Expertise
and insight
for the future

Hang Duy Khiem

Android Optical Character Recognition Solution Based on TacoTaco Framework

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

18 April 2019

Author(s) Title Number of Pages Date	Hang Duy Khiem Android Optical Character Recognition Solution Based on TacoTaco Framework 44 pages + 0 appendices 18 April 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructor(s)	Patrick Ausderau, Principal Lecturer Mykhailo Shchurov, Thesis Advisor

TacoTaco is an Android application development framework based on Model-View-Interactor (MVI) model and Robert Martin's Clean Architecture. It serves as the main Android framework in the internal development processes of Wolt Enterprise Osakeyhtiö. Although it has been used extensively in the primary Android app of the company, its practicality in a generic Android application has not been thoroughly tested.

The thesis aims to provide a comprehensive analysis to ascertain if the framework can perform well against a different concrete use case. This was achieved by building an Android application using the framework as its base.

Optical Character Recognition (OCR) is a technology used to transfer text from one media to another, e.g. from paper to a digital format. Even though the technology has been around for almost a century, its recent surge in popularity stemmed the invention of smartphones. Thus, it serves as a reliable use case for the application. Because of its versatility and accuracy, the Tesseract OCR engine was chosen as the fundamental upon which the application is built.

It is concluded from the application that TacoTaco fits for general usage and that the OCR results from Tesseract are decently collected. However, more comparison with other frameworks and OCR engines, as well as in-depth performance optimization, would be a good addition to further verify their usages.

Keywords	Android, OCR, TacoTaco, framework

Contents

List of Abbreviations

API Application Programming Interface.

CRUD Create, Read, Update, and Delete.

DAO Data Access Object.
DI Dependencies Injection.

I/O Input Output.
IDE Integrated Development Environment.

JDK Java Development Kit.
JNI Java Native Interface.
JSR Java Specification Requests.
JVM Java Virtual Machine.

MVC Model-View-Controller.
MVI Model-View-Interactor.
MVP Model-View-Presenter.
MVVM Model-View-ViewModel.

NDK Native Development Kit.

OCR Optical Character Recognition.
OS Operation System.

PDF Portable Document Format.

SDK Software Development Kit.
SQL Structured Query Language.

UI User Interface.
URI Uniform Resource Identifier.

Glossary

Android Smartphone operation system based on Linux kernel, developed mainly by Google.

Android Studio Integrated development environment used specifically to develop android application.

Cmake Open sources tool to manage the building process of software. It allows user to test, compile and create packages of the source code..

Controller Component to control the UI and user interaction. Used in many modern architecture, including TacoTaco.

Interactor TacoTaco component that handles the business logic of an application..

Java reflection	A java development technique that allow for examining and modify methods, classes, interfaces at runtime.
Looper	Android component to keep track of the message loop for a thread.
Room Persistent Library	A solution to abstract SQLite access in Android development.
Single Activity	Android Development paradigm that allow for only one activity.
SQLite	A database management system embedded on most Android devices currently in use.
Thread	Smallest unit of work that can be managed by a scheduler that included in modern operating systems.
ViewPager	Android component to manage the lifecycles of multiple views, and allow users to flip through them.

1 Introduction

Optical Character Recognition (OCR), or to put simply, the ability of machines to translate text from one medium to another, has been a topic of scientific interest even before the conception of the first computer, with the invention of the Optophone and Statistical Machine in 1914 and 1931 respectively. However, for the next few decades, there was little application of such technology to the general public, as most of these mechanisms were intended for a very specific user group (e.g. handicapped person, media cooperation, etc). [1] [2]

It was not until the early and mid-1990s that the capability of OCR technology reached the typical consumer, when companies such as HP, Xerox, AABBY started to manufacture scanner, printer and software with embedded OCR [3] [4] [5]. Today, thanks to the availability of a high-resolution camera, coupled with powerful computing processor commonly found on a smartphone, OCR is more widespread than ever. As of 2019, the most popular mobile application that utilises OCR technologies - Google Translate - got more than half a billion of downloads worldwide [6].

The full potential of the field, however, has not been fully explored. To date, countless innovative contributions have been pouring nonstop to develop the technology. As a result, this introduces endless possible applications to improve the life of ordinary people. To explore these possibilities, besides the core concept and implementation of OCR, a good understanding of application construction on the mobile platform is needed.

Android Operation System (OS), since its introduction to the general public in 2008, has grown to be the leader of the smartphone landscape. With all the smartphone capacities for OCR, and the reach of 2 billion actives devices, Android is a great fit to facilitate the comprehension of the mobile application development.

Combining the state of the art technology, toolkit and best practices from Google - Android maintainer - and third parties, Wolt Enterprise Oy has created Taco Taco application framework, with the goal of being the go-to starting point for Android application develop-

ment. Through using, exploring and extending the framework, the underlying mechanism of building a functional, maintainable and flexible application can be methodically understood.

Although TacoTaco has previously been trialled in Wolt main application, being the driving force behind the changes in the Android development strategy of the company, there has not been any usage outside of it. This is the reason why more in-depth tests against an external application are needed for TacoTaco to be a generally applicable framework.

To accumulate the knowledge regarding the current state of OCR on Android and evaluate the main characteristic and implementation of the TacoTaco framework, especially regarding the usage in a real-world scenario, which is building a picture-to-word application, are the main topics of this thesis. An OCR-capable application is built using TacoTaco, and conclusion of the topic can be then drawn from the process of developing this application.

The thesis is divided into four main sections: the theory, the implementation, discussion, and the conclusion.

2 Theory

2.1 General Android development tools and practices

The current landscape of Android development is very much different from the starting point almost ten years ago. More and more approach to better organization and work-flow has been adopted by the developer community. This is evident in the number of tutorials, tools, and frameworks made for this particular subject, which figure 1 clearly demonstrates.

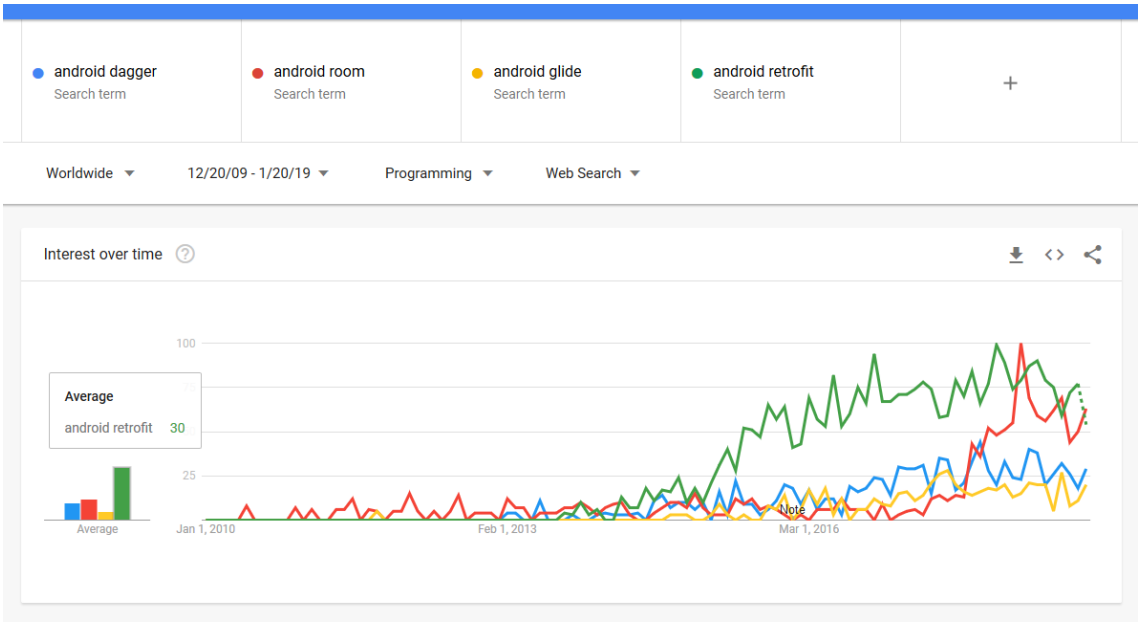

Figure 1: Android Dagger dependencies injection framework and Room database search trends show the maturity of Android development. Copied from Google Trends (2019). [7]

The need to choose a clear direction in the methods of development arose early in the process of developing the sample application. Among the numerous ways going forward, these had been chosen as the main mechanism of the application:

- Dagger Dependencies Injection Framework
- Room Database
- Android Reactive Extension
- Glide Image loading library

2.1.1 Dagger Dependencies Injection Framework

Dependencies Injection (DI) is a programming technique used to supply the instance variables of an object, instead of creating the variables inside of the object. Doing this can simplify the coding process and enhance the configurability as well as maintainability of the application [8].

To demonstrate DI concept, listings 1 and 2 are provided. In listing 1 the database is the dependency of the Example class and is created in the constructor. On the other hand, in the Example class in listing 2, the database has been created elsewhere and is passed into the class as a dependency.

```
1 class Example() {
2     private lateinit var database;
3
4     Example(){
5         database = Database()
6     }
```

Listing 1: Creating the variable

```
1 class Example() {
2     private lateinit var database: Database
3
4     Example(database: Database){
5         this.database = database
6     }
```

Listing 2: Injecting the dependency

Dagger framework was made by Square, and later officially adopted and maintained by Google. The framework is available to all JVM compatible languages, including Java, Kotlin, Scala, Groovy while not being limited to them. With concepts such as Module (to provide dependencies) and Component (to inject dependencies), along with injection annotation compatible with Java Specification Requests (JSR) 299[1], the framework helps to build complicated dependencies graph with a few lines of code. [9]

Unlike many DI frameworks available on the market, such as Guava and Spring, Dagger does not use Java reflection to create the dependencies graph. Instead, Dagger is imple-

[1] https://jcp.org/en/jsr/detail?id=299

mented in order to create the graph in compile time, thus mitigate the performance impact on Android runtime. This makes Dagger a good fit for the thesis project. [10]

2.1.2 Room Persistent Library

Room Persistent Library is a SQLite Object relation mapping layer. Room Persistent Library version 1.0 was introduced by Google on November 6th, 2017, as a mean to streamline the communication between Android application and the included SQLite database, replacing the traditional means of accessing SQLite by `SQLiteHelper`. [11]

Rather than querying Structured Query Language (SQL) directly to the SQLite engine, which is very much error prone and could result in data corruption if carried out incorrectly, the user of Room Persistent Library will only need to interact with the model that they come up with in their Android code, regardless if it is written in Java or Kotlin. By defining these models, the user is, essentially, creating a table in the database. Annotations such as `@PrimaryKey`, `@ColumnInfo`, etc., are also provided in order for the user to shape the table based on their data model.

To perform Create, Read, Update, and Delete (CRUD) operations to the entries in these database, a Data Access Object (DAO) interface is needed. In the DAO, methods for accessing the underlying SQL database are listed. Annotations are used to bind the SQLite query to these methods, to provide the adaptability necessary to be used fluently within the Android context. Three simple CRUD operations are also provided as is by the library: `@Insert`, `@Delete`, `@Update`.

Finally, a wrapper for the database is needed to refer to the database within the application. This reference can be created by a builder provided by Room itself, and it acts as the main access point to the underlying SQLite database. [12] The relationship between the DAO, the database wrapper, the entities, as well as the actual SQLite database, is demonstrated in figure 2.

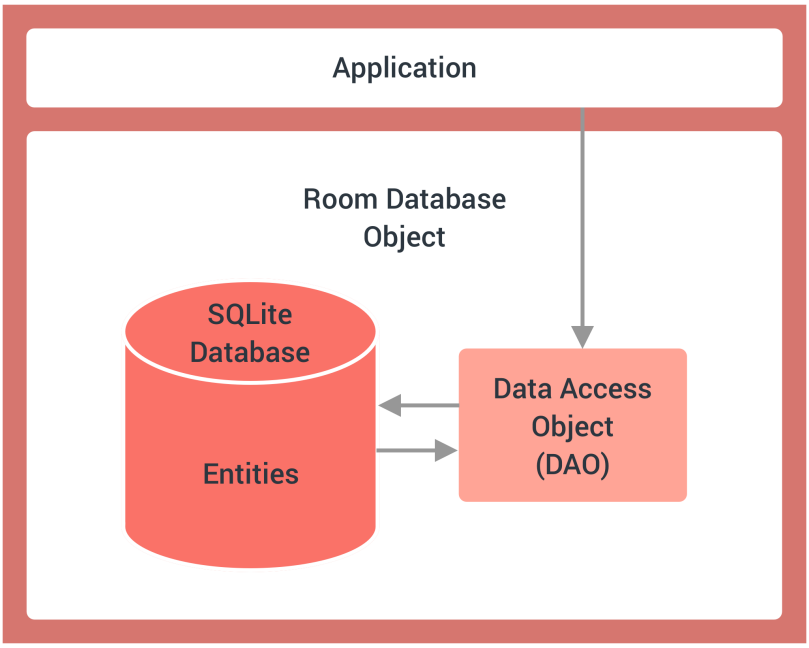

Figure 2: Architecture diagram of Room Persistent Library

2.1.3 Android Reactive Extension

Reactive Extension is an open source standard created by the community to extend imperative languages by adding the functional reactive paradigm into them. This paradigm allows for the manipulation of a stream of data in the same way that a collection of data could be manipulated, abstracting away the complication behind the streaming process, such as Input Output (I/O), synchronization, thread safety, etc. [13]

To accomplish this goal, Reactive Extension makes heavy use of the Observable/Observer software design pattern. Every unique data point, for example, database data, network data, cache data can be regarded as a data publisher. Observables, or trivial packages of data, are emitted by these publishers, creating a stream of data. On the end, observers subscribe to these streams and get notified about each and every new data. They can then manipulate the data, or send them to other observers. [14]

One of the main usages of a typical Android application is to manipulate and display data from these data points. Thus, RxJava[2] and RxAndroid[3] were made based on the Reactive Extension standard. RxJava adds the ability to uses data stream into Java and Kotlin,

[2]https://github.com/ReactiveX/RxJava
[3]https://github.com/ReactiveX/RxAndroid

providing the application developers with Observable/Observer type, and the related functionality like maps, zip, join. Meanwhile, RxAndroid gives the developer the ability to bind the data processing to separate Thread and Looper, simplifying the threading process.

By combining both RxJava and RxAndroid, developers can stream data from a remote location to their Android device, observe the data in the I/O thread, process the data in another background thread, and display the data in the main thread. This approach, while being functional and concise, results in no visible delay in the user interface, which was only achievable previously using the callback heavy method of `AsyncTask`.

2.1.4 Glide Image Loading Library

Although displaying images in Android is as simple as setting the image resources to the correct `ImageView`, managing memory resources and juggling overall performance is not a trivial task. For instance, loading a large bitmap onto an `ImageView` that is not properly initialized for bigger images will result in an `OutOfMemoryException`. To address this problem, along with other issues encountered while loading image, a more sophisticated approach is needed.

Many image loading libraries have been open sourced with the goal of making the image loading and displaying process faster, more efficient, more straightforward, while still remains as developer-friendly as `ImageView` Application Programming Interface (API). The major libraries that are popular and commonly used in development include Fresco from Facebook[4], Picasso from Square[5] and Glide from BumpTech[6]. These libraries compete aggressively on efficiency, customizability, and simplicity. For a basic application, any library should be sufficient, and consequently, Glide was chosen because of prior familiarity.

Believing that image must not only be displayed fast and correctly but also without causing jank and stuttering on the user interface, Glide performs a number of optimization to the process of loading images. These include re-use of resources to minimizing expensive garbage collections and heap fragmentation, smart downsampling and caching, lifecycle

[4] https://github.com/facebook/fresco
[5] https://square.github.io/picasso/
[6] https://github.com/bumptech/glide

awareness for resources release. [15]

Many different types of image data points work well with Glide: Network stream, cached file, Android Uniform Resource Identifier (URI), decoded bitmap, etc. Glide also provides its user with many customization options, whether its image placeholder, transformation on caching strategy. With these features, coupled with the simple usage, not only does Glide lower the time needed to developed an app, but it also facilitates greatly the development process. [16]

2.2 Framework definition and overview

2.2.1 Software development framework definition

As more of these tools are introduced into the code base to solve complicated business and display requirement, the code base itself gets more sophisticated. The more complex a code base become, the easier it is to make mistake working with it: boilerplate introduced to perform trivial work, hard to test component that is tightly coupled with each other, redundant modules to perform one function or just dead code that has been forgotten. These problems can, at best, make applications hard to work with, lengthen the development time, and affect developer satisfaction, and in the worse case, can hinder the advancement of the application in the long term.

To resolve and prevent these complications, more generic tooling is needed to organize and manage the code base: Framework. By its definition, a software development framework is an abstraction layer that provides an application with the foundation upon which more functionality can be built. This abstraction layer consists of the base solutions for most frequently encountered use case: common patterns, tested functionality, and supporting structures. [17]

These aspects of frameworks make them quite similar to what known in software development as libraries. However, unlike libraries where only functionality is provided, a framework also acts as a general guideline for the development process. Whether adding a new component, modifying existing ones, or structuring the dependencies between these components, the control flow of the application is guided by the framework. This may appear

rigid at first, but in the long run results in much better structured and self-documented code base. Because of these characteristics, frameworks are considered extensively reusable: a framework targeting a specific platform can be used as the groundwork to build different applications for that particular platform. [18]

2.2.2 Android framework general

An Android application may deal with different parts of the Android environment, including all sensors, devices, storage, as well as process, thread and file system. To help developer interact with these complex underlying functionality, Android Software Development Kit (SDK) was published along with the OS. Because it provides the much-needed abstraction layer and is the foundation on which more Android application can be written, Android SDK can be considered a framework [19].

The Android framework structure is provided in figure 3. At the core of the framework lies the Android Kernel, which is a modified version of the Linux Kernel[7]. Its main function involves managing the operation of the device and its hardware. The Android Runtimes wraps around the Kernel, exposing the native functionalities by translating Java byte code to machine code. Useful native libraries, such as OpenGL, SQLite, are implemented using these components, as well as direct access to the Kernel. Finally, the Android application framework creates the building blocks of Android application - Fragment, Activity, Service, to name a few - based on the foundation provided by the inner layers. [20] However, because of the interaction with the low-level components, Android SDK itself is low level.

The first successful batch of high-level, architectural frameworks implemented for Android was based on Model-View-Controller (MVC) [21], which derived from their original version written on SmallTalk in the 1980s. Soon after, based on the implementation of Google I/O 2009 Large scale application development and MVP presentation[8], and Microsoft's own Windows Phone, Model-View-ViewModel (MVVM) and Model-View-Presenter (MVP) based frameworks were also made available on the OS. To gain better access to data, the principles of Robert C. Martin's Clean Architecture[9] is also introduced to work alongside

[7] https://www.kernel.org/
[8] http://www.gwtproject.org/articles/mvp-architecture.html
[9] https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

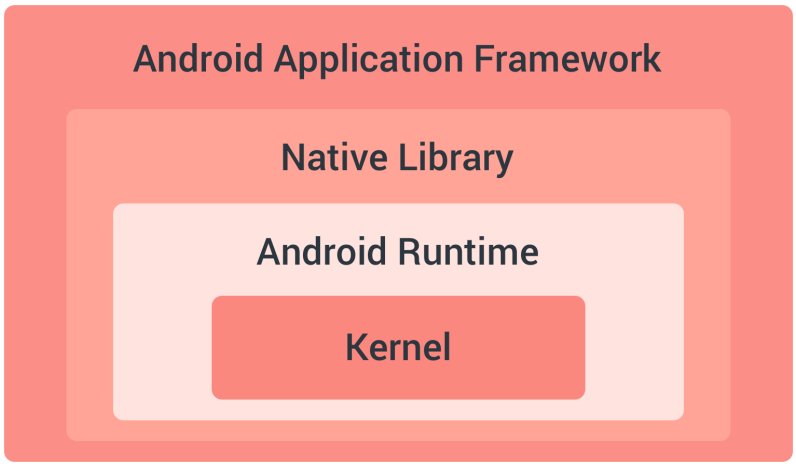

Figure 3: The original Android application framework

with these frameworks. Non-architectural, display-oriented frameworks such as Flow and Conductor also make their presence known on the platform. [22]

2016 was the year in which Google decided to finally take a stance on which architecture should Android developer use. A Github repository named Android Architecture Blueprints[10] was published, containing guide and samples on how Android application should be built. One year later, During Google IO 2017, Android Architecture Components[11] was published, as the official way that Google uses to build their own application. This later evolves to what currently known as Android Jetpack[12], a collection of libraries, components and architectural framework that follow the best guideline and practices, ensuring smooth Android development process.

As welcome as these changes are, Google was a step too late. With multiple high quality, extensively tested and widely used architectural and non-architectural frameworks available currently on the market, development for Android has been more fragmented than ever. There exists no single way to build an application. Thus, by trial and error, Wolt Oy built their first Android Application, which was the foundation of the TacoTaco Framework.

[10] https://github.com/googlesamples/android-architecture
[11] https://developer.android.com/topic/libraries/architecture
[12] https://developer.android.com/jetpack

2.3 TacoTaco in depth

2.3.1 History, motivation and general characteristic of TacoTaco

Wolt Oy was created in October 2014 by 6 co-founders, with the goal to help users discover and get great food in their city. The first Android development team of Wolt consisted of only one person, who was also taking part in developing the backend application and other algorithms. Getting the product ready was vital to the survival of the company, thus the general maintainability of the Android code base was not the first priority.

Realizing the importance of having a good Android client, Wolt Oy hired the first addition to the team. Mykhailo ”Mike” Shchrurov, the first co-creator of TacoTaco, started working on the code base in May 2016. This was when the first framework was introduced gradually into Wolt Android application. This framework was an implementation of the MVP pattern, using the Conductor presentation framework for UI composition, and interfacial contract as a mean to enforce the MVP constraint. Supporting library, such as Dagger DI, RxJava, RxAndroid, and Fresco were also added, to improve the base setup of the project.

On November 2016, the second co-author of TacoTaco - Juha Leppilahti - joined the team, helping with refactoring the code base to MVP and adding more feature to the application. The refactor took about 6 months to its completion, but by then, multiple drawbacks of the framework and its guideline started to show: there was no single state object leading to complication when dealing streamed UI event, and there was also a navigation problem from UI components on one UI node to component from another, as well as Conductor misbehaving with `ViewPager` and lifecycle, etc. This called for another refactor, which resulted in the conception of TacoTaco.

TacoTaco design and initial development started in Spring 2017, with the aim of developing a replacement that can fix the problem encountered with the MVP based framework. Learning from contemporary architectural frameworks, as well as past mistakes, the first design of TacoTaco pinpointed the general characteristic of the framework: `Single Activity`, State-driven rendering, Command based intercomponent communication and clean data access. This was the backbone philosophy on which TacoTaco was built. To achieved this, the team took inspiration from Model-View-Interactor (MVI) architecture

model and Robert Martin's Clean Architecture.

TacoTaco was first included in the main Wolt Android application on the 3rd of August, 2017. The version brought the much needed changes into the code base introducing concepts such as Controller, Interactor and Renderer. Many features, such as an event bus based on RxJava, smart findViewById, transition animations, were added along the way, resulting in the release of TacaTaco 2.0, which is the current version of the framework. This is the version that will be discussed in this thesis.

In TacoTaco 2.0, the main architectural principles on which the first version was built, remains unchanged. An application built on the TacoTaco framework should have a `Single Activity` UI rendering model. The rendering of the application should be based on current application states. Interactable component of this application should only communicate with other non-UI components by concrete Command. Should the application need to process and display external data, the principle of Clean Architecture needs to be followed. To achieve this, primitive TacoTaco architectural components - Controller and Interactor - need to be adopted.

2.3.2 Controller

In the context of TacoTaco, a Controller is a primitive unit that represents a UI component and its interaction logic. As such, a Controller can be thought of as a button, a part of the screen, or the UI screen itself. The logic of touching, flinging, and scrolling these components also resides within the Controller.

Controllers are constructed in a tree-like structure. This means that a controller can have other controllers as children, and another controller as its parent. The views which children represent, are displayed on part of the screen that is designated by the parent. As a result of such hierarchy, view composition, i.e. the act of making a view from smaller functional parts, is greatly simplified. A complicated view can be destructed into smaller components which perform one function well, and vice versa: from smaller components of a view, the hierarchy of such view can be constructed to better comprehend the bigger picture. The first node of this tree hierarchy is called the RootController. The installation of the RootController happens on the activity creation. Instead of inflating its own view, in

the onCreate Method, an activity can instead call `TacoTaco.install()`, which returns the RootActivity that covers the activity screen. From here, more controllers can be added to display different UI components and navigate between screens without the need of Fragments or other Activities. This also serves as the reason why TacoTaco is Single Activity.

Each controller has its own model, which predefines the possible states of the controller. Whenever a change occurs to the state from the business logic, TacoTaco triggers an event that provides information for the controller to act on. This information includes the new state of the controller, the previous state it was in before the change, and an optional payload indicating what has changed. Using the information, the controller can then modify the UI to match with the new state.

Animation and Transition are also within the responsibility of TacoTaco. In TacoTaco, the animation happens when the render method is called, along with other UI updates. As such, a model may contain flags for animations. Whenever these flags are received by the controller, UI animation methods can be started, stopped, paused or reset.

Transition to and from a controller in TacoTaco is, however, handled in a different way. Whenever a transition is triggered, the controller may choose to handle it or not. In the case that the transition is to a child controller, the controller can direct the child to the correct place for rendering. However, transitions to a sibling controller, or to controllers that are siblings to the parent controller are also possible. In this scenario, the controller dispatch the transition to the parent, where the transition is handled.

The lifecycle of a controller depends heavily on the transition state. Transition events always motion changes in the lifecycle stages, through an internal component called LifecycleConductor. A controller can transit into any ViewGroup that exists in its direct parent layout and can push the controller currently occupying the ViewGroup into the background. Because of this, LifecycleConductor tracks the backstack that is attached to these ViewGroups, and modifies the lifecycle of the two controllers in the correct manner. To continue the example, the controller transitioning into the ViewGroup will be attached to the parent and inflated, while the exiting controller will be put to the background, possibly deflated and detached.

Whenever a controller detects that it is being detached from the parent controller, a state saving mechanism will spring to action. This mechanism will save all the hierarchy of the view, from its root to its leaf, into a bundle and save it with the normal instance state. Once the controller is attached to its parent again, the correct bundle containing the states and children hierarchy will be passed on to it by its parents. This whole process can also happen when Android OS signalling low memory.

2.3.3 Interactor

Much like Controller, Interactor is a primitive component in TacoTaco, which can be extended to build an application. However, Interactor serves an entirely different purpose than the Controller. If a controller can be used to manage the UI components, render and handle interaction, the Interactor is the one that takes care of the business logic of the application.

An Interactor is considered a dependency of Controller. Simple controller, for example, a yes-no dialog, can be built without using an interactor. However, the Interactor is crucial for anything that is greater in scale.

By using TacoTaco, a link connecting the controller and the interactor can be created. Through this link, the interactor has access to all the information that the controller has: the arguments, the model of the UI state, and the current state of the UI. Unlike the controller, which can only read the states and render accordingly, the interactor has the ability to change the state. Through the interactor the developer can control what they want the controller to show.

With these functionalities, the interactor acts as the data gateway for the controller. This gateway, along with the process of rendering the UI from the data sources, is depicted in figure 4. All data that is displayed in the controller, must pass through the interactor. The controller only displays the states, thus all these data must also be transformed to the respective state that can be displayed. This transformation happens inside the interactor. When the data is ready, the interactor informs the states that the data of the states have changed, by calling the method updateModel(). Through this method, payloads can also be provided to the controller to handle the state change more easily.

Vice versa, the interaction on the UI, which reported by the controller, is handled by the interactor as well. The interactor is notified about these interactions through a special primitive: Command, which is passed through the sendcommand() method on the controller. The commands can include information about the interaction itself: the type of the interaction, the data that comes with the interaction and the hint of which the interaction can be handled. Upon receiving these commands, the interactor can choose what to do with them. This can be achieved by overriding the abstract method handleCommand(). When the type of the command is determined, actions such as data fetching, data storing, and state manipulation can be performed safely.

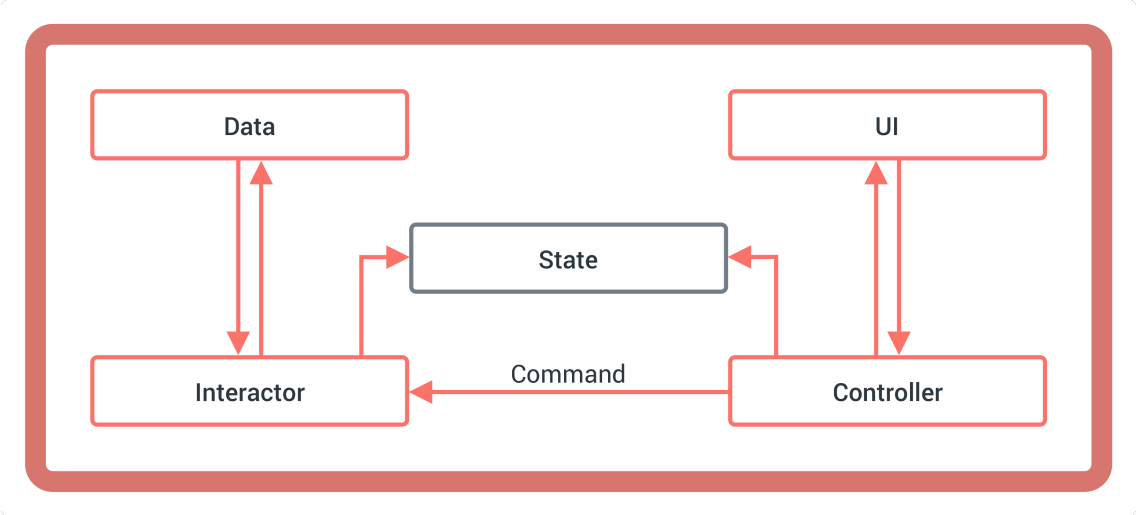

Figure 4: Taco Architecture and data flow

Navigation to another screen can also be a result of these commands. Though the controller is the primitive that handles the navigation, the interactor is the one who triggers them. Whenever the navigate() method of the interactor is triggered, in the internal of the interactor, the transition is passed silently to the controller which is attached to the interactor. hanldeTransition is then called to perform the pending transition.

The link to the controller also provides the Interactor with callbacks from the controller lifecycle. As such, many controller lifecycle events can be used by the interactor to manipulate the state. These events include `onAttach`, `onDetach`, `onForeground` and `onBackground`, as depicted in figure 5. The `onAttach` event is especially important since it is related to the `onSavedInstanceState()` method, and therefore, can tell the interactor if it is newly created, or have been brought back to life through the saved states. Although not as important, the other three lifecycle callbacks are suitable to perform data initialization, destruction and local state change.

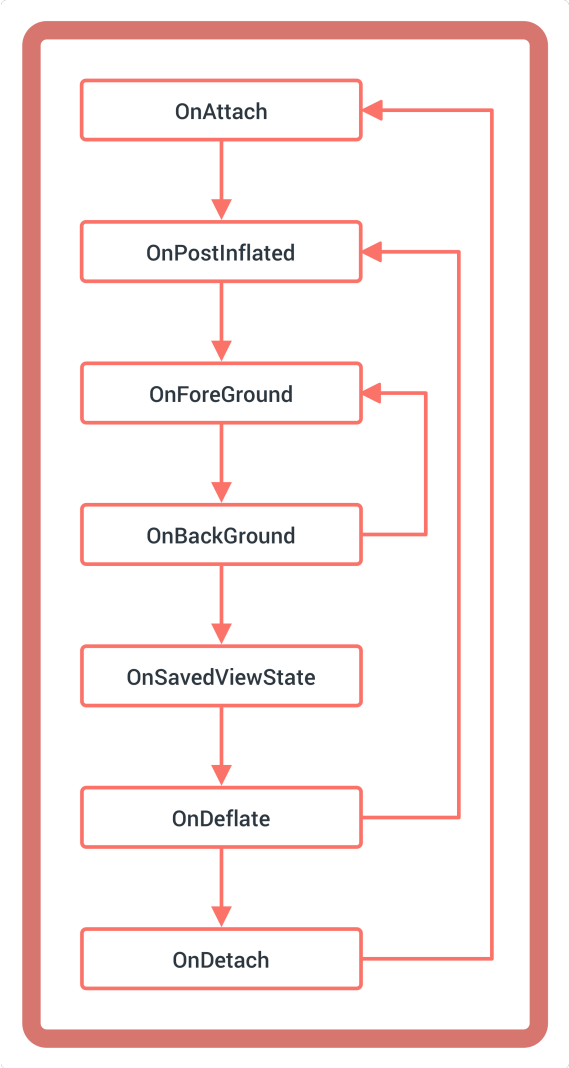

Figure 5: Lifecycle statge of TacoTaco Controller

The Interactor handles all the application's critical responsibility: data loading, data to state transformation, state uploading, interaction, lifecycle reaction and navigation. This is the reason why the interactor is considered the main business logic of a TacoTaco component, while the Controller is more oriented toward the UI.

2.3.4 Helper components

Apart from the two primitives that are used to construct Android applications, TacoTaco also includes two other components to help with day to day functionalities: Renderer and ControllerAnalytics. These components make the rendering work and the tracking of impression and interaction less menial tasks.

While the controller can handle the rendering logic base on current states, this can contribute to controller bloating, i.e. too many tasks being handle inside the controller. To prevent controller bloating from happening, a Renderer can instead handle the rendering functionality of the controller. Extending the Renderer abstract class, a single render can be attached to a controller. This grants the renderer knowledge of the controller states and thus can calculate the difference between the old states and new states. The controller, thus, has to implement the more basic displaying method and animation. Whenever there is a state change, the renderer can do the calculation and call the respective controller method to display the UI in a correct manner. This acts as separations of concerns, and lighten the load of the controller.

The ControllerAnalytics is created with the same goal in mind. As analytics - the tracking of user behaviour in an application - is a vital part to the survival of an application, TacoTaco's authors decided that it did not make sense to have controller bloated with the implementation of these. Instead, a ControllerAnalytics can be listed as a dependency of a controller. Like other dependencies, the ControllerAnalytics component has the access of the application display states, as well as the lifecycle methods and the commands sent by the interactor. With this information, a user impression and interaction with a view can be recorded easily, without over-complicating the controller logic.

Although serving mission-critical functionality, the renderer and ControllerAnalytics are both non-compulsory components of TacoTaco. These functionalities, albeit important, can be implemented within the controller, provided that the complexity of the controller is not the limiting factor.

2.3.5 Data flow within TacoTaco

Controller and Interactor, along with their helpers, Renderers and ControllerAnalytics, serve as the enforcer of the first three principles of TacoTaco: SingleActivity application, Command oriented interaction and finally, state-based rendering. However, the last principle of TacoTaco, data communication via Clean Architecture, needs more components to function properly.

According to Robert C. Martin, a data model should not have any dependencies [23].

Data should only be served as dependencies of their use cases, which are represented in TacoTaco as a UseCase class. These Usecases class can save and load these data into a data model - the primitive model which is given to the UseCase by the data source. Also within the UseCase, this data model can be transformed into the domain model to use in the UI component, or transform the domain model given by the UI component to the data model to be uploaded or saved to the data source. These Usecases once again can only have data as their dependencies and serve as the dependencies of the UI component, which is represented in TacoTaco by the interactor.

This method of accessing the data can be represented by one straight dependencies line for each data model: The UI is dependent on the Usecase for methods to load Data, and the UseCase is dependent on the data source to provide it. Using this data architecture lowers the complexity of the dependencies graph, making the code base much easier to understand as well as lowering the compile time when dependencies injection is used.

The dependencies graph can be hard to manage, as more and more controller, interactor, and data sources are used. As such, a method of managing these dependencies in introduced into TacoTaco: Dagger Dependencies Injection Framework.

Everything that acts as a dependency in TacoTaco can be provided and injected by Dagger. UseCases can be injected with its data sources, Interactor can be injected with different use cases. Likewise, Interactor, ControllerAnalytics, and Renderer can be overridden inside controller by injection. Controllers are the place where all these injections happen upon compilation. To achieve this auto dependencies graph, developers must override the method inject of the abstract Controller class, providing with the injection of controller all the instance variables that the controller dependencies graph may need.

2.4 Brief History of OCR

2.4.1 History of OCR as a field of science

Although the research for a solution for the OCR problem had begun earlier, only by the 1870s was the first OCR related invention - the retina scanner - designed by an American Inventor by the name of Charles R. Carey [24]. In 1914, Octophone was invented by

Edmund Fournier d'Albe to serve as the character tonal reader for visually-impaired people [1]. This period serves as a foundation for the OCR technologies that were to come later.

By 1930s, companies and enterprises were getting more interested in the results of OCR. IBM acquired the patent for "Statistical Machine" by Emanuel Goldberg [2]. This machine can read character and convert them into the widely used telegraph code. In 1938, "Microfilm Rapid Selector" was invented by a Vannevar Bush, and reportedly much faster than the Statistical Machine. Another similar machine that converts text to Morse code, the Gismo, was built in 1951 by American cryptoanalyst David H. Shepard [25].

The period between 1954 and 1975 was characterized by the increasingly consumer-oriented invention, digitalization of OCR results and the first application in scanning. On the consumer side, the first Optacon was invented in 1961, providing reading aids for the blind. The device was portable, marking the era of widely distributed OCR. On the application side, IBM 1287 was the first scanner to be able to read handwriting. To facilitate the detection of character, a font type was also invented in this period. This was quickly deprecated by the invention of omni-font OCR by Ray Kurzweil [26].

By 1975, OCR technologies were matured and stable enough to be used in official businesses. Credits card receipts readers, price tags scanners, and passport checkers were invented and used widely. Companies were also created to solely research, support and invent for OCR. ABBYY, Kurzweil Computer Products, and Caere Corporation still exist today, under the same or different names [5].

From 2000 to now, is the post-modern era of OCR. The technology is now used worldwide, in one form or another. Google Translate, an application that has more than half a billion downloads [6], uses OCR extensively to detect text and translate them on the fly. As another example, Adobe Reader, a popular Portable Document Format (PDF) reader, includes OCR support for any file it can open. During this period Tesseract was open sourced. [27]

2.4.2 History of Tesseract

Although only recently open sourced, Tesseract development started as early as 1984, in HP Labs, Bristol. Initiated as a PhD project of R. W Smith, Tesseract appeared suddenly at UNLV Annual Test of OCR Accuracy in 1995, and took the 3rd place in the test. However, no more development on the engine was conducted until it was open sourced in 2005. [27]

In 2006, Google announced its support and adoption for the engine [28]. Since then, multiple new versions of Tesseract have been released: Version 2.0 increased the accuracy and support of English language, 3.0 introduced support for non-Latin languages, and 4.0 introduced neural network, helping to further the accuracy of the framework. Currently, Tesseract supports more than 100 languages, including left-to-right languages such as Arabic and Hebrew, ideographic languages such as Japanese and Chinese, and non-Latin languages such as Greek.

2.5 How Tesseract works

The first step of the OCR process is to detect the layout of the paper. The text can be within any of the columns, rows, boxes appear in the paper. These are separated into different smaller parts in order for the machine to grasp the text more easily. [27; 29]

After receiving the layout on the first step, Tesseract puts outlines on the lines of the text. This process is called connected component analysis. This process is very computationally intensive but gives Tesseract major advantages when performing on non-black-on-white texts. The connected components are then put into different nestings, defining in Tesseract as blobs. In layman's term, a blob is a unit of work that contains a small part of the input image, portrayed by the outlines resulted by the connected component analysis. [29]

As demonstrated in figure 6, the next step in the process involves organizing these blobs into lines of text. This is performed using the line finding algorithm, which helps to detect skewed text without losing image quality by fitting four baselines on to the blobs, determining the height of each character.

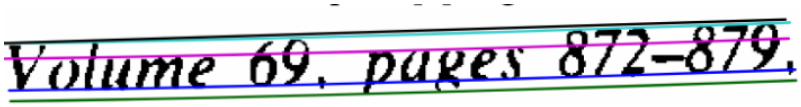

Figure 6: Line fitting a skewed text line [27]

The lines of text that are returned from line finding are now in good enough shape to be broken into words. Depending on the character spacing of the font, text is divided into two main categories: Fixed-width texts and proportional texts. Fixed-width texts are usually based on monospaced fonts, while proportional texts have different character size. The differences between these two type can be seen in figure7. Evidently, fixed-width texts are easier to work with and can be detected using character chopping and space comparison. On the other hand, proportional texts have to undergo a process called fuzzy spacing, to determine the actual wording of the text.

Finally, words are feed into the word recognizing algorithm in 2 passes. The first pass tries to recognize all the words, and whichever word passes the test will be kept as training data. These training data is used to determine the text further down the page, as well as in the second pass. In the second pass, based on the training data, words which were previously hard to recognize can be detected with more ease. After the two passes, the word should be recognized and presented as outputs of the program. [27; 29]

Figure 7: Different between fix-width text (monospace) and proportional text

2.6 OCR training

When carefully examining the last step of the Tesseract recognition process, it can be seen that there is another output of the application besides the result: training data. It can also be concluded that by modifying and enhancing this training data and plugging it in before the second pass of the process, the accuracy of the engine can be improved immensely. In fact, as of currently, the training data is very much an integral part of Tesseract; the

engine will not work correctly without a good set of training data. [27]

The process of obtaining the data through the last steps of the process, however, is not an optimal one. Therefore, efforts have been made to modify Tesseract, creating tooling inside it where training data can be obtained without going through the recognition process over and over again. [30]

The training procedure in version 3.0, even with the tooling, was a lengthy procedure. It involved manually creating a training data set, including images, text, and a special 'box' file - containing all the characters that can be recognized in this data set. Font properties files also need to be prepared beforehand. This file consists of all the fonts and styles used in the output if the font is determined by Tesseract. After obtaining the data set and font properties, multiple different programs are run to get more necessary files: the `unicharset_extractor` to extract charset from box file, the `shapeclustering` to obtain the master shape file, and the `mftraining` outputs the shape prototypes. These files are to be used, with optional dictionaries, in the program `combine-tessdata`, to output the actual tessdata to be used with Tesseract. [31]

In the more recent version 3.5, the whole procedure is simplified into a script called `tesstrain.sh`. The minimum input the script requires is only the training text file. Unicharset, number text file, and frequencies of characters can also be provided for more accurate results. [30]

The 4.0 version training is entirely different from the previous versions, because of its usage of the Long-term-short-memories neutral network. The steps are necessary to generate the basic training data for version 4.0, which is not much different from that of version 3.5. However, the input then is fed into `lstmtraining`, a multipurpose tool that helps the training process. Trained models are received as an output, and it has much higher accuracy than the previous versions trained data. [32]

2.7 Tesseract on JVM and Android

Tesseract, like every other application, cannot properly run without being compiled on each specific platform. Understanding that compilation is a complicated process, the

Tesseract community provided the application in pre-compiled forms, for Mac, Windows, and Linux. Pre-compiled Tesseract comes in two parts, the command line applications that are needed to run Tesseract as well as to train data, and the libtesseract, which exposes the actual Tesseract API, to be used by the command line application, the trainer, as well as other third-party applications. [33; 34]

Java, however, is designed to run cross-platform and cross-architecture. Underneath, Java runs on Java Virtual Machine (JVM), which is a virtual machine that translates Java Byte Code into platform-specific codes, and thus only allows binaries specific to JVM to run. This means that pre-compiled versions of Tesseract will not work in Java. [35]

To bypass these limitations, Java Native Interface (JNI) framework was created. This framework maps platform specific primitives to and from JVM data types, thus serving as a bidirectional interface, allowing Java to call native code and vice versa [36]. By using JNI, libtesseract methods are exposed, enabling `TessBaseAPI` to be called within the JVM platform. This is the core concept of libraries such as Tess4j[13] and Javacpp-Tesseract-Preset[14], and they work well with the compiled binaries of the native platform.

However, because of the lack of compiled binaries on the phone that the application is installed on, this approach does not scale to Android development. As such, Android provides Android Native Development Kit (NDK) as a way to include binaries in the application installation process. NDK, using Cmake in the background, invokes Android `makefile` when the compilation process starts and produces binaries targeting common platforms [37]. By using this technique, tess-two successfully ported Tesseract onto Android.

Tess-two, like other Tesseract JNI implementation, exposes the `TessBaseAPI`. To detect a text image using Tess-two, a `TessBaseAPI` instance needs to be created. This instance then needs to be initialized with the correct languages and be provided with the respective trained data. Finally, an image should be provided as input. This enables the `@getText()` method to work, which returns the result of the OCR. [38]

[13] http://tess4j.sourceforge.net/
[14] https://github.com/bytedeco/javacpp-presets/tree/master/tesseract

3 DeTexT Implementation

DeTexT is an Android based OCR application, with the main goal of capturing textual object through the camera, into a copyable, machine-readable text. It is implemented using TacoTaco as the key architecture and demonstrates the usability of the framework.

3.1 Overview of the application

3.1.1 Package system

Understanding the importance of having a good code base foundation, the first step in Detext development is to determine a meaningful package structure. By laying down the basic structure and its rules, components can be effortlessly divided into the proper package based on their functionality, which increases the modularity in the organization. To accomplish this, the Detext application was first divided into the data package, domain package, and presentation (view) package, each signifying a respective layer of the Clean Architecture.

The first layer of the Clean Architecture is the data layer where a `data` package is created, containing all data related components. A data layer consists of two sub-packages: `model`, which contains the data entity that data sources output, and `sources`, which represents the interface to actual data-sources, such as DAO, network interface, and sensors.

The second application package is the `domain`, characterizing the domain layer in the Clean Architecture model. This is where the business logic of the application is located. In this package, similar to the `data` packages, sub-packages exist to help further the organization of the code base. The `interactor` sub-package, like its name suggests, is occupied by the TacoTaco Interactor. Likewise, the `model` sub-package contains the domain model of the data. Use cases for these data, i.e. the interface between the interactor and the data sources, are called repositories and they exist within the `repository` package. Finally, a `delegate` package consists of delegate components, whose interfaces

serve business functions without attachment to a data source.

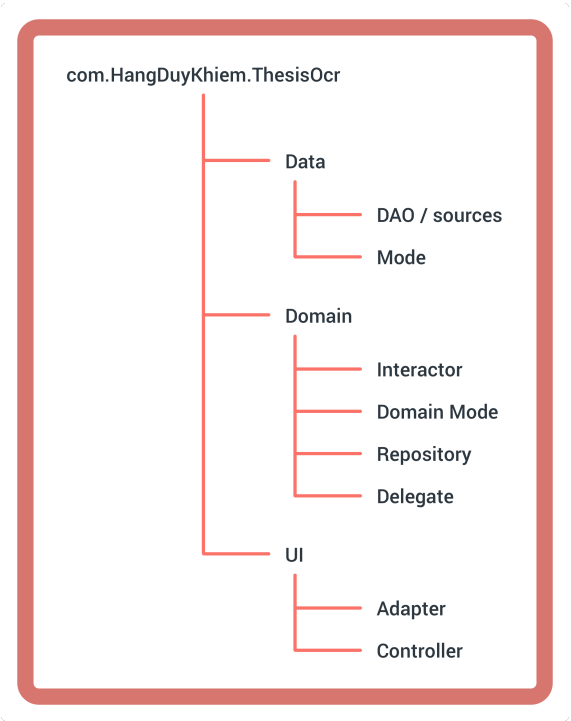

Figure 8: Detext package structure

The final major application package is made up of application UI components. This package is simpler than the previous packages, and it contains TacoTaco controllers, the main activity associated with it, and the adapter which RecyclerView inside controllers can use to build in-app lists.

The entry point of the application, in this case, ThesisApp class, is the only file that exists in the root application package. There is also another minor package in the main application which contains miscellaneous components, function, interface, and utilities that help with building the application. This package structure is exhibited in figure 8.

3.1.2 Dependencies Injection system

The first step in the setup of DI is defining which dependencies are going to need injections. This is achieved by going through the TacoTaco Architecture principles. Data sources interface, Repository, Interactor, Delegate, are the main candidates. Other than that, commonly used objects such as the Context, the main Activity, and helpers should also be included. These dependencies can then be provided by the Dagger modules,

which are located in the sub-packages module, or by the @Inject annotation.

Next, it is vital to determine where the dependencies can be injected. In dagger, Component serves as the injector of dependencies, and as such, can help to separate the responsibility level. At the root of the dependencies graph is the main Application injector, which can inject app-wide dependencies to its leaves, in other word the second layer. The second layer contains Activity injector, which contains all the activity-wide dependencies that the last layer will need. The last layer represents the UI layer, which consists of the controllers. The Controller Dagger Component responsibility is to gather all the dependencies from its parent layer, plus the dependencies used in controller business logic, and inject them to each controller.

Lastly, the scope of these dependencies is determined by using annotation `@Scope`. This scope determines if a dependency instance provided can be used in the same component, or if a new instance is needed. For example, a Singleton scoped dependency exists throughout the application's lifetime and is used by every component that requested the dependency. Controller scoped dependency is specific for the controller requested, and as such is a different instance in another controller.

3.2 DeTexT Implementation details

3.2.1 Entry and UI initialization

The first and foremost step of building any application is to provide its entry point. For an Android application, this entry point is defined in the manifest and can be either an activity or an application class. For this project, because of the prior decision on DI structure, and to initialize application wide dependencies, ThesisApp Application class was used.

As specified by TacoTaco principle, a single Activity is used. This activity is connected to the main Application and it is the entry point to the UI. This can be achieved by declaring only one activity in the application in the manifest, as demonstrated in listing 3.

```
1 <application
2     android:name=".ThesisApp"
3     android:label="@string/app_name">
```

```
4         <activity android:name=".view.MainActivity">
5           <intent-filter>
6             <action android:name="android.intent.action.MAIN" />
7             <category android:name="android.intent.category.LAUNCHER" />
8           </intent-filter>
9         </activity>
10  </application>
```

Listing 3: Declaring the MainActivity as the first activity of the application

Unlike other applications where the layout XML of the activity is inflated, in this activity, the RootController of the application is installed. The RootController is the root of the controller graph and acts as the main UI renderer of the whole application. Because the RootController has no arguments, and no state to render, special generic object `NoArgs` and `NoModel` are supplied.

When RootController is attached, the lifecycle method `onAttached()` is called in the interactor. Since RootController does not have any UI itself but instead it relies on the rendering of its children, the RootIneractor immediately calls the navigate method using the transition to SplashController, as shown in listing 4. This navigation is handled by the RootController, and SplashController then is attached to the main FrameLayout of the UI.

```
1  override fun onAttach(restored: Boolean) {
2      if (!restored) {
3          navigate(ToSplashControllerTransition)
4      }
5  }
```

Listing 4: RootController navigates immediately to SplashController when Attached

Apart from the SplashScreen, however, this Controller handles many more Transitions which are used to display the dialog controllers and to navigate to Tesseract results screen. These transitions are basically backstack manipulation logics, where a new backstack including the controller is set to the ViewGroup to be inflated. This navigation logic is highlighted in listings 5 and 6

```
1      fun pushChild(
2          controller: BaseController<*, *>,
3          backstackId: Int,
4          animation: TransitionAnimation? = null
5      ) {
6          val oldBackstack = getBackstack(backstackId)
7          if (oldBackstack.lastOrNull()?.tag != controller.tag) {
```

```
8               setBackstack(backstackId, oldBackstack + controller,
                    animation)
9           }
10      }
```

Listing 5: This method pushes controller on to a back stack and display it. There is also a `popChild()` method which has the opposite logic.

```
1           when (transition) {
2               ToSplashControllerTransition -> {
3                   val controller = SplashController()
4                   pushChild(controller, R.id.flMainContainer)
5               }
6               ...
7           }
```

Listing 6: Example of transition handling logic. This will display the SplashController.

3.2.2 Android Permission

Handling Permission in Android application is a particularly complex process. This is because of the introduction of permission features throughout Android version upgrade, which, while provided more security, created compatibility issues between versions. This is why the permissions of Detext are handled up front in the splash screen of the application.

Whenever`onAttached` is called, SplashInteractor immediately checks for permissions that the application needs. This is done through the PermissionDelegate component, which provides the needed callback for permission handling. If there is no permission, SplashInteractor updates the model to force the controller to display a dialog requesting them. Upon permissions granted, the interactor navigates to the main screen of the application.

3.2.3 Application main screens

The MainTabController, just as SplashController, replaces the later in the main FrameLayout and inflates its layout file in this location. Much like the RootController, MainTabController serves a FrameLayout in which multiple different child Controllers can be inflated. The only distinction between them is the BottomNavigationView, which provides methods to trigger the children inflation.

Upon being attached to the UI of the application, the Interactor navigates to inflate the first child, ScanController, to the layout. When one of the three navigational buttons is pressed, the respective Controller then replaces the ScanController, through manipulating the backstack of the FrameLayout, because of the tabbed navigation scheme.

This backstack manipulation logic keeps the index of the controller. If a new index is requested, a new controller instance will be created and displayed. On the other hand, the old instance will be reused and displayed in the case of an old index request. This is demonstrated in listing 7.

```
1     private fun goToTab(tag: String, createControllerAction: () ->
          Controller<*, *>, recreate: Boolean = false) {
2         val backstack = getBackstack(R.id.flTabsContainer).toMutableList()
3         if (recreate) {
4             backstack.removeAll { it.tag == tag }
5         }
6         val index = backstack.indexOfFirst { it.tag == tag }
7         if (index == -1) {
8             val controller = createControllerAction()
9             backstack.add(controller)
10        } else {
11            backstack.add(backstack.removeAt(index))
12        }
13        setBackstack(R.id.flTabsContainer, backstack,
              FadeInFadeOutAnimation())
14    }
```

Listing 7: Tab transition handling

ScanController is actually the first screen of the application that greets the user. The main goal of this controller is to get the URI of the image intended for the OCR process. As such, two way to do this is provided as buttons of the view: pick from the OS gallery, or to capture from the hardware camera. These buttons, when pressed, trigger commands that invoke the functionalities of the respective delegate components: FilePickerDelegate and CameraDelegate.

FilePickerDelegates and CameraDelegate function similarly. In FilePickerDelegates, the file picking intent is sent to the OS by the activity, which is injected in as its dependency. The activity dependencies also register the `onActivityResult` callback, which receives the file name and passes it along to the ScanController through another callback. This functionality of FilePicker is demonstrated in listings 8 and 9. CameraDelegate relies on

the same set of callbacks, but instead of receiving the file name, it needs to generate and provide these naming so that the camera can put the data of the captured image into the file name.

```
val intent = Intent()
intent.type = "image/*"
intent.action = Intent.ACTION_GET_CONTENT
activity.startActivityForResult(Intent.createChooser(intent, "Select
    Picture"), REQUEST_IMAGE_FILE)
```

Listing 8: Registering activity callback to pass image data

```
activity.registerCallbacks(object : ActivityCallbacks() {
    override fun onActivityResult(requestCode: Int, resultCode: Int, data:
        Intent?) {
        if (requestCode == REQUEST_IMAGE_FILE) {
            val uri = data?.data
            if (uri != null) {
                onImagePickAction(uri)
            }
        }
    }
```

Listing 9: Registering activity callback to pass image data

3.2.4 Implementation of Tesseract

Tesseract in Detext is implemented through Tess-two. In Tess-two, a JNI interface is provided to create and interact with TessBaseAPI, which provides the results of the OCR process. A delegate interface is created in DeText, to access these methods in a unified way.

Initialization of TessBaseAPI is a computationally intensive procedure. It is the first method of the TesseDelegate and it is called in the ResultsInteractor in another thread in order not to cause lag or pause on the UI. This is done by wrapping the whole initialization process in a RxJava Completable, which is subscribed on the IO thread and observed on the UI thread. The initialization needs the language strings that OCR uses, and the path to the `tessdata` folder, where the Tesseract training data is located. After receiving the strings, the process checks if the required training data exists under the path, and throws if they are not, or finishes the initialization if they are.

When the initialization process of TessBaseAPI is completed, TesseDelegate enables the method `getText()`. This method receives URL as input and opens a data InputStream on to the content associated. The InputStream then gets decoded into bitmap, using a BitmapFactory. Then the image is scaled so that its maximum height and width is under 1000 pixels, as shown in figure 10. This is done to get faster results from Tesseract. After the processing, the bitmap image can be set as the target image by calling TessBaseAPI `setImage()` method. The result of the OCR process can then be obtained by calling the `getUTF8Text()` method.

```
1  private fun getBitmap(uri: Uri): Bitmap {
2      val input = context.contentResolver.openInputStream(uri)
3      val result = BitmapFactory.decodeStream(input)
4      input?.close()
5      val resizedBitmap: Bitmap
6      if (result.height > result.width && result.height > 1000) {
7          resizedBitmap = Bitmap.createScaledBitmap(
8              result, (result.width.toFloat() / result.height *
                   1000).toInt(), 1000, false
9          )
10         result.recycle()
11     } else if (result.width > result.height && result.width > 1000) {
12         resizedBitmap = Bitmap.createScaledBitmap(
13             result, 1000, (result.height.toFloat() / result.width *
                   1000).toInt(), false
14         )
15         result.recycle()
16     } else {
17         resizedBitmap = result
18     }
19     return resizedBitmap
20 }
```

Listing 10: getBitmap method to resize and output bitmap from URI.

Image processing, as well as detecting the text on the image, are very expensive computer procedures. As such, similar to the initialization process, the `getText()` method of TesseDelegate wraps the whole process inside an RxJava Observable, which emits the results and can be subscribed to obtain the OCR. This can be seen in listing 11

```
1  fun getText(uri: Uri): Observable<String> {
2      return Observable.fromCallable {
3          val bitmap = getBitmap(uri)
4          tessBaseAPI.setImage(bitmap)
5          return@fromCallable tessBaseAPI.utF8Text
6      }.compose(applySchedulers())
7  }
```

Listing 11: OCR process being wrapped in an Observable.

The two public methods of TesseDelegate, `getText()` and `initLanguages()`, thus, are all RxJava observable, which makes chaining of them possible. This is exactly the mechanism of ResultsInteractor. When the Completable from `initLanguages()` method signifies completion, the URI received from ScanController is immediately put as an argument into `getText()` method. The results of the OCR is then emitted, and is immediately handled by updating the model to reflect the result in the UI. By doing this, all I/O and blocking operation is done on the I/O thread, resulting in smooth UI. This process is demonstrated in listing 12. In this listing, it can also be seen that, before the whole process begins, the model is updated to display a loading screen, and in the case of failure, UI will display the failure state based on the new model.

```
1  private fun getResult() {
2      updateModel(model.copy(loadingState = WorkState.InProgress))
3      val uri = Uri.parse(model.uriString) ?: return
4      disposables.add(
5          tesseDelegate.initLanguages(currentLanguages.joinToString("+"))
6              .andThen(tesseDelegate.getText(uri))
7              .subscribe(
8                  {
9                      updateModel(model.copy(loadingState =
                           WorkState.Complete, resultString = it))
10                     ocrResultRepository.saveOcrResult(uri = uri, result =
                           it)
11                 },
12                 { updateModel(model.copy(loadingState =
                       WorkState.Fail(it))) }
13             )
14     )
15 }
```

Listing 12: ResulInteractor observing the RxJava OCR process and update model accordingly.

The UI in front of the ResultInteractor is simple. When it is attached to the RootController, the image sent from ScanController is immediately put into the ImageView using glide. A ProgressBar is added on top of the view so that the loading state is correctly presented. If the model is updated with the results, its TextView is updated with the results string in the model, and shows the results to the users. On the UI, there is also the Copy button, where results can be copied and used elsewhere. This sends a command to Interactor, trigger the clipboard `newPlainText()` method, in order for the text to be available system-wide.

3.2.5 Results Storage

When the results are displayed, Detext can also store them for later review. To save and retrieve all the reviews from previous and current sessions, persistent data storage is needed. Room Persistent Library is currently recommended by Google as a method of persisting data, and thus it was used to build Detext results storage.

As a first step to the process, the data entity is declared. Since only the information about the results string and the image URI is enough to display the ResultController, the entity for Room is very simple, containing the two as columns. Another column is added, to mark the timestamp when the results were generated by Tesseract. The entity is shown in listing 13.

```
@Entity(tableName = "results")
data class OcrResultData(
    @PrimaryKey(autoGenerate = true) var \gls{ui}d: Int? = null,
    @ColumnInfo(name = "file_name") var uri: String,
    @ColumnInfo(name = "result") var result: String,
    @ColumnInfo(name = "timestamp") var timestamp: Date
)
```

Listing 13: OcrResultData entity with columns for name, result, and timestamp

A DAO interface is also created, reflecting the CRUD operations that can be done to the data. The delete and insert methods return nothing, while the get method is wrapped in a RxJava Single, which is provided by default by Room because the I/O operations to get the lists of OcrResultData can be expensive. These properties of the component are reflected in Listing 14.

```
@Dao
interface OcrResultDAO {
    @Query("SELECT * FROM results")
    fun getAllOcrResult(): Single<List<OcrResultData>>
    @Insert
    fun insertAll(vararg users: OcrResultData)
    @Delete
    fun deleteAll(vararg users: OcrResultData)
}
```

Listing 14: DAO interface to perform simple operation on the database.

Finally, the database wrapper is created, to put the entity and the DAO interface into use.

A type converter for the date format is used to ensure the storage of date is successful. A schema of this database, upon compilation, is automatically generated, for the purpose of later migration. The database instance then is provided by the AppModule - the application wide dagger module - and is used to inject the DAO into the use case.

The use case for the result is called OcrResultRepository, located in the Repository package. The file contains method exposing the DAO interface and transforming result to the domain model, returning the respective RxJava Single and Observable. The use case, once again, is injected into the HistoryInteractor, where the data is loaded and sent to HistoryController adapter for displaying. The HistoryController, when the state is updated including the results data, updates the RecyclerView adapter to show the result list.

3.2.6 Building and deploying Detext

Although the critical aspects of the application have been discussed, many minor implementations are left out to keep the scope of the thesis small. These features, however, can still be further analysed to verify the use case of Detext. This is achieved by building and running the application on emulator or devices. The source code of the application can be obtained through Github service, under the URL https://github.com/hangduykhiem/ThesisOCR/.

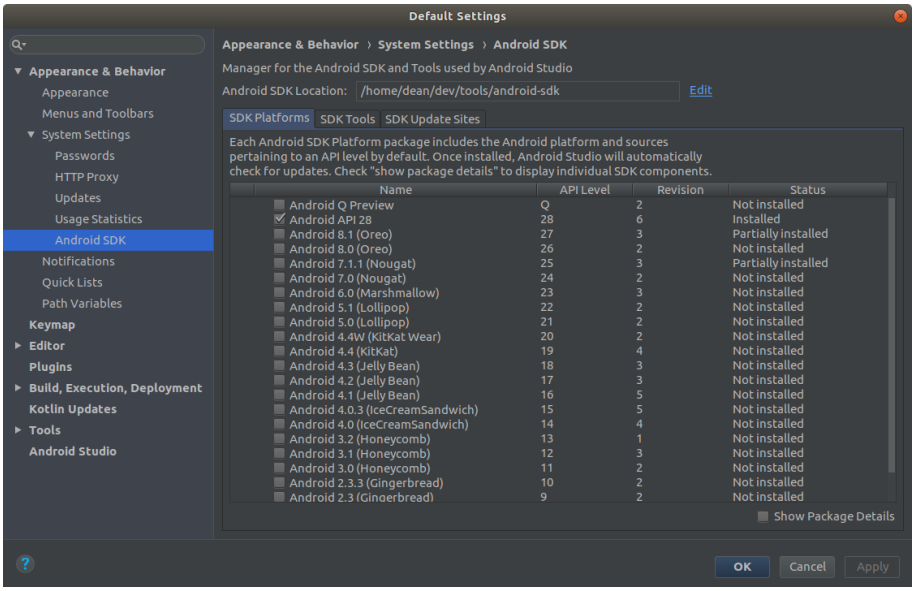

Figure 9: Android SDK can be downloaded with Android Studio SDK manager

The application can be built using Android Studio, the main Integrated Development En-

vironment (IDE) for Android development. This can be obtained and installed on any computer running Windows, Linux or Mac OSX. The installation guide and the binary, are provided by Google, through the Android Developer Website[15].

To compile the application, both Java Development Kit (JDK) and Android SDK must also be provided for Android Studio. The target versions that the project compiles again are JDK 1.8 and Android SDK for API 28. JDK can be downloaded and installed through Oracle[16] main distribution, or through OpenJDK[17], and the Android SDK can be set up with in the Android Studio SDK management tool. This tool is demonstrated in figure 9.

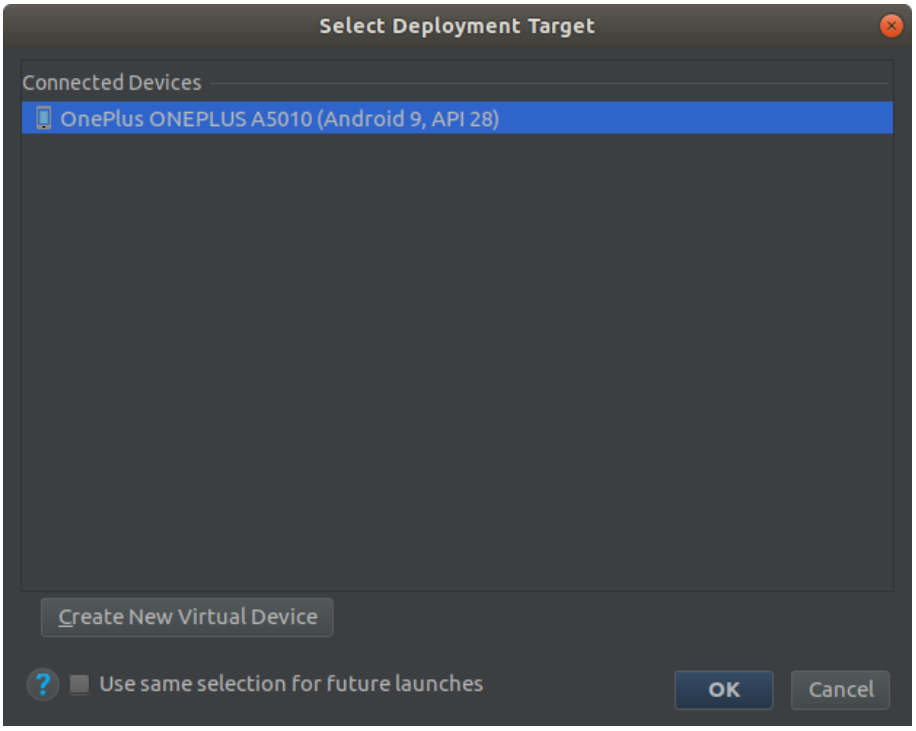

Figure 10: Android Studio target selection dialog. This dialog is triggered by the run command.

After all the prerequisite tools have been correctly set up, the project can be imported directly through the Import utility in Android Studio. This assists in the retrieval of dependencies and indexing the application file hierarchy. The run command is then available, which prompts a deployment target selection dialog. This is shown in figure 10. The application is installed and launched in the selected device.

If the application correctness is strictly required, more configurations to the development environment are necessary. First, the essential tools were installed on an Ubuntu

[15] https://developer.android.com/
[16] https://www.oracle.com/technetwork/java/javase/downloads/index.html
[17] https://openjdk.java.net/

18.04 machine. Next, the 8u202 version of JDK and revision 6 of the Android SDK was used to compile the application. Android Studio version was 3.1.4. Finally, the application was installed and tested a One Plus 5T device running OxygenOS build number A5010_42_190222.

4 Result and Discussion

4.1 On the result and performance of Tesseract

A screenshot of the application and the OCR result it gave is provided in figure 11. For a scan with an older font style, low resolution, and dirty overall, the text captured was definitely readable and understandable, disregarding a few mistakes caused by text skewing. However, when it comes to actual captured pictures and not black and white scan, Tesseract does not behave as well; occasionally outputs gibberish, meaningless text. Undoubtedly, more optimization can be done to DeText and Tesseract to improve this behaviour.

For better detection of text in pictures, image optimization may help. As many trials with the application have proven, the colourful background can confuse Detext and lower the quality of the OCR result. This can perhaps be mitigated by transforming the picture to black and white. Tesseract results also suffer when the contrast of the picture is low, and the text is blended into the background. By increasing the contrast of the image before input it into Detext, better results were observed. As such, it might be good to introduce contrast and black-and-white filter into the application.

The version of Tesseract running in Detext was 3.5, because of technical limitation existed already when the application was built. However, this limitation has been solved by the successful compilation of Tesseract 4.0 in Android. Because of the included neural network, text detection might be better than the current implementation. This is definitely another improvement that can be included in the next version of DeText.

The training data included in Detext was only for 4 languages. This is because there is no downloading mechanism where more languages could be added. Also, the training data is provided by the community, so the quality is high, but conceivably, the self-trained data may behave better with Detext use cases. Incidentally, when detecting for more than 2 languages at the same time, the performance of DeText suffered greatly.

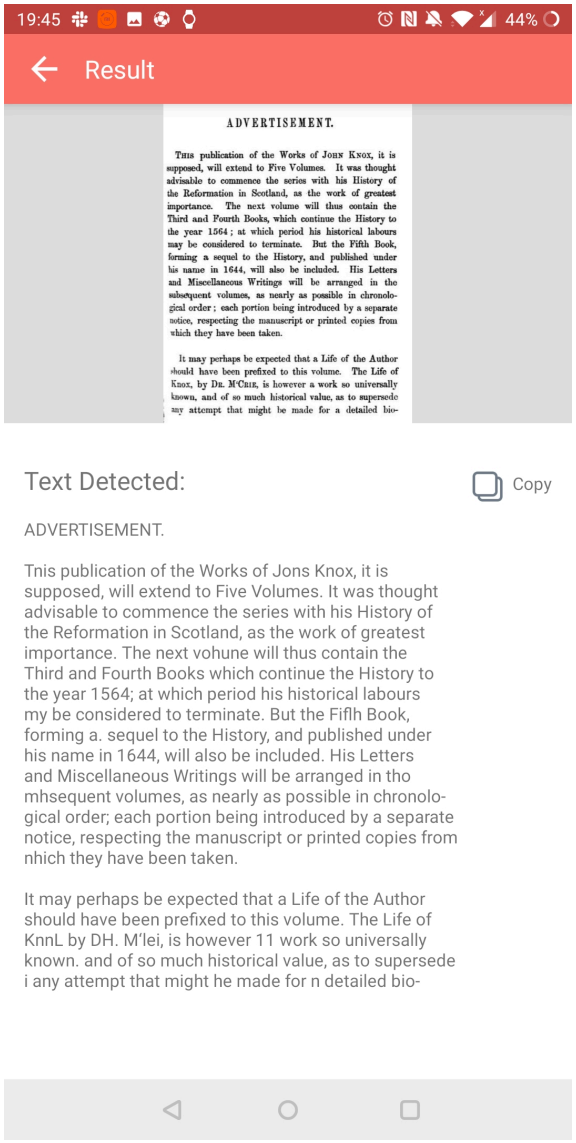

Figure 11: Detext results for a scalation

4.2 On TacoTaco as an application framework

Much of the TacoTaco application development process, including its major concepts such as Controller, Interactor, and their usage, have been demonstrated in the implementation of DeText. From this implementation and the actual working application, it can be concluded that TacoTaco is a suitable framework for this use case of OCR detection. This, along with the actual Wolt Android application, proves that TacoTaco is mature enough to be flexibly applied to other use cases.

The package structure of separating the data, domain and UI layers worked out decently for the application. However, as the scope of the application increases, the number of

files representing the interactor, controller and data sources also expands and can result in difficulties managing the code base. Further modularisation of the code base using feature packages is one possible solution for this.

It is also worth noting that the implementation of OCR was deemed not enough to exhibit one TacoTaco architectural principle: clean data access. In the base OCR process, there was not a need to communicate with any data source. The only exception to this was for the image file, which is provided by the OS. Thus, a use case of fetching previous histories was added to demonstrate the principle of Clean Architecture. Although the objective is achieved, this slowed down the development process of DeText to some extent.

In comparison with other application development frameworks based on MVP, MVC and MVVM, TacoTaco is quite different, state-centric. When something changes in the state, it will propagate the change to the UI. This is completely different from MVP and MVC framework, where the state of the application is vague, and the rendering is based on the presenter and controller respectively. In MVVM framework, the state of the view located in the viewModel, so while having a state, it is the data binding that updates the state instead of the interactor in TacoTaco. This state-centric principle makes TacoTaco applications much easier to debug, as the whole state tree can be inspected to see what went wrong with the application.

However, regarding the performance and code base organization of TacoTaco in comparison to other frameworks, not much can be deduced. It would only make sense to observe these metrics by building Detext using the different frameworks and draw conclusions from the finished application. There are ways to achieve this, through the use of a method called Architecture Tradeoff Analysis Method, which evaluates the criteria of these frameworks to see which one is the most suitable for building a specific application [39].

5 Conclusion

The purpose of this thesis was to implement an Android OCR application as a way to evaluate the capacity of the TacoTaco framework in a real-world scenario, as well as to assess the Tesseract OCR framework Android implementation. To achieve the goal, the DeText application was created, and its development process was carefully examined.

By taking a close look at TacoTaco in the context of Detext, conclusions on the framework accessibility were made. The principles of TacoTaco serves as a foundation upon which a sensible application can be built. Concepts such as Controller, Interactor, Clean Architecture, were familiar from previous and current architectures and worked well in the development of Detext. As such, TacoTaco is mature and functional enough to be implemented in a wider application area.

Despite not being included in the original research topic, findings on Tesseract OCR engine implementation, performance, and accuracy were also thoroughly discussed. It was concluded that the current implementation of Tesseract through in Detext is not straightforward, with tess-two acts as a JNI wrapper so that native Tesseract API can be called. However, it is achievable and seemed to work well with black-and-white scanned texts. In scenarios with less contrast and more colours, Tesseract performance suffers, but more optimization can be done.

Further studies on the implementation of Tesseract 4.0, additional download function to fetch training data straight from the repository, and the optimization of multiple language OCR are needed to enhance the current application, its behaviour, and performance. It is also worth making a deeper analysis at the accuracy versus performance of Tesseract under more stressful conditions, and compare the result to different OCR open source and commercial implementations, such as OCRopus[18], Google Vision[19], etc.

The performance and non-functional-requirement comparison between TacoTaco and other Android application frameworks were not analyzed in detail in this thesis. Thus,

[18] https://github.com/tmbdev/ocropy
[19] https://cloud.google.com/vision/

it would be interesting to conduct further studies on the implementation of the application in the respective frameworks, along with their impacts on the executions, as well as the workflow and the quality of the codebase.

Bibliography

1 Fouknierd'Albe EE. On a Type-reading Optophone. The Royal Society. 1914 7;90.

2 Emanuel G. Statistical machine, US1838389A. IBM; 1927.

3 Xerox MRP Family4215/MRP, 4219/MRP, 4220/MRP, 4230/MRPIntelligent Printer Data Stream (IPDS)Configuration and Reference Guide. Xerox Corporation; 1995.

4 Twenty Years of Innovation: HP LaserJet and Inkjet Printers 1984–2004. HP Inc; 2004.

5 The technology for converting books and documents into electronic files. ABBYY; 2019. Available from: https://www.abbyy.co.il/?categoryId=72050&itemId=168963.

6 Google translate application detail. Google LLC;. Available from: https: //play.google.com/store/apps/details?id=com.google.android.apps.translate [cited March 19, 2019].

7 Android Dagger, Android Room, Android Glide, Android Retrofit - Explore - Google Trends. Google LLC; 2019.

8 Razina E, Janzen DS. Effects of dependency injection on maintainability. In: Proceedings of the 11th IASTED International Conference on Software Engineering and Applications: Cambridge, MA; 2007. p. 7.

9 Dagger User's Guide. Google LLC;. Available from: https://google.github.io/dagger/users-guide [cited March 19, 2019].

10 Manage Your App's Memory. Google LLC;. Available from: https://developer.android.com/topic/performance/memory [cited February 17, 2019].

11 Room Persistent Library. Google LLC;. Available from: https://developer.android.com/topic/libraries/architecture/room [cited February 19, 2019].

12 Save data in a local database using Room. Google LLC;. Available from: https://developer.android.com/training/data-storage/room/ [cited February 19, 2019].

13 ReactiveX Introduction;. Available from: http://reactivex.io/intro.html [cited March 24, 2019].

14 ReactiveX documentation;. Available from: http://reactivex.io/documentation [cited March 24, 2019].

15 Glide Performance. Bump Technologies;. Available from: https://bumptech.github.io/glide/#performance [cited March 24, 2019].

16 Glide Documentation. Bump Technologies;. Available from: https://bumptech.github.io/glide/ [cited March 24, 2019].

17 Bäumer D, Gryczan G, Knoll R, Lilienthal C, Riehle D, Züllighoven H. Framework Development for Large Systems. Commun ACM. 1997 10;40:52–59.

18 Riehle D. Framework Design: A Role Modeling Approach. Softwaretechnik-Trends. 2000 01;20.

19 Overview of the Android Source Code. Google LLC;. Available from: https://source.android.com/setup/ [cited February 20, 2019].

20 Android Platform Architecture. Google LLC;. Available from: https://developer.android.com/guide/platform [cited March 26, 2019].

21 Sokolova K, Lemercier M. Android Passive MVC: Novel Architecture Model for the Android Application Development; 2013. .

22 Maharjan B. Puzzle game using Android MVVM Architecture. Metropolia Ammattikorkeakoulu; 2018.

23 Martin RC. The Clean Architecture; 2012. Available from: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html [cited March 19, 2019].

24 Ning L. An Implementation of OCR System Based on Skeleton Matching; 1993. .

25 Romero V, Toselli AH, Vidal E. Multimodal Interactive Handwritten Text Transcription. WORLD SCIENTIFIC; 2012. Available from: https://www.worldscientific.com/doi/abs/10.1142/8394.

26 Hauger JS. Reading Machines for the Blind: A Study of Federally Supported Technology Development and Innovation. Virginia Polytechnic Institute and State University; 1995. Available from: https://books.google.fi/books?id=BLy1IAAACAAJ.

27 Smith R. An Overview of the Tesseract OCR Engine. In: Ninth International Conference on Document Analysis and Recognition (ICDAR 2007). vol. 2; 2007. p. 629–633.

28 Vincent L. Announcing Tesseract OCR; 2006. Available from: http://googlecode.blogspot.com/2006/08/announcing-tesseract-ocr.html.

29 Smith R, Antonova D, Lee DS. Adapting the Tesseract Open Source OCR Engine for Multilingual OCR. In: MOCR '09: Proceedings of the International Workshop on Multilingual OCR; 2009. Available from: http://doi.acm.org/10/1145/1577802.1577804.

30 Training Tesseract 3.03–3.05. Google LLC;. Available from: https://github.com/tesseract-ocr/tesseract/wiki/Training-Tesseract-3.03%E2%80%933.05 [cited March 24, 2019].

31 Training Tesseract 3.00–3.02. Google LLC;. Available from: https://github.com/tesseract-ocr/tesseract/wiki/Training-Tesseract-3.00–3.02 [cited March 24, 2019].

32 Training Tesseract 3.00–3.02. Google LLC;. Available from: https://github.com/tesseract-ocr/tesseract/wiki/TrainingTesseract-4.00 [cited March 24, 2019].

33 Tesseract OCR. Google LLC;. Available from: https://github.com/tesseract-ocr/tesseract [cited March 24, 2019].

34 Tesseract APIExample. Google LLC;. Available from: https://github.com/tesseract-ocr/tesseract/wiki/APIExample [cited March 24, 2019].

35 Lindholm T. Java Virtual Machine Specification. Oracle Corporation; 2013.

36 Sarkar B. Invoking Assembly Language Programs from Java Blog. Oracle Corporation;. Available from: https://community.oracle.com/docs/DOC-983730 [cited March 24, 2019].

37 Introduction to NDK building. Google LLC;. Available from: https://developer.android.com/ndk/guides/build [cited March 24, 2019].

38 Theis R. Tess-two repository;. Available from: https://github.com/rmtheis/tess-two [cited March 24, 2019].

39 Kazman R, Klein M, Clements P. ATAM: Method for Architecture Evaluation. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University; 2000. CMU/SEI-2000-TR-004. Available from: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=5177.