner.runTests(categories)}
            val updatedCategories = testResults.groupBy {
it.categoryId }.map { createUpdateCategory(it) }
            testResultDao.insert(testResults)
            categoryDao.update(updatedCategories)
            runningTests.value = false
        }
    }
}
```

Listing 22 Using Coroutine in CategoryViewModel.kt.

In listing 22, CategoryViewModel is extending BaseViewModel class which already configures a CoroutineScope with Dispatcher.Main. Therefore, suspend function such

as *createUpdateCategory, category.update* can be used within a *launch* builder. However, when *testRunner.runTests* function is called, it must be wrapped inside another coroutine with *Dispatcher.Default* type because this is a long-lasting CPU execution and needs to be run in *Default* dispatcher.

4.2.2 Common library for testing abstraction

Commonlibrary is an Android module added to Biometertest application for separating the common logic of biometer testing. By using this module as a common base, every newly added biometer shall apply the same logic and be easily tested. Figure 14 below shows a quick overview of how commonlibrary module looks like.

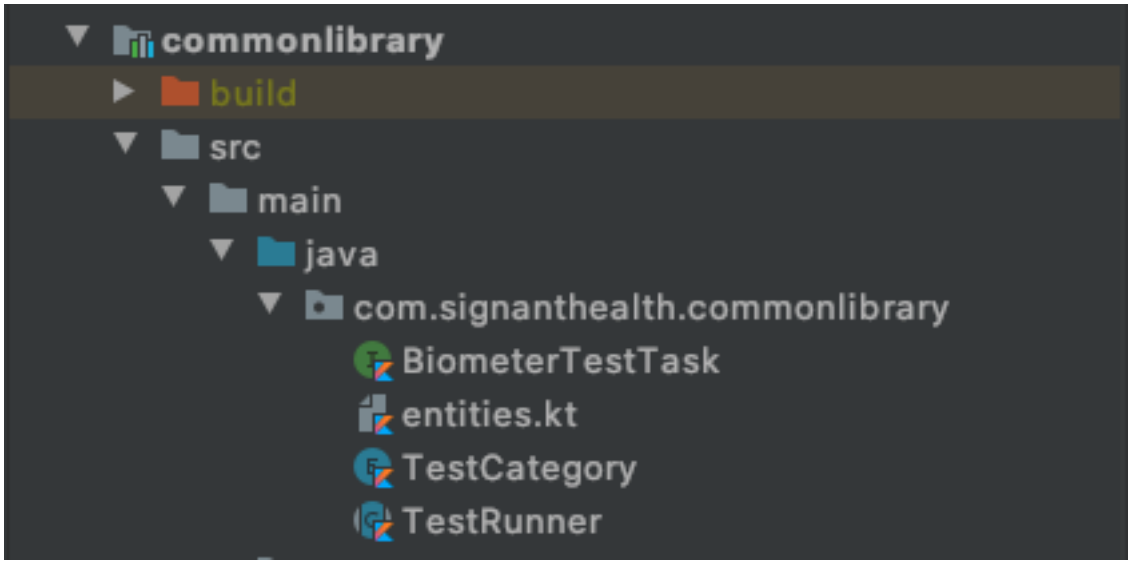


Figure 14 Commonlibrary module.

As can be seen from figure 14, commonlibrary consists of four specific classes, *BiometerTestTask* interface, entities.kt file, *TestCategory* enum class, and *TestRunner* abstract class. For entities.kt file, it contains definitions of Room database entities such as Category, TestResult and this has been discussed previously in Room persistent database section. Regarding BiometerTestTask interface, it defines only one method *runTests* that needs to be overridden by other implemented classes such as *OpenSessionTestTask* or *CloseSessionTestTask*. For TestCategory class, a list of categories is defined in TestCategory enum (listing 23 below).

```
enum class TestCategory {
    OPEN_SESSION,
    GET_SETTINGS,
    GET_MEASUREMENTS,
    GET_LATEST,
    SET_SETTINGS,
    WAIT_MEASUREMENTS,
    SEND_COMMAND,
    CLOSE_SESSION
}
```

Listing 23 TestCategory enum class.

Last but not least, TestRunner abstract class acts as a base class to instantiate test runner for different type of biometers. It defined a late init variable called *biometerDriver*, and two abstract functions named *initBiomterDriver* and *runTests* (see listing 24).

```
abstract class TestRunner(val btDevice: BTDevice) : BTDevice() {
    lateinit var biometerDriver: IBiometerDriver
    abstract fun initBiometerDriver(context: Context)
    abstract fun runTests(categories: List<TestCategory>):
List<TestResult>
}
```

Listing 24 TestRunner.kt abstract class.

4.2.3 Contour Next driver as a library

In Biometertest application, contournextdriver is a dependency module that has only the specific testing logic of the Contour biometer. It contains two main components: the contournextdriver jar and a java resource folder related to Contour testing implementation such as test runner class, biometer test tasks (namely *OpenSessionTestTask*, *GetSettingsTestTask*, *CloseSessionTestTask*). With this module pattern, all specific logic of a certain biometer (Contour in this case) shall be clearly separated and easily maintained. Besides, if there is a demand for expanding to other biometer devices (such as Dexcom, TaiDoc, MyGlucoHealth), the project module structure is definitely an optimal way to scale and develop in the long run.

Contournextddriver jar file is an implementation of the low-level Bluetooth communication between the Contour meter and TrialCollector application. Data downloading and meter handling would be provided to the TrialCollector by the driver functionalities, in other words, the driver interfaces. To test these driver's interfaces, *AscensiaContourDriver* class would be the starting point for initializing the Contour driver. This public class is used for Contour meter specifically and extends *IBiometerDriver* java interface class, which contains all interface methods that need to be analyzed. In *AscensiaContourDriver* class, there are methods such as *openSession, closeSession, getMeasurements,* which inherits from the *IBiometerDriver* interfaces. These methods are what need to be tested by the Biometertest application and can be grouped as testing categories to be performed.

Contournextdriver module also consists of a list of biometer test tasks which represent all testing categories and a *ContourTestRunner* class for initializing a *biometerDriver* as well as mapping and running the selected testing categories. Brief details of how *ContourTestDriver* class is constructed are unveiled in listing 25.

```
class ContourTestRunner(context: Context, btDevice: BTDevice) :
      TestRunner(btDevice) {
    . . .
    private val biometerTestTasks = mapOf(
        TestCategory.OPEN_SESSION to
        OpenSessionTestTask(biometerDriver, btDevice),
        . . .
        TestCategory.CLOSE_SESSION to
        CloseSessionTestTask(biometerDriver, btDevice)
    )

    override fun initBiometerDriver(context: Context) {
        . . .
        biometerDriver = AscensiaContourDriver(context, writeLog)
        . . .
    }

    override fun runTests(categories: List<TestCategory>):
    List<TestResult> {
        return categories.map { biometerTestTasks[it]!!
```

```
                .runTests() }.flatten()
    }
}
```

Listing 25 ContourTestDriver.kt.

In listing 25, when overriding the abstract method *initBiometerDriver* from *TestRunner* class, *ContourTestRunner* instantiates the driver by creating an *AscensiaContourDriver* instance and this driver instance would be passed as a parameter to all *BiometerTestTask* objects such as *OpenSessionTestTask*, *CloseSessionTestTask.* Each of these biometer test tasks are then linked with a certain *TestCategory* enum when being declared inside the *biometerTestTasks* mapping object. Afterward, depending on which test categories are required to be run, *runTests* function will perform the corresponding biometer test tasks and return a list of test results.

Looking more closely into one of the biometer test tasks, for example: *OpenSessionTestTask* (listing 26 below), a collection of test cases – or also known as test suite – is run when a specific task is performed.

```
class OpenSessionTestTask(private val biometerDriver: IBiometerDriver,
private val btDevice: BTDevice):BiometerTestTask {
    override fun runTests(): List<TestResult> {
        return listOf(openSession(), openAlreadyOpenedSession())
    }
    //Call to openSession() on a closed session, should return
    tcbdErrNone
    private fun openSession(): TestResult {
        . . .
        var errorCode = ErrorCode.tcbdErrNone
        errorCode = biometerDriver.openSession(btDevice)
        biometerDriver.closeSession()
        return TestResult("OpenSession 1", . . . )
    }
    //Call to openSession() on an open session, should return
    tcbdErrLibAlreadyOpen
    private fun openAlreadyOpenedSession(): TestResult {
        . . .
        var errorCode = ErrorCode.tcbdErrNone
        biometerDriver.openSession(btDevice)
```

```
        errorCode = biometerDriver.openSession(btDevice)
        biometerDriver.closeSession()
        return TestResult("OpenSession 2", . . .)
    }
}
```

Listing 26 OpenSessionTestTask.kt.

As can be seen in listing 26, when *runTests* method – overridden from *BiometertestTask* interface - is invoked, a group of test cases such as *openSession* and *openAlreadyOpenedSession* is performed and returned as a list of *TestResult* objects. These test results contain useful information such as the test error code, status, description, logs, and execution time. Besides, the *biometerDriver* (also known as *AscensiaContourDriver*) is used to call the driver interface method and returns the matching error code. Taken *openSession* test case as an example, *biometerDriver* calls the *openSession* interface method with a *btDevice* (current paired biometer device) parameter and if it returns *tcbdErrNone* then the test case passes.

4.2.4 Other external dependencies

During Biometertest development, there is a need for visualizing the testing results and having a tool to easily keep track of the testing activities. Therefore, a new feature has been added to the application to improve the testing assessment process as well as debugging test failure. With kotlinx.html library, an HTML report can be generated after each test suite is performed. To start using this external library, first, it has to be added to the build.gradle file (see listing 27).

```
dependencies {
    //Kotlinx.html
    implementation("org.jetbrains.kotlinx:kotlinx-html-jvm:${ko.
    tlinx_html_version}")
}
```

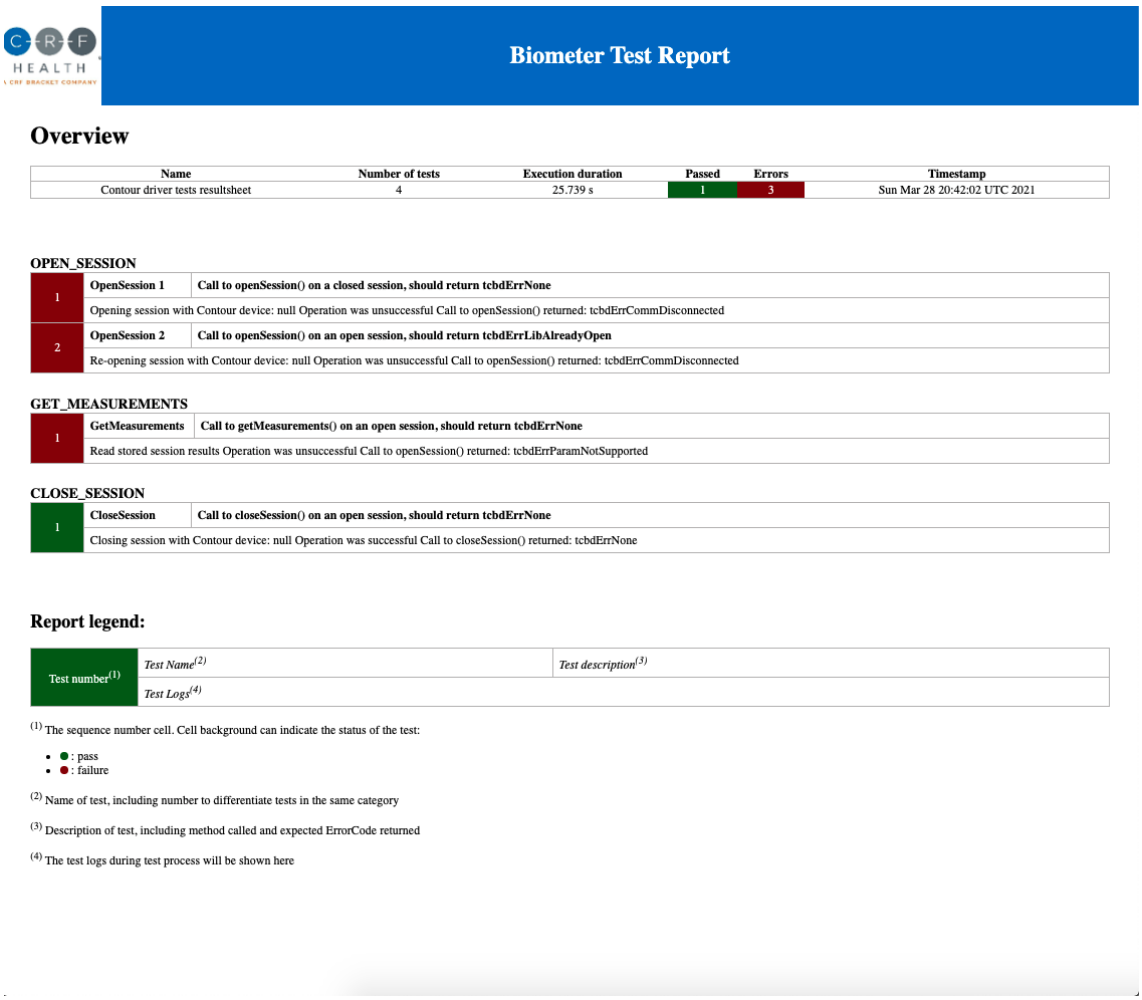
Listing 27 Adding kotlinx.html in build.gradle file.

Next, in *ReportGenerator* class, a report.html file is created by html dom generation and stream transformation every time a test activity is run (listing 28).

```
class ReportGenerator : KoinComponent {
    private val categoryDao: CategoryDao by inject()
    suspend fun createReport(context: Context, testResults:
MutableList<TestResult>) {
        val document = DocumentBuilderFactory.newInstance()
        .newDocumentBuilder().newDocument()
        val html = document.create.html {
           //html dom is created here
           . . .
    }
    val filePath = context.getExternalFilesDir(Environment.
    DIRECTORY_DOCUMENTS)
    val file = File(filePath, "report.html")
    intoStream(html, file)
}
```

Listing 28 ReportGenerator.kt.

As can be seen in listing 28, after HTML DOM is created by kotlinx.html library, it is transformed and written as a report.html file located in external directory of the device. The specific location of the report can be found at the absolute path in the device: *SDcard/Android/data/com.signanthealth/files/Documents/report.html* and the report would be as figure 15 below.



Biometer Test Report

Overview

Name	Number of tests	Execution duration	Passed	Errors	Timestamp
Contour driver tests resultsheet	4	25.739 s	1	3	Sun Mar 28 20:42:02 UTC 2021

OPEN_SESSION

1	OpenSession 1	Call to openSession() on a closed session, should return tcbdErrNone
	Opening session with Contour device: null Operation was unsuccessful Call to openSession() returned: tcbdErrCommDisconnected	
2	OpenSession 2	Call to openSession() on an open session, should return tcbdErrLibAlreadyOpen
	Re-opening session with Contour device: null Operation was unsuccessful Call to openSession() returned: tcbdErrCommDisconnected	

GET_MEASUREMENTS

1	GetMeasurements	Call to getMeasurements() on an open session, should return tcbdErrNone
	Read stored session results Operation was unsuccessful Call to openSession() returned: tcbdErrParamNotSupported	

CLOSE_SESSION

1	CloseSession	Call to closeSession() on an open session, should return tcbdErrNone
	Closing session with Contour device: null Operation was successful Call to closeSession() returned: tcbdErrNone	

Report legend:

Test number(1)	*Test Name(2)*	*Test description(3)*
	Test Logs(4)	

(1) The sequence number cell. Cell background can indicate the status of the test:

- : pass
- : failure

(2) Name of test, including number to differentiate tests in the same category

(3) Description of test, including method called and expected ErrorCode returned

(4) The test logs during test process will be shown here

Figure 15 Report.html overview.

Looking through the report.html in figure 16, an overview of a test run is shown with quite a lot of information. Readers can see how many tests have been performed, total execution time, a number of pass and fail tests, and the exact timestamp at the test run happened. Furthermore, each test category is listed with test cases including test number, name, description, logs, and status. Finally, at the end of the report, there is a “Report legend” section explaining how to read the report correctly, including test status color and cell meanings. By using this feature of the Biometertest application, Signant Health developers and testers can have a brief description of what tests have failed and reasons that cause the failures, therefore, have a better solution for debugging and fixing the biometer driver functionalities.

5 Application practicalities

5.1 Comparison to other test automation tools

Software testing is expensive and requires enormous investment. Currently, there are a wide variety of test automation tools which can automatically execute test cases, generate test results conveniently without any human involvement, and reduce drastically testing cost and effort. However, software developers need to be fully aware of all available test automation instruments in order to implement and gain entirely benefits from them. [27, p. 90.] Back on the topic of improving test automation for Contour driver, although there are many present test automation tools such as Appium, Robotium, Selendroid, …, these test automation solutions can merely create requests for returning and interacting with UI elements, also known as GUI testing [28, p. 3629]. These current tools do not have the ability to call and verify the correctness of a low-level implementation such as the biometer driver; therefore, Biometertest application was developed to meet that need of validating the behaviour of the in-house developed biometer driver code.

Furthermore, there are other reasons why Biometertest application is a better solution over other existing test automation tools. With the current implementation of Biometertest codebase, developing and debugging the driver are no longer pain points. While running the application with Android Studio, a breakpoint can be set inside the contournextdriver code (through the jar classes), thus, while running the application, developers can easily spot any error conditions and read the useful information of the current data structure (figure 16). This used to be a real headache for Signant Health's developers because it takes a long time to test and debug the biometer interface APIs manually via a study protocol.

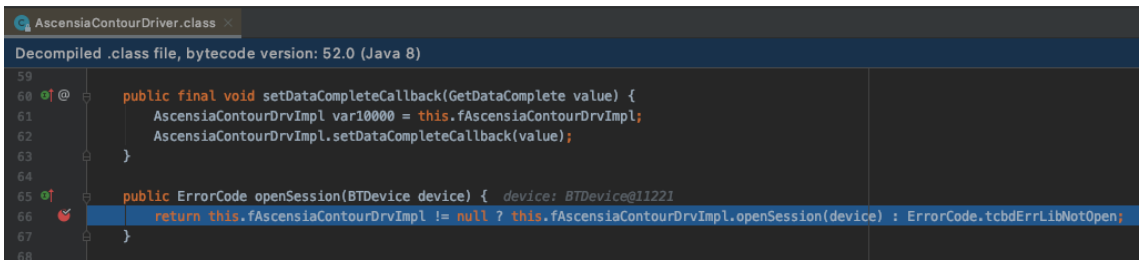


Figure 16 Debugging AscensiaContourDriver.kt class inside Biometertest application

Beside the handy functionality of allowing development team to maintain and improve the biometer driver, the application also greatly contributes many newly acquired skills (such as Android Jetpack libraries, dependency injection, Coroutine) to other company Android-based products, thus, improves developers' competency for current and future projects. In conclusion, these reasons explain why Biometertest application was chosen to be developed, rather than other existing test automation tools instead.

5.2 First step in increasing test automation coverage

Although Biometertest application is not a complete automation solution when there are still manual activities in running the application, such as pairing the Contour device, pressing buttons to perform tests, using Biometertest has proved to drastically reduce the manual testing time for the driver implementation. Usually, to validate the functionalities of a driver inside the TrialCollector application, there is a time-consuming traditional process to be followed and this process can be briefly described in steps as below:

- First, a study protocol, which is also known as the implementation of a clinical study, has to be deployed to TrialCollector application. For Signant Health's employees, this step could take even more than half an hour to complete the end-to-end flow for study deployment.

- Next, TrialCollector has to be initially set up with many manual steps for creating the study subject and then paired with a meter (such as Contour, Dexcom, TaiDoc). This step takes 5-10 minutes to be fulfilled.

- After that, to start testing the driver behaviour, developers have to run each of the protocol commands in TrialCollector application for calling the biometer interfaces and verify that there is no error happening during the interaction. This step involves many manual works and there is no record to keep track of.

To illustrate more clearly the difference between using Biometertest application and traditional way to test the driver functionalities, figure 17 is plotted below.

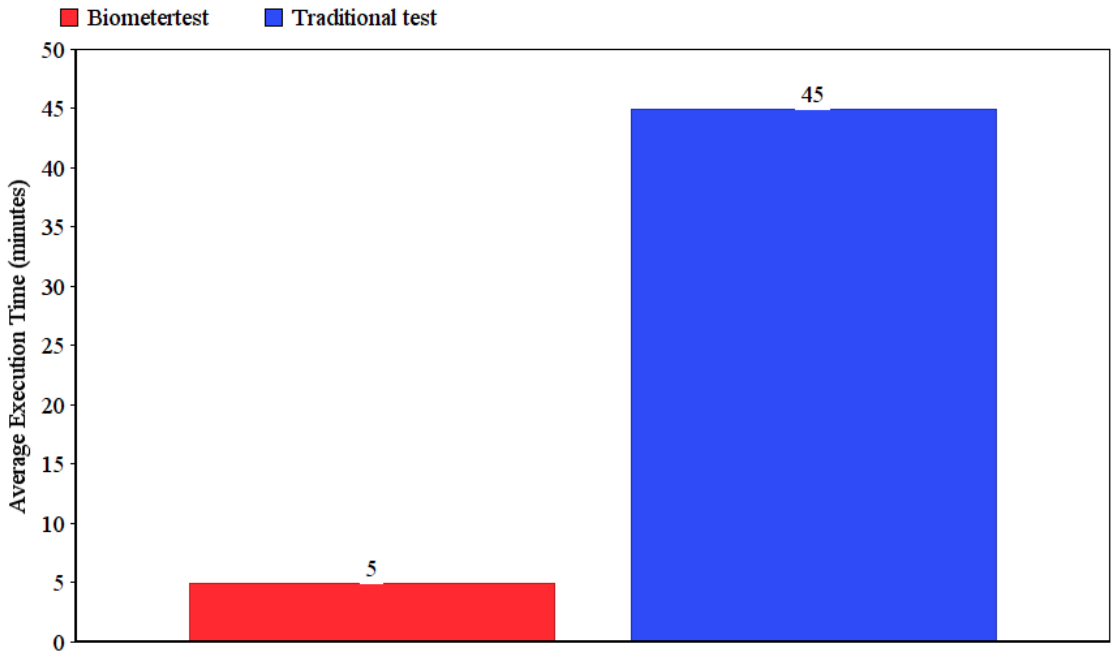


Figure 17 Average execution time comparison between using Biometertest app and traditional test

In figure 17, there is an enormous reduction in testing time by using Biometertest application. For traditional testing, it takes an average of 45 minutes to finish the manual testing compared to 5 minutes of using the newly developed app. Therefore, the Biometertest is a great initial step to improve the test automation coverage for biometer drivers as well as reduce manual testing activities for third-party devices such as Contour. In the future, there is a high chance to fully automate this application by using Jenkins build to run a suite of tests with predefined test cases, provided that there is a mock implementation for simulating the real meter data. With that improvement, Biometertest application would be a complete automation tool and serve as a valuable asset of the company.

5.3 Future improvement and possible features

At the moment, Biometertest solves the main problem of speeding up the testing process of the Contour driver. Nonetheless, there are still many areas which need further improvement in the future:

- Firstly, the application .apk file can be auto-generated and stored by Artifactory deployment. In this way, the latest version of the application can be easily accessed by Signant Health employees through every Artifactory build.

- Secondly, a possibility to run and display tests for multiple meter devices (for example: two Contour meters) at the same time can be added to the application. This improvement can greatly speed up the testing process when there are multiple biometer devices to be validated.

- Thirdly, adding support to different biometer devices such as Dexcom, Taidoc, MyGlucoHealth would be a possible improvement for the application. With the current project structure (abstracting drivers into separate modules and have one module for common logic), this is the most feasible achievement in the future.

- Fourthly, a mock implementation can be developed to simulate the data of the real meter device to eliminate the manual pairing process. Currently, Dexcom mock has been developed by Signant Health's developers so Contour mock can have the same implementation as well as other meters.

- Lastly, as the application still cannot automate the testing activities of the driver, this can be improved in the future by a continuous integration tool such as Jenkins to run a suite of tests with a single command line, therefore, removing completely the human manual involvement.

6 Conclusion

The goal of the project was to introduce a solution for improving test automation of Contour Next One meter via developing an application using the latest Android technologies. In the end, the project's objectives were achieved by implementing Android Jetpack tools and performing Bluetooth connection with the biometer. Most importantly, validation specialists are now able to fasten the Contour validation process by using the testing functionalities of the application, and the time for each biometer driver testing has been significantly reduced. Besides, due to the intuitive approach provided by Biometertest application, SignantHealth's developers have a better tool to debug and develop the Contour driver as well as track the test results in the future.

Furthermore, a fully developed application was delivered with operative and functional capabilities after the project finished. Users can scan and connect with the biometer through Bluetooth connection, perform meter functionalities testing individually or in groups, and investigate testing results in detail. From a developer's point of view, codes in Contour driver can be debugged through setting breakpoints in the driver jar classes, which allows faster error detection and makes the software development process tension-free and uncomplicated.

Even though there is still room for improvement in the future such as providing the .apk in a more structured way on Artifactory, enhancing testing capability by running with multiple devices, expanding the driver to other biometers, or fully automate the testing process, there is no doubt that the application has contributed greatly to the effort of improving test automation coverage throughout the company and marked a milestone of switching from manual testing to automated testing.

References

1. Rudolf Ramler and Klaus Wolfmaier. Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost; 2006.

2. Juha Itkonen, Mika V. Mäntylä, and Casper Lassenius. How Do Testers Do It? An Exploratory Study on Manual Testing Practices; 2009.

3. S. O'Dea. Market Share of Mobile Operating Systems Worldwide 2012-2020; 2020.

4. Google. Android Jetpack Documentation. URL: https://developer.android.com/jetpack. Accessed 9 October 2020.

5. Glenford J. Myers. The Art of Software Testing, Second Edition; 2004.

6. Andreas Leitner, Ilinca Ciupa, Bertrand Meyer, Mark Howard. Reconciling Manual and Automated Testing: the AutoTest Experience. URL: https://ieeexplore-ieee-org.ezproxy.metropolia.fi/stamp/stamp.jsp?tp=&arnumber=4076909&tag=1. Accessed 3 March 2021.

7. Ossi Taipale, Jussi Kasurinen, Katja Karhu, Kari Smolander. Trade-off between Automated and Manual Software Testing; June 2011. URL: https://www.researchgate.net/publication/257798848_Trade-off_between_automated_and_manual_software_testing. Accessed 3 March 2021.

8. Jetbrains. Kotlin Documentation. URL: https://kotlinlang.org/docs/reference/comparison-to-java.html. Accessed 9 October 2020

9. Stackoverflow. The 2020 Developer Survey. URL: https://stackoverflow.blog/2020/05/27/2020-stack-overflow-developer-survey-results/. Accessed 9 October 2020.

10. Neil Smyth. Android Studio 3.6 Development Essentials - Java Edition; 2020.

11. Google Android Developers Blog. Use Android Jetpack to Accelerate Your App Development. URL: https://android-developers.googleblog.com/2018/05/use-android-jetpack-to-accelerate-your.html. Accessed 2 October 2020.

12. Ascensia Diabetes Care Holdings AG. Contour®Next One User Guide; 2018. URL: https://www.contournextone.ca/siteassets/web90002727_cntrnxtone_ug_r11-18.pdf. Accessed 2 October 2020.

13. Signant Health confidential document. System Design Specification: Android Trial Collector - Contour Driver; 26 September 2019

14. Signant Health confidential document. System Design Specification – Biometer Interface: Android TrialCollector; 15 August 2019

15. Signant Health confidential Confluence page. Biometertest Application; 23 April 2020

16. John Gossman. Introduction to Model/View/ViewModel Pattern for Building WPF Apps; 10 August 2005. URL: https://docs.microsoft.com/en-us/archive/blogs/johngossman/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps. Accessed 3 March 2021

17. Vandana Srivastava. MVC vs MVP vs MVVM Architecture in Android; 07 October 2019. URL: https://blog.mindorks.com/mvc-mvp-mvvm-architecture-in-android. Accessed 3 March 2021

18. Google. Save Data in A Local Database Using Room. URL: https://developer.android.com/training/data-storage/room. Accessed 3 March 2021

19. Google. Defining Data Using Room Entities. URL: https://developer.android.com/training/data-storage/room/defining-data. Accessed 3 March 2021

20. Google. Accessing data using Room DAOs. URL: (https://developer.android.com/training/data-storage/room/accessing-data. Accessed 3 March 2021

21. Google. Navigation. URL: https://developer.android.com/guide/navigation. Accessed 3 March 2021

22. Google. ViewModel Overview. URL: https://developer.android.com/topic/libraries/architecture/viewmodel. Accessed 3 March 2021

23. Google. LiveData. URL: https://developer.android.com/reference/android/arch/lifecycle/LiveData.html. Accessed 3 March 2021

24. Google. Data Binding Library. URL: https://developer.android.com/topic/libraries/data-binding). Accessed 3 March 2021

25. Google. Kotlin Coroutines on Android. URL: https://developer.android.com/kotlin/coroutines. Accessed 3 March 2021

26. Kotlin github. Kotlinx.coroutines Reference Documentation. URL: https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/. Accessed 3 March 2021

27. Vahid Garousi, Frank Elberzhager. Test Automation: Not Just for Test Execution; 28 March 2017. URL: https://ieeexplore-ieee-org.ezproxy.metropolia.fi/document/7888399. Accessed 3 March 2021

28. Shiwangi Singh, Rucha Gadgil, Ayushi Chudgor. Automated Testing of Mobile Applications using Scripting Technique: A Study on Appium; October 2014. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1049.9945&rep=rep1&type=pdf. Accessed 3 March 2021