Expertise
and insight
for the future

Giang Pham

Develop maintainable animated Android applications

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

24 April 2021

Author	Giang Pham
Title	Develop maintainable animated Android applications
Number of Pages	59 pages
Date	24 April 2021
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Mobile Solutions
Instructors	Kari Salo, Principal Lecturer

Animations are indispensable in Android applications due to their many benefits. They not only bring a quality look and feel to the software but also deliver useful messages to end-users. Even though the animations are visually pleasing to end users, the code that creates them sometimes is not pleasant to software engineers. This thesis aims to discuss how to develop highly animated Android applications and make the source code maintainable at the same time.

To make the discussion objective, an animation-centric Android application is taken as a reference. In this study, the most complicated animations of the software are analyzed and mimicked programmatically by using the Android SDK, the Kotlin language, and the theory of clean code. Essentially, software conventions are followed to enhance the program's comprehensiveness.

While those animations are successfully recreated as a result of this study, the project also leaves reusable animation components and a logically magnificent codebase.

When the project grows, new animation components can easily be built on top of existing components easily. This is achievable if the developer focuses on the code structure before writing the actual implementation.

Keywords	Android, mobile application, animation, transition, clean code, maintainable code

Contents

List of Abbreviations

UI	User interface. The visual display where interactions between humans and machines happen.
UX	User experience. The feeling of a person about using a specific product or service.
SDK	Software development kit. A set of tools for software development under the form of an installable package.
API	Application programming interface. A contract of software that requires other parts of that software or external software to respect.
XML	Extensible markup language. A self-descriptive markup language that is used to store and transport data. In Android development, it is used to define application resources.

1 Introduction

Animation is the transformation of a UI element property, such as its color, position, or size, over time (Hashimi et al., 2010). Adding animations properly could bring a polished look and a professional feeling to users. Even though Google gives Android developers multiple ways of creating beautiful animation for their applications, it is still challenging to write maintainable animation code (Android Developers, 2019). It is usual to encounter gigantic blocks of animation code in Android development. The following script was taken from a popular repository with more than 3500 stars on Github.

```
Animator
        anim = ViewAnimationUtils.createCircularReveal(view, cx, cy, 0, finalRadius);
anim.setDuration(durationMills);
anim.addListener(new AnimatorListenerAdapter() {
    @Override
    public void onAnimationEnd(Animator animation) {
        super.onAnimationEnd(animation);

        if (requestCode == null) {
            thisActivity.startActivity(intent);
        } else if (bundle == null) {
            thisActivity.startActivityForResult(intent, requestCode);
        } else {
            thisActivity.startActivityForResult(intent, requestCode, bundle);
        }

        thisActivity.overridePendingTransition(android.R.anim.fade_in,
android.R.anim.fade_out);

        triggerView.postDelayed(new Runnable() {
            @Override
            public void run() {

                Animator anim =
                        ViewAnimationUtils.createCircularReveal(view, cx, cy,
finalRadius, 0);
                anim.setDuration(durationMills);
                anim.addListener(new AnimatorListenerAdapter() {
                    @Override
                    public void onAnimationEnd(Animator animation) {
                        super.onAnimationEnd(animation);
                        try {
                            decorView.removeView(view);
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                });
                anim.start();

            }
        }, 1000);


    }
});
anim.start();
```

Script 1. An example of a long block of animation code (Hugo, 2017).

Despite being widely recognized by the open-source community, there still exists lengthy and incomprehensible animation code with multiple levels of indentation inside the project. The author also makes a few mistakes when writing the animation, but it will be discussed later. This happens when developers do not take into account the maintainability of their code. It leads to code reviewing difficulties within the team and a time-consuming test cycle due to the unreliable codebase, thus, delaying the release plan that means to deliver new experiences to end-consumer. This thesis studies different methods to solve that challenge.

This thesis focuses on creating utility code that could be used to make animation code more readable and maintainable. The implementation of this project highlights the demand for maintainable animation code in an Android application.

In order to benchmark and illustrate the practicality of the utility code, it is reasonable to reproduce some animations from a real-life world-class project such as Wolt. Wolt Enterprises Oy is a food delivery company based in Helsinki, Finland. The company provides its users with native mobile applications, which received several satisfied reviews about UI and UX on respective platform stores (Wolt Enterprises Oy, 2015a; Wolt Enterprises Oy, 2015b). A part of that success comes from the appropriate usage of animations.

2 Theoretical background

2.1 Animations in mobile applications

Animation is the transformation of a UI element property, such as its color, position, or size, over time (Hashimi et al., 2010). A modern mobile phone user experiences an unlimited number of visual effects on the device. Almost every interaction leads to UI changes. The user might miss tiny changes, whereas a huge amount of changes might lead to confusion. However, when animations are used reasonably, they serve meaningful purposes. (Mathis, 2016.)

Animations make the application self-explanatory, especially when the state of the application changes (Android Developers, 2019a). Figure 1 illustrates a circular reveal transition in the application Whim. The second screen gradually expands its circular bounds around the button W until it covers the whole space. The transition looks natural and connects the two screens over a short period. It creates visual continuity when the second screen appears. (Android Developers, 2019d.) This is just one of the infinite transition implementations that mobile users have seen.

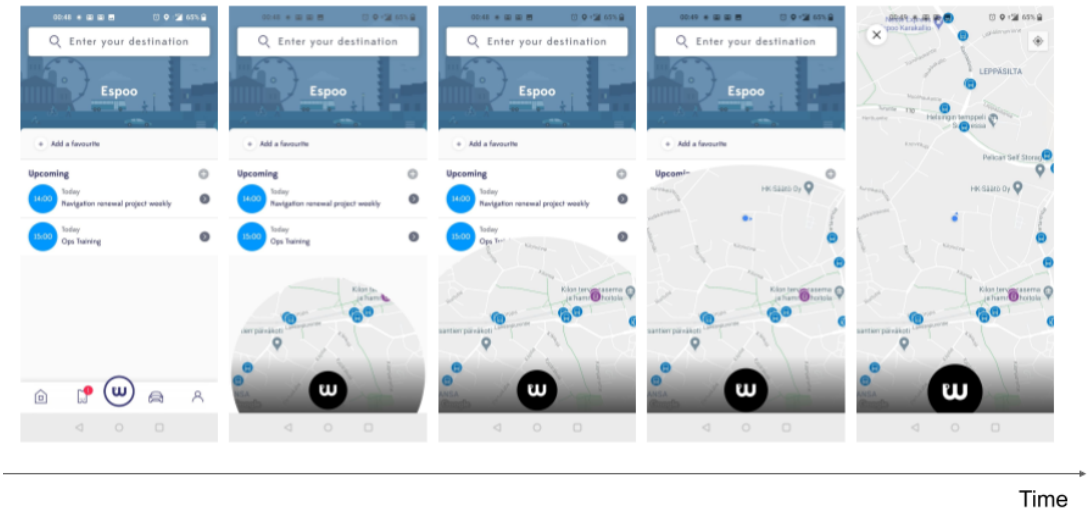


Figure 1. Timeline of the circular-reveal transition in the application Whim. Screenshot (MaaS Global, 2011)

Sometimes, animation is used to draw users' attention and direct them to the next application flow. In the material design system, this technique is called user education.

(Material design, 2020b.) Figure 2 shows the incoming call screen of a OnePlus device. Under the caller image, there is a circle button with a phone icon inside. It expands and shrinks periodically. At the same time, the red arrow and the green arrow repeatedly slide out of that button. This combination teaches the user to take the desired action by holding the circle button and sliding it towards one of the directions. In this case, while the red arrow indicates declining action, sliding the button towards the opposite way means accepting the call.

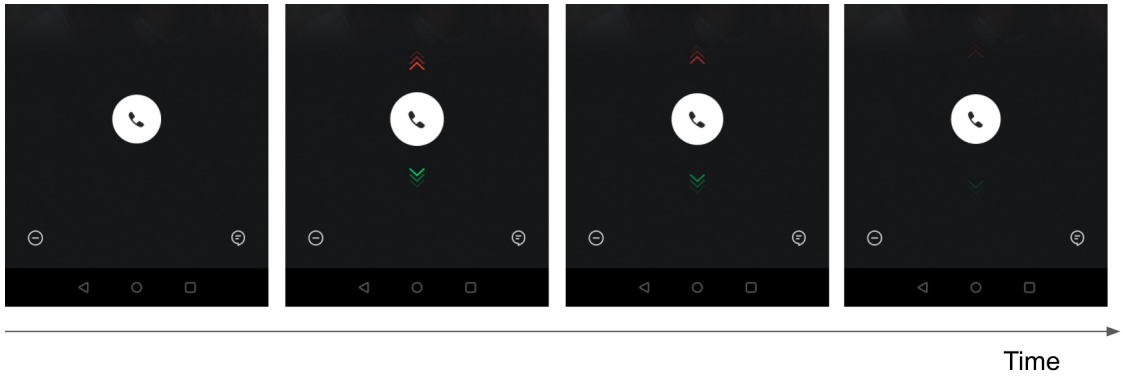


Figure 2. Timeline of the animation for an incoming call in a OnePlus 8 Pro smartphone.

Moreover, animations are added for other particular goals. Some designers leverage their morphing effect to make their applications look vivid and polished (Android Developers, 2019a). Others decide to include animated graphical UI for branding purposes (Andreas et al., 2019).

2.2 Animation components

2.2.1 Animated element

Animated element refers to a UI element whose attributes are changed continuously over a period to form an animation. Specifically, these attributes are size, opacity, color, rotation, position, and shape of the animated element (Kantola, 2017). They are visual targets of an animation.

2.2.2 Easing curve

Easing curve, or easing function, represents acceleration, the rate of change in something's speed. In the real world, an object does not always maintain a constant speed or instantaneously changes its velocity. Physically, it has to be changed gradually by acceleration. (Penner, 2002.) It is worth mentioning that velocity does not only refer to how fast a movement is. In motion design, easing curves also control how fast an attribute of an element changes its value. In other words, they are interpolation formulas for attributes. (Yuen, 2017.)

Figure 3 presents graphs of some common easing functions. While the horizontal axis of the graph denotes the timeline of an animation, the vertical axis indicates the value of a UI element's attribute at a specific time.

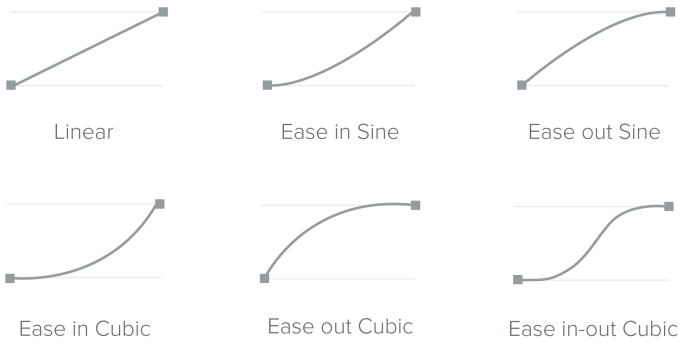


Figure 3. Graphs of common easing functions. Copied from PROTOIO Inc (2014).

The linear easing curve maintains the velocity at a constant value during an animation. That means the velocity abruptly switches from 0 (zero) to that value at the beginning of the animation and again instantaneously changes from that value to 0 (zero) at the end. As mentioned, it is physically impossible because the acceleration becomes infinite. Therefore, linear motion usually feels unnatural. (Penner, 2002.)

Numerous researchers highly recommend using the ease-in-out curve for animation because it brings the most natural feeling in most cases. According to them, the curve reveals how an object moves in the real world, accelerating from the static state and slowing down before stopping completely at the destination. (Dragicevic et al., 2011; Izdebski et al., 2016; Penner, 2002.)

However, ease-in-out is not the answer for every case, and linear easing is not always discouraged. For example, material design, a recommended design system for the Android platform, gives specific use cases for certain curves such as spring curve and overshooting curve (Material Design, 2020a). Occasionally, designers choose not to follow these recommendations for unusual situations. They attempt to customize their curves. One popular way to create these curves is using Bézier curves of degree 3, also called cubic Bézier equation:

$$B(t) = (1-t)^3 P0 + 3(1-t)^2 tP1 + 3(1-t)t^2 P2 + t^3 P3 \qquad 0 \leq t \leq 1 \qquad (1)$$

In formula (1), P_0, P_1, P_2, and P_3 are control points. Their x-coordinates must always be between 0 and 1. Mathematically, these are all the restrictions for the formula. However, in motion design, P_0 and P3 always stay respectively at (0, 0) and (1, 1). (Izdebski and Sawicki, 2016.) Figure 4 illustrates the graph of a cubic Bézier that has P_1 and P_2 at (0.48, 0.05) and (0.07, 0.87). By adjusting the position of P_1 and P_2, numerous curves are created and used not only in Android applications but also in consumer-facing applications in general.

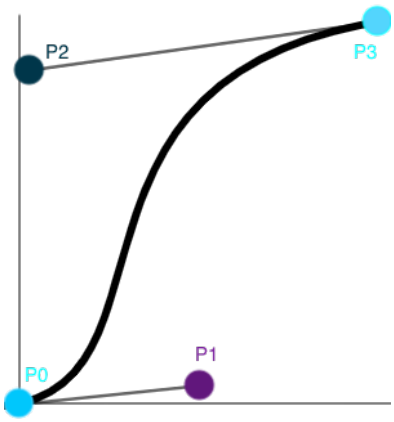


Figure 4. An example of a cubic Bézier graph

In general, the easing function is the most complicated yet significant component of animation. It carries the animation's vibe that effectively connects the motion of a UI element to users' subconsciousness and orients their feelings (Barclay, 2019).

2.2.3 Duration

Unless the motion depends on other factors, every animation takes time. Without it, easing functions become useless because intermediate steps of a transition are not rendered in any frame. To ensure the user experience, the timing of an animation must be reasonable. Microsoft advises making the animation neither too slow nor too fast. A slow animation decreases user productivity and becomes cumbersome when users are familiar with application flows. In contrast, an animation, which takes less than 50 milliseconds (ms), is not comprehensible and considered a jarring experience. (Microsoft, 2018.)

To obtain the full benefit of animations, finding the perfect duration is crucial. Even though it is not a strict rule, optimal animations usually take between 150ms and 350ms (Klimczak, 2013). In the Android design system, it often depends on the size of the animated area. For instance, a full-screen transition might take up to 300ms whereas the number is 100ms for a small switch animation. (Material Design, 2020a.) The duration might be longer on tablets as their screens are bigger.

An exception for a long animation is network-loading animation, which depends on how fast the data is transferred over the Internet. It could take a few seconds when the signal is weak. In this case, it does not reduce productivity because the user needs to wait before interacting with the application anyway (Microsoft, 2018).

2.2.4 Delay

Various animations involve multiple animating items. They move either at the same time or with small offset timing. (Nielsen Norman Group, 2020). Usually, that small offset indicates the separateness of the animated elements. That is useful when the designer wants to emphasize those elements have different functionalities even before the user realizes that. (Willenskomer, 2017.)

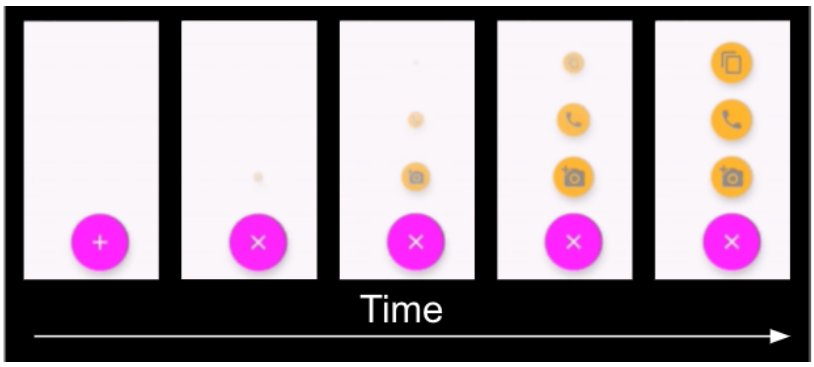


Figure 5. Timeline of a floating action menu. Screenshot (Kiat, 2019).

FabMenu, a library written by Kiat (2019), works as shown in Figure 5. This is an implementation using delays. When the pink button is clicked, at first, its icon transforms from (+) sign into (x) sign. After that, three small yellow buttons sequentially appear with small delays between them. These buttons serve distinctive purposes in the application, which is a suitable case to apply the delay technique.

2.3 Android UI components

Before using any animation frameworks, a consumer-facing-application developer needs to understand the basic components that form the UI. It is a prerequisite. In Android development, these components are activities, fragments, and views.

2.3.1 Activity

Activity is a class that creates a window to display UI elements via **setContentView** method. All activities that need to be visually shown must have a corresponding **<activity>** tag declaration in **AndroidManifest.xml**. (Android Developers, 2020a.)

```
# ./app/src/main/java/MainActivity.kt

internal class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}


# activity_main.xml
```

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <com.google.android.material.button.MaterialButton
        android:id="@+id/btnTabsTransition"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:backgroundTint="@color/colorPrimary"
        android:text="Tabs transition"
        android:textAllCaps="false"
        android:textColor="@android:color/white"
        tools:ignore="HardcodedText" />

</LinearLayout>

# AndroidManifest.xml

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="la.me.leo.animatedapp">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Script 2. Declaration of an activity and its layout file.

In script 2, **MainActivity** is declared in *AndroidManifest.xml*. It extends **AppCompatActivity**, which is a standard activity in Android development nowadays. During its creation, *setContentView* is called to guide the activity to use the correct layout file, *activity_main.xml*. The layout file has a few views such as a **LinearLayout** or a **MaterialButton**. They can be referenced in Kotlin code if necessary. Views will be discussed later in the following sections.

Interestingly, an activity has its lifecycle. Figure 6 presents the basic lifecycle of activity without taking into account the device resource consumption due to its insignificance for this thesis. An activity exists between the invocations of *onCreate* and *onDestroy*. However, it is only tangible when *onStart* is called and stays visible until *onStop* happens. Between *onResume* and *onPause*, the activity is considered in the foreground state, which means that users are able to interact with the activity. If a popup dialog appears

on top of the activity, that activity is still considered (partly) visible but is not in the foreground state. (Android Developers, 2020.)

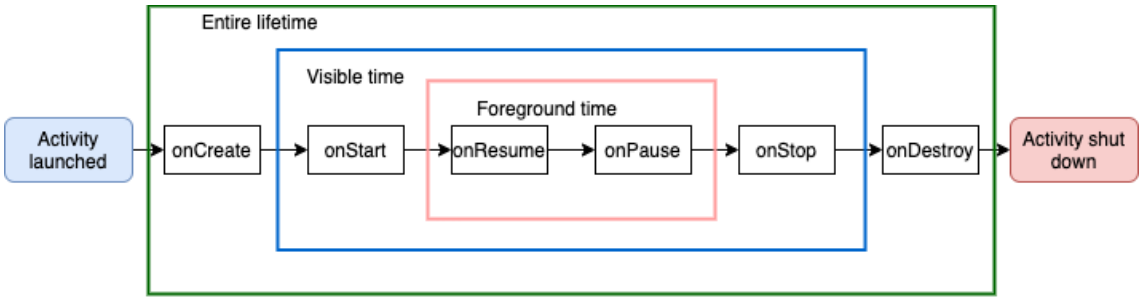


Figure 6. A simple version of the activity lifecycle.

Animations, most of the time, run in the foreground state of an activity unless it is the transition animation between one activity to another. All of the mentioned lifecycle methods are completely managed by the system (Android Developers, 2020a). Therefore, having a basic understanding of the activity lifecycle helps developers respect the system and avoid animation crashes due to illegal states. For example, animating an element after *onDestroy* is called results in a memory leak (Spitsin, 2017), an issue where the application uses up memory resources and eventually crashes.

2.3.2 Fragment

A **Fragment** is considered as a sub-activity. There are two reasons for that consideration. Firstly, it is a set of UI elements represented to users. Secondly, a developer can combine multiple fragments and control their lifecycle within a single activity. Interestingly, a fragment can be reused in different activities or even be hosted in another fragment. (Android Developers, 2019c.)

Similar to activity, a fragment has its layout defined in an XML file. The layout is inflated in the *onCreateView* lifecycle method as described in script 3.

```
# ./app/src/main/res/layout/fragment_main.xml

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
```

```
    <com.google.android.material.button.MaterialButton
        android:id="@+id/btnTabsTransition"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:backgroundTint="@color/colorPrimary"
        android:text="Tabs transition"
        android:textAllCaps="false"
        android:textColor="@android:color/white"
        tools:ignore="HardcodedText" />

</LinearLayout>


# ./app/src/main/java/MainFragment.kt

class MainFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        return inflater.inflate(R.layout.fragment_main, container, false)
    }
}
```

Script 3. Declaration of a fragment and its layout file.

A fragment has its lifecycle. That lifecycle works closely with the lifecycle of the activity hosting the fragment. In addition to methods of the activity lifecycle, there are a few more callbacks as shown in Figure 7. *onAttach* happens when the fragment is attached to or associated with the hosting activity. In contrast, *onDetach* is called when the association is no longer needed. As mentioned, *onCreateView* is in charge of creating the UI elements hierarchy from the XML layout file for the fragment. *onDestroyView*, on the other hand, is called when that hierarchy is removed. (Android Developers, 2019c.)

A developer should remember not to access any view after *onDestroyView* is invoked. Otherwise, it leads to a crash because the accessed view is removed from the hierarchy. This applies when an animation is still running on an element after it is removed.

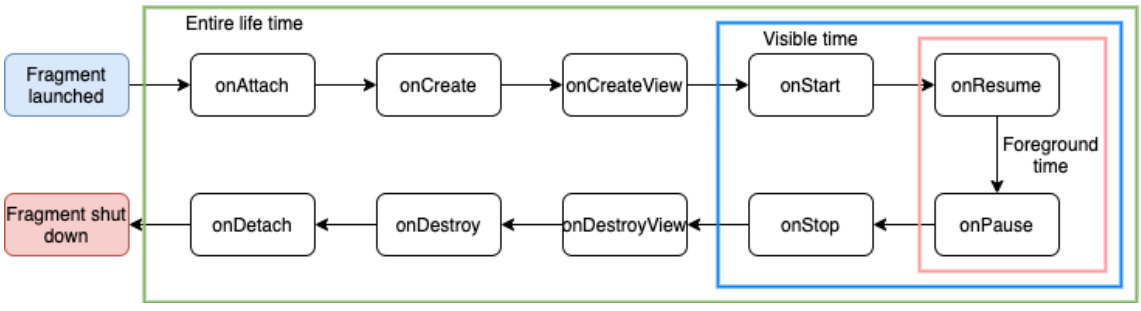


Figure 7. A simple version of the fragment lifecycle.

In the future, it is recommended to have only one activity per application so that developers work mostly with fragments (Lake, 2018). There are rational reasons why Android creators suggest that pattern, but they will be discussed in an upcoming section. Because of this development tendency, understanding the fragment lifecycle is essential.

The most common way to manage a fragment inside a parent (an activity or a parent fragment) is to use **FragmentManager**. It requires a container **ViewGroup**, usually a **FrameLayout**, inside the parent layout to host those fragments. A developer is capable of adding, removing, or replacing fragments inside a container. These actions must be done in a transaction.

On occasion, a transaction has to be reverted when a user presses the back button on the device. A typical case for this is demonstrated in Figure 8. When the user presses a button in fragment A, the application navigates to fragment B. Pressing the back button should bring that user to fragment A.

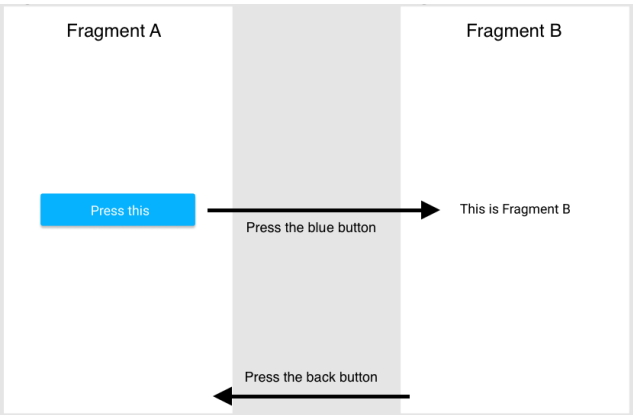


Figure 8. An example of fragment navigation

In that case, not only the transaction when the blue button is clicked should remove fragment A and add fragment B, but it also needs to be added to the back stack before committing the transaction as shown in script 4.

```
# layout of the parent that host fragment A and fragment B

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
```

```
    <FrameLayout
        android:id="@+id/fragment_container"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</LinearLayout>


# executed code when the button in fragment A is clicked


// begin the transaction
val transaction = supportFragmentManager.beginTransaction()
// remove fragmentA from container if it exists
transaction.remove(fragmentA)
// add fragmentB to container with id of fragment_container
transaction.add(R.id.fragment_container, fragmentB)
// add the transaction to a back stack so that it can be reverted
transaction.addToBackStack(null)
// commit the transaction to take effect
transaction.commit()
```

Script 4. An example of adding a transaction to the back stack.

A fragment transaction, most of the time, results in a transition between scenes. In animated applications, designers might want to introduce their custom transitions instead of using the default effect from the system. With the Android framework, it is possible to achieve that goal. This matter will be explained later in this paper.

2.3.3 View

Any subclass of **View** represents a UI component. Its bound is a rectangular area where it is drawn on the screen and intercept interaction events such as tapping, holding, or scrolling. (Android Developers, 2020g.)

A **ViewGroup** is a special view that acts as a container for other views and controls the position relations of these views (Android Developers, 2020g). In script 5, the **ImageView** is held within a **LinearLayout**, which is a subclass of **ViewGroup**. Some regular view groups are **ConstraintLayout**, **LinearLayout,** and **FrameLayout**. They have different mechanisms to layout their child-views.

In layout files, every XML tag corresponds to an instance of a view such as buttons, labels, or containers. These views can be referenced in Kotlin/Java code using the method *findViewById*. In script 5, the **ImageView** with identifier *ivArticle* is referenced in a Kotlin class. (Android Developers, 2020g.)

```
# layout file

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ImageView
        android:id="@+id/ivArticle"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

</LinearLayout>


# reference a view in Kotlin code

val ivArticle = findViewById<ImageView>(R.id.ivArticle)
```

Script 5. An example of referencing a view from XML layout to Kotlin code.

In general, a referenced view could safely be an animated element of an animation as long as it exists in the view hierarchy.

2.4 Android animation APIs

2.4.1 Animator

Animator is a modern animation system for Android development. Animators allow developers to change the properties of any Kotlin/Java object over time. There are multiple types of animators. They are **ObjectAnimator**, **ValueAnimator,** and **AnimatorSet**. Although **ViewPropertyAnimator** is not classified as an animator, it is based on **ValueAnimator**. (Liu et al., 2018.)

They all gradually adjust one or more properties from one value to another value during a period. All of them allow developers to delay the starting point of the animation and specify the easing curve for the animation using the **Interpolator**. Sometimes, they bring similar effects, which confuses developers (Elye, 2020). Understanding each type helps them make a better decision on which animator should be used.

First and foremost, **ViewPropertyAnimator** is used when the developer does not intend to control the animation after it starts. Unlike subclasses of **Animator**, **ViewPropertyAnimator** does not support animation coordination or repeating because

it does not inherit from **Animator**. However, it is slightly more efficient according to Liu (2018). The API of **ViewPropertyAnimator** is simple. For example, script 6 is used to change the opacity (alpha channel) of a view to 1 in 200 milliseconds.

```
view.animate()
    .alpha(1f)
    .setDuration(200)
```

Script 6. Change the opacity of a view in Kotlin code.

Second, **ObjectAnimator**'s capability is to move the value of a property in a range of integers or floats. This does not only apply to view specifically but to any Kotlin/Java object in general. As long as the property has getter and setter, **ObjectAnimator** controls it using Java reflection. Due to reflection usage, it is less performant than **ViewPropertyAnimator**. (Liu et al., 2018.) Script 7 shows how to rotate a view around the horizontal axis. The property here is called *rotationX*. This may also be achieved with **ViewPropertyAnimator** which is more optimized as mentioned. Nevertheless, since **ObjectAnimator** inherits from **Animator**, it provides more control over the animation. For example, it is possible to cancel this animation later if the developer wishes to.

```
val animator = ObjectAnimator.ofFloat(view, View.ROTATION_X, 0f, -90f)
    .setDuration(400)
animator.start()
```

Script 7. Rotate a view around the horizontal axis in Kotlin code.

Thirdly, **ValueAnimator** animates values such as colors or numbers and applies the animated values to one or more targets. It is mostly used for custom animations, but confusingly, in specific cases, it also brings the same effects as **ObjectAnimator**. (Liu et al., 2018.) A simple explanation for this is that **ObjectAnimator** inherits from **ValueAnimator**. While **ObjectAnimator** focuses deeply on animating an object's property, **ValueAnimator**'s usage is more generic. In figure 9, a rare animation is introduced. After the text "Setting up your account", there are three dots that appear sequentially. One simply should not use **ObjectAnimator** in this case because the property's value is not an integer or a float but a string. In the code, a **ValueAnimator** animates a value from 0 to 4 and it is the developer's job to map it to the number of dots displayed.

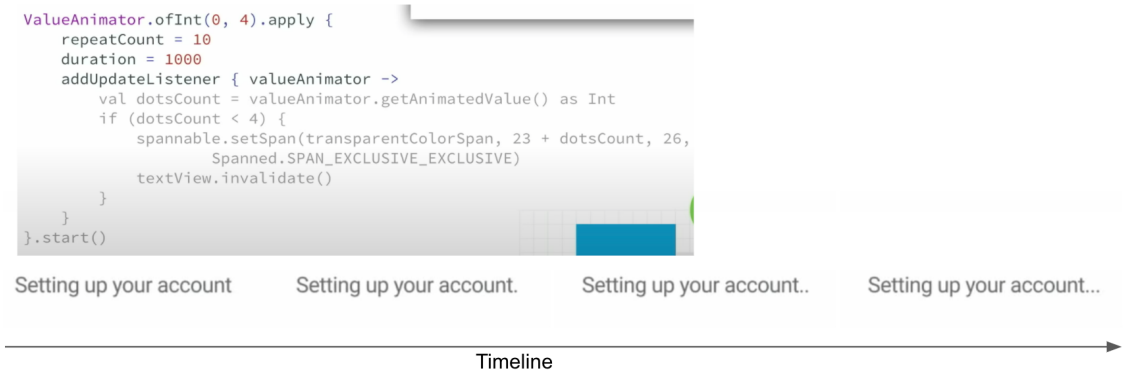


Figure 9. An implementation of animation using ValueAnimator. Screenshot (Liu et al., 2018).

Last but not least, **AnimatorSet** is a special animator because it does not animate anything on its own. It combines other animators and coordinates them. (Liu et al., 2018.) Figure 10 shows a transition between two views. They display 2 texts, which are “Hang on. Setting up your account” and “Oh no! Something went wrong”. When the transition happens, the first text fades out before the second text fades in and is scaled at the same time. From the implementation, the API of **AnimatorSet** is self-descriptive and close to human language. The *animatorSet plays* the *fadeInText* animation *with* the *scaleText* animation *after* the *fadeOutText* animation.

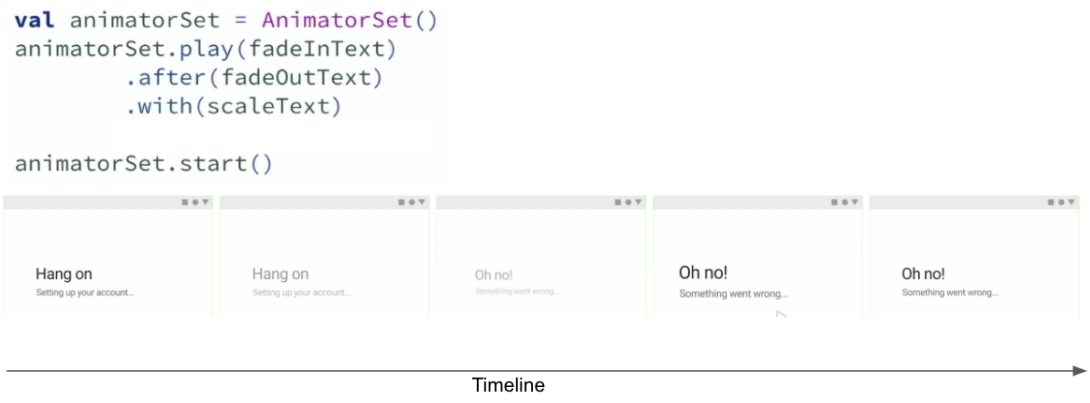


Figure 10. A usage of AnimatorSet. Screenshot (Liu et al., 2018).

Referring back to the example in the introduction section (script 1), the author starts the second animator after the first animator by listening to its end event. It is a misusage as the proper way is to play them sequentially in an animator set. Another mistake is that

the author delays an animator by calling *postDelay* on a view while using *setDelay* directly on the animator is a cleaner approach.

There is another animation system in Android development. It is called **ViewAnimation**. However, this system is ancient. Nick Butcher also states that **Animator** provides most of the features that **ViewAnimation** has. Because of its versatility, Android developers should consider **Animator** over **ViewAnimation**. (Liu et al., 2018).

2.4.2 Transition

In Android development, transitions are divided into several types. Enter transition happens on views of a scene when that scene appears. Exit transition, on the other hand, occurs on views of a scene when that scene is replaced or hidden by an appearing scene. A shared element transition refers to the motion where a UI element on the disappearing scene transforms into another element on the appearing scene. (Android Developers, 2020f). Figure 11 illustrates a shared transition where an item inside the list from one screen transforms into a big image inside another screen.

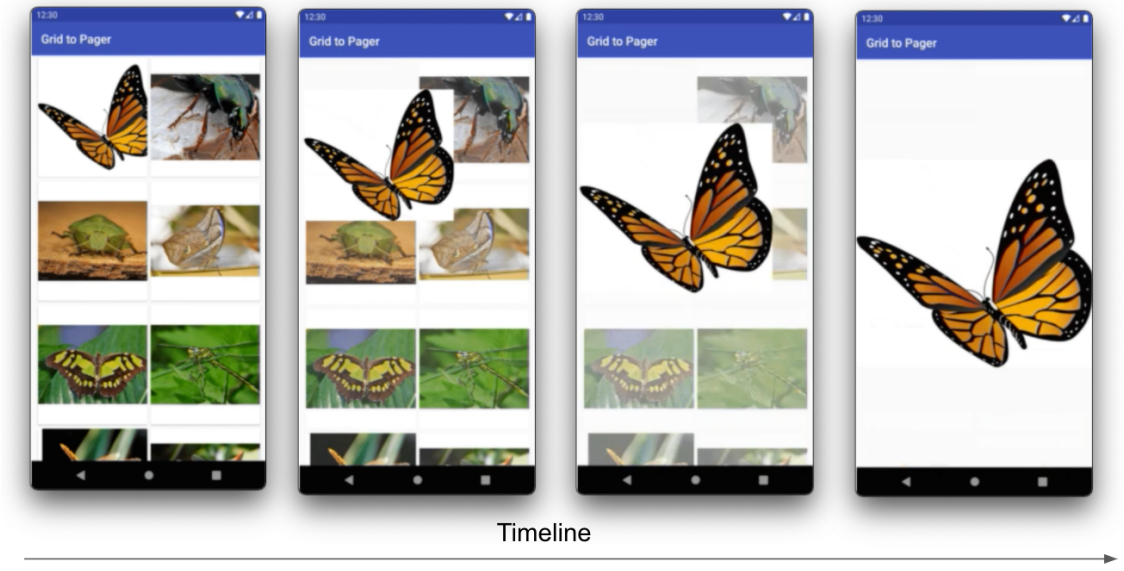


Figure 11. A shared transition between two screens. Screenshot (Android Developers, 2020d.)

Normally, a transition is represented by an instance of the class **Transition**. To create a custom transition, a developer simply writes a subclass of that class and overrides

createAnimator, *captureStartValues*, and *captureEndValues* methods. The latter two methods capture a snapshot of a view before and after then transition while the first method creates the actual animator for the transition based on the two snapshots. Some ordinary built-in transitions are **Fade**, **ChangeBounds**, and **Slide**. (Android Developers, 2019b.)

For the shared element transition case, different situations require different declarations.

2.4.3 Activity transition

Activity transition refers to the transformation effects that happen when the application navigates between activities. To use activity transition in Android, developers have to enable *windowActivityTransitions* in the application theme. They need to either specify the enter, exit, or shared element transitions in that theme XML declaration or programmatically set them using Kotlin code. (Android Developers, 2020f). Script 8 shows how to specify them using the former way.

```
<style name="BaseAppTheme" parent="android:Theme.Material">
    <!-- enable window content transitions -->
    <item name="android:windowActivityTransitions">true</item>

    <!-- specify enter and exit transitions -->
    <item name="android:windowEnterTransition">@transition/explode</item>
    <item name="android:windowExitTransition">@transition/explode</item>

    <!-- specify shared element transitions -->
    <item name="android:windowSharedElementEnterTransition">
        @transition/change_image_transform
    </item>
    <item name="android:windowSharedElementExitTransition">
        @transition/change_image_transform
    </item>
</style>
```

Script 8. Specify activity transitions in the application theme (Android Developers, 2020f.)

The Android documentation gives an impression of a simple API. However, in reality, using activity transition is usually painful. In Android Dev Summit event, Lake (2018), an engineer at Google, says that the API works differently across devices and Android versions. According to him, developers have to check for the device version in their code before using **Fade** and exclude the status bar and navigation bar from the transition if the version is higher than Lollipop (Android 5.0). Otherwise, it visually breaks the

transition. Lake argues that introducing such checks in UI code makes the code unreliable. This is one of several cases.

2.4.4 Fragment transition

Similar to activity transition, fragment transition means the motion effect that visually links fragments when navigating between them. The APIs of enter transition and exit transition for fragment are fairly simple. The developer has to call *setEnterTransition* and *setExitTransition*. (Android Developers, 2020d.)

For shared transition, it is more complicated as there are several required steps to achieve the transition. Firstly, the shared views from the disappearing fragment and appearing fragment need unique transition names. Those views and their transition names are then supplied to the **FragmentTransaction** when the navigation logic starts. Last but not least, the transition must be set to the destination fragment via *setSharedElementEnterTransition*. (Android Developers, 2020d.)

2.4.5 Layout transition

Modifying the elements in the view hierarchy of an activity or a fragment results in visual changes. The transition that connects the hierarchy changes, in this case, is called layout transition. (Android Developers, 2020b.)

The framework works by storing the initial state and the final state of the view hierarchy and animating the changes between these states. To help the framework store the initial state, the developer has to call *TransitionManager.beginDelayedTransition* method and provide the view containing all the views that may be animated and an instance of the desired transition. (Android Developers, 2020b.)

Script 9 demonstrates the usage of this API. It uses **AutoTransition** to animate the layout after adding a **CheckBox** to the hierarchy of the view *root*.

```
val transition = AutoTransition()
TransitionManager.beginDelayedTransition(root, transition)
root.addView(CheckBox(activity))
```

Script 9. Using layout transition in Android development

This API is powerful since it only requires one method call to achieve decent animation before modifying the UI. Of course, in case a custom transition is required, programmers need to implement the transition themselves instead of using built-in ones such as **AutoTransition**.

2.4.6 Lottie

With the contribution of the open-source community, there are many Android libraries that assist developers to build magnificent animations. Unfortunately, most of them are built for particular purposes. For instance, paper-onboarding-android is a library used to create one animation for the onboarding screen of an application (Ramotion, 2020). Therefore, it is hardly reused in any other places of that application. Normally, a team should not introduce a third-party library that serves only one purpose but write the code themselves because a library may drastically increase the size of the application (Android Developers, 2021b). However, it is not the case with Lottie.

Lottie is not included in Android SDK. It is a library written by Airbnb developers. Lottie allows developers to render Adobe After Effects animations natively not only on Android phones but also on iOS devices and web browsers. (Airbnb.io, 2020.)

In case developers want to control the movement of UI elements, Lottie is not an option. Android animator APIs or transition APIs should be used instead. Lottie should be used when developers need to show cartoon-like animations in a specific area on the screen.

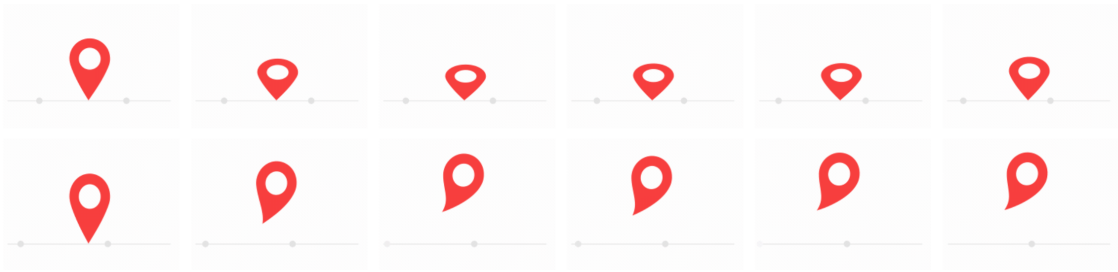


Figure 12. Frames of a cartoon-like animation. Screenshot (Airbnb.io, 2020).

The library APIs are straightforward. First, designers need to export their animations in JSON format using Adobe After Effects program and Bodymovin extension. The JSON file's size depends on how long and complex the animation is. It may take up to more than 1MB if the animation contains multiple image layers. That JSON file is then included in the source code as a raw asset in the *res* directory. To show it on a screen, the developer just adds **LottieAnimationView** to the layout and passes the file to the *setAnimation* method as shown in script 10. (Airbnb.io, 2020.)

```
# ./app/src/main/res/raw/onboarding.json
{
    // lottie json content
}

# ./app/src/main/java/OnboardingFragment.kt

val lottieView = findViewById<LottieAnimationView>(R.id.lottieView)
lottieView.setAnimation(R.raw.onboarding)
```

Script 10. Using Lottie to show animation

Lottie also provides more control over the animation. For example, developers can set the frame range to be played with *setMinFrame* and *setMaxFrame* methods or listen to the progress of the animation with *addAnimatorListener*. This is possible because Lottie translates the JSON file into an Android animator. (Arbnb.io, 2020.)

Interestingly, there are many use cases where designers want to show cartoon-like animations. If it is introduced in the application once, soon, there is likely more to be added. Therefore, developers can reuse Lottie in a codebase. It is a fair trade-off as the library adds less than 290kB (uncompressed) but reduces the drawing work for developers in several places (Arbnb.io, 2020).

2.5 Maintainable code

Maintainability refers to how easy to modify code in a system. It is one criterion to determine the quality of software. It affects the business tremendously. When the codebase is easily maintained, the cost and time for the maintenance job decrease. It means that there is more time for other meaningful tasks like developing new functions. Thus, the product is delivered to the market fast. More importantly, maintainability is a prerequisite for certain other quality characteristics. Improving software in any aspect

requires modifications, which is either tiny or huge. In any case, a more maintainable system makes these changes easier for engineers. (Visser et al., 2016.)

2.5.1 Simplicity

Primarily, developers need to write short and simple units of code (Visser et al., 2016). Writing long and complicated code does not only take more time to write and test but also makes it more difficult to analyze and modify in the future (Carullo, 2020). Robert C. Martin (2010), as known as Uncle Bob, also suggests those code units should hold single responsibility and do not cause side effects. Visser and his colleagues (2016) recommends the audience to limit the code unit's length to only 15 lines and limit the condition branches to 4 for each unit. This way, the code can be fully covered by tests; and the chance to forget edge cases is lessened.

2.5.2 Reusability

Simplicity helps developers avoid duplicating logic in different places. Duplicated code is considered bad practice because a change in one place leads to changes in all other places. Reusability is a principle that all software engineers should follow. It saves time and effort since the code is written once and reused by other components. It reduces redundancy to a minimum level. (Carullo, 2020.) One mistake that most beginners make is that they use a string multiple times in the codebase. If a string is used the second time, extract it to a constant and use the constant instead. It reduces typo mistakes when typing the string again.

2.5.3 Readability

Readability can be measured by how fast the code can be understood (Carullo, 2020). Software is often developed by a team instead of a single person. Therefore, it is important that other developers easily understand the code of one team member. Even if there is only one member in the team, which is rare, his/her brain may also forget what it created after a certain period and need to read again to remember what the code does. In either case, readability is essential.

There are different methods to increase readability. The most basic approach is formatting. Engineers agree with each other on how to format their source code. Formatting must be consistent in a team. (Martin, 2010.) For example, if the team decided to use 4-space indentation, no one should commit a line with 2-space indentation.

The second method is using meaningful names in the code. Ambiguous names must be avoided in all circumstances. It does not matter if the name is short or long. The important thing is that readers know their meaning and purpose without difficulties. (Carullo, 2020.) Martin (2010) says that consistent naming conventions should be used across different components to prevent mental mapping. For instance, if 'delivery location' refers to a place that users want something to be sent to and exist in the codebase, a developer should not introduce the term 'address'.

2.5.4 Modularity

Modularity is about achieving loose coupling between components of the software. Tight coupling leads to maintenance problems as modifying one dependency may lead to modifying other components. In contrast, loosely coupled modules isolate parts of the codebase and reduce the impact on other parts when modifying one part. (Visser et al., 2016.)

Each separated module should be cohesive. It contains only code that serves a defined function. In other words, a module encapsulates a function implementation, hides it from other modules, and only exposes that functionality. (Carullo, 2020.)

Gradle module in Android developments

Projects that use the Gradle build system, especially the majority of Android projects, splitting code into library modules is highly recommended.

The first benefit is cleaner third-party dependencies management using a special module. That module is called *buildSrc*. It is compiled first and will be visible to all other modules. Module *buildSrc* will be placed in the root of the project and declares necessary libraries in Kotlin files. It also requires a build script so that Gradle understands it is a

part of the build process. The build script name is *build.gradle.kts* (Edwards, 2018.) Script 11 demonstrates how to manage dependencies in the *buildSrc* module and use them in the *app* module.

```
# ./buildSrc/build.gradle.kts

plugins {
    `kotlin-dsl`
}

repositories {
    jcenter()
}

# ./buildSrc/src/main/java/Dependencies.kt

object Versions {
    const val transition = "1.4.0-beta01"
    const val androidX = "1.1.0"
    ...
}

object Dependencies {
    const val transition = "androidx.transition:transition-ktx:${Versions.transition}"

    const val appCompat = "androidx.appcompat:appcompat:${Versions.androidX}"
    ...
}

# ./app/build.gradle.kts

plugins { ... }

android { ... }

dependencies {
    implementation(Dependencies.appCompat)
    implementation(Dependencies.transition)
}
```

Script 11. Declare third-party dependencies in buildSrc to reuse in other modules.

With *buildSrc*, updating a dependency is fast because developers only need to modify the version of that dependency in the **Versions** object.

Secondly, separating modules makes compilation time faster compared to a single module project because they are built simultaneously. Moreover, modifying code in one module only requires that module to be rebuilt before running the program. Build files of untouched modules are cached for optimization. (Android Developers, 2020e.) In large projects which involve multiple development teams, this separation makes the ownership of each team and the dependencies between teams clearer. Ideally, one module is only owned by one team so that it does not require tight collaboration across teams to change

code in that module. For instance, if team A depends on team B, and both rely on team C, each team can simply have their modules inside the project and declares these dependencies. They are visualized in figure 13.

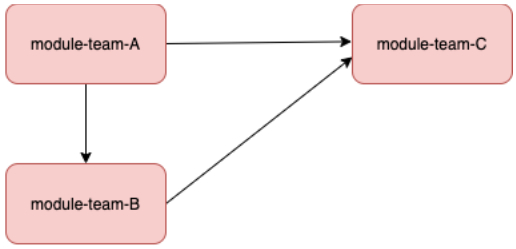


Figure 13. Module dependency graph

To implement this dependency in the project, developers need to follow a few steps. Similar to *buildSrc*, normal modules must have their build scripts. Unlike *buildSrc*, a module must be included in *settings.gradle.kts* file and cannot use other modules' exposed code without declaring them as in script 12. (Gradle.org, 2020.)

```
# ./settings.gradle.kts

include(":module-team-a", ":module-team-b",":module-team-c")
rootProject.name = "Animated App"

# module-team-a/build.gradle.kts

plugins {
    id("com.android.library")
    id("kotlin-android")
}

android { ... }

dependencies {
   implementation(project(":module-team-b"))
   implementation(project(":module-team-c"))
}

# module-team-b/build.gradle.kts

plugins {
    id("com.android.library")
    id("kotlin-android")
}

android { ...}

dependencies {
   implementation(project(":module-team-c"))
}


# module-team-c/build.gradle.kts
```

```
plugins {
    id("com.android.library")
    id("kotlin-android")
}

android { ... }

dependencies {
    ...
}
```

Script 12. Declare a module and use it in another module via Gradle.

Even if the whole codebase is owned by one team only, it is better to have multiple modules to encapsulate functions and reduce exposing unnecessary code. It is also easier to introduce multiple teams into the project as it grows in the future.

3 Implementation

3.1 Implementation approach

To create snippets of code that create beautiful animations and ease the future maintaining work, it is objective to write them while building a real-world example of highly animated Android applications. Writing these components independently causes difficulties when integrating them into a project later as it is impossible to foresee the requirements of that project.

Wolt: Food delivery is among the most animated applications on Play Store and App Store. It is a food-delivery application developed by a Finnish startup called Wolt Enterprises Oy and published in 2015. Since then, Wolt has been entitled to several awards related to design such as Vuoden Huiput's Golden Awards for two categories Digital Design and Product Design in 2016 (Grafia.fi, 2016).

One big contribution to these great achievements is introducing proper motions in the UI. As discussed in section 2.1, animations increase the emotional connection and usability part of the user flow. They lead users to their goals faster with less friction and less overload. Using Wolt, users may realize the application contains many motion effects. The demonstrated project for this paper will mimic a few effects from this world-class mobile application and extract the core code for producing these effects into a component so that it can be reused, maintained, and extended in the future.

The selected animations for the project are the onboarding animations, the transition between tabs, and the the scrolling behavior in the venue detail page. These three are chosen because they are different in terms of attributes and implementation. Therefore, building those increases the coverage of animating techniques in Android development and the completion of the component.

Java used to be the official language for Android development. However, Google announces Kotlin becomes the new successor of Java in this area due to its expressiveness, conciseness, and safeness (Android Developers, 2021a). Therefore, the programming language chosen for this project is Kotlin.

3.2 Software architecture

3.2.1 UI architecture

Even though there is a new engine to build Android UI, Jetpack Compose, a declarative UI framework developed by Google, is still in the alpha stage and is not stable for production usage (Android Developers, 2020c). Therefore, the most common way to build UI elements in Android projects is to use Android native views.

To host views, developers have to use fragments and activities. They can use only activities because fragments are optional. However, activity's behavior such as its default enter transition depends on the manufacturer, OS version, and user's selected theme. Although Google provides APIs that allow developers to override the default behavior, the APIs do not work gracefully for all devices and may bring negative side effects as mentioned in section 2.4.3. Lake (2018) even argues that even if Google fixes the issue in the newer Android API level, it is still difficult to support backward compatibility. For an animated application, customized transitions are used frequently. It would be impossible to test if they behave correctly in thousands of devices. To reduce the flakiness of transition, this project will minimize the usage of activity and follow what Lake suggests in Android Dev Summit 2018 event, embracing single-activity pattern.

Single-activity pattern implies in one Android project, there should only be one activity acting as the entry point of the application. All UI elements of the application are hosted inside fragments, which are managed by that single activity. (Lake, 2018.)

In the end, a fragment is considered as a sub-activity. It has the full capability to replace an activity.

3.2.2 Modules architecture

As described in section 2.5.4, modularity is one factor for a maintainable project. Splitting a project into multiple modules brings tremendous advantages for Android developers.

First and foremost, this implementation undoubtedly leverages the advantages of the *buildSrc* module to manage third-party libraries.

Secondly, the shared code for animation that is used throughout the project can be encapsulated in one module, called **core-animation**. This module is the heart of this project because it contains the purpose of this implementation, APIs for writing maintainable animation code. It hides the complex code inside and only exposes short and simple APIs to dependent modules so that the animation code in those modules is cleaner.

Resource files such as Lottie JSON files stay in one module, called **app-resource**.

Since the three animations presented in this project happens on three different screens, namely the onboarding screen, the tabs screen, and the venue detail page, three Android modules are created for each of them. They will depend on **core-animation** and **app-resource**.

The previous section states that this project follows the single-activity approach. Therefore, it is sensible to have one module to host that entry activity of the application. By default, that module is named *app* when a new project is created with Android Studio. The activity controls the fragments inside it, meaning that this module needs the three screen-modules. Moreover, if the module *app* needs to use resource files or animation code, it requires both *core-animation* and *app-resource*.

The architecture of this project is shown in figure 14.

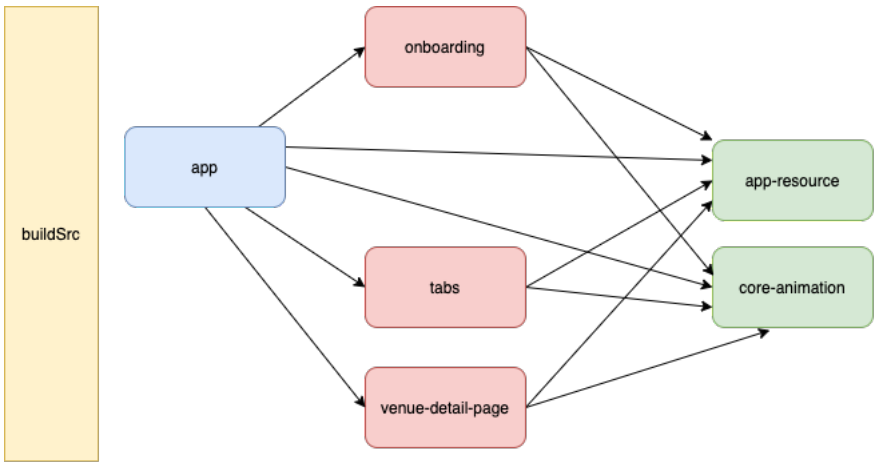


Figure 14. Module dependency graph

3.3 Onboarding animation

3.3.1 Specification

Figure 15 visualizes how the animations in the onboarding screen should look like.

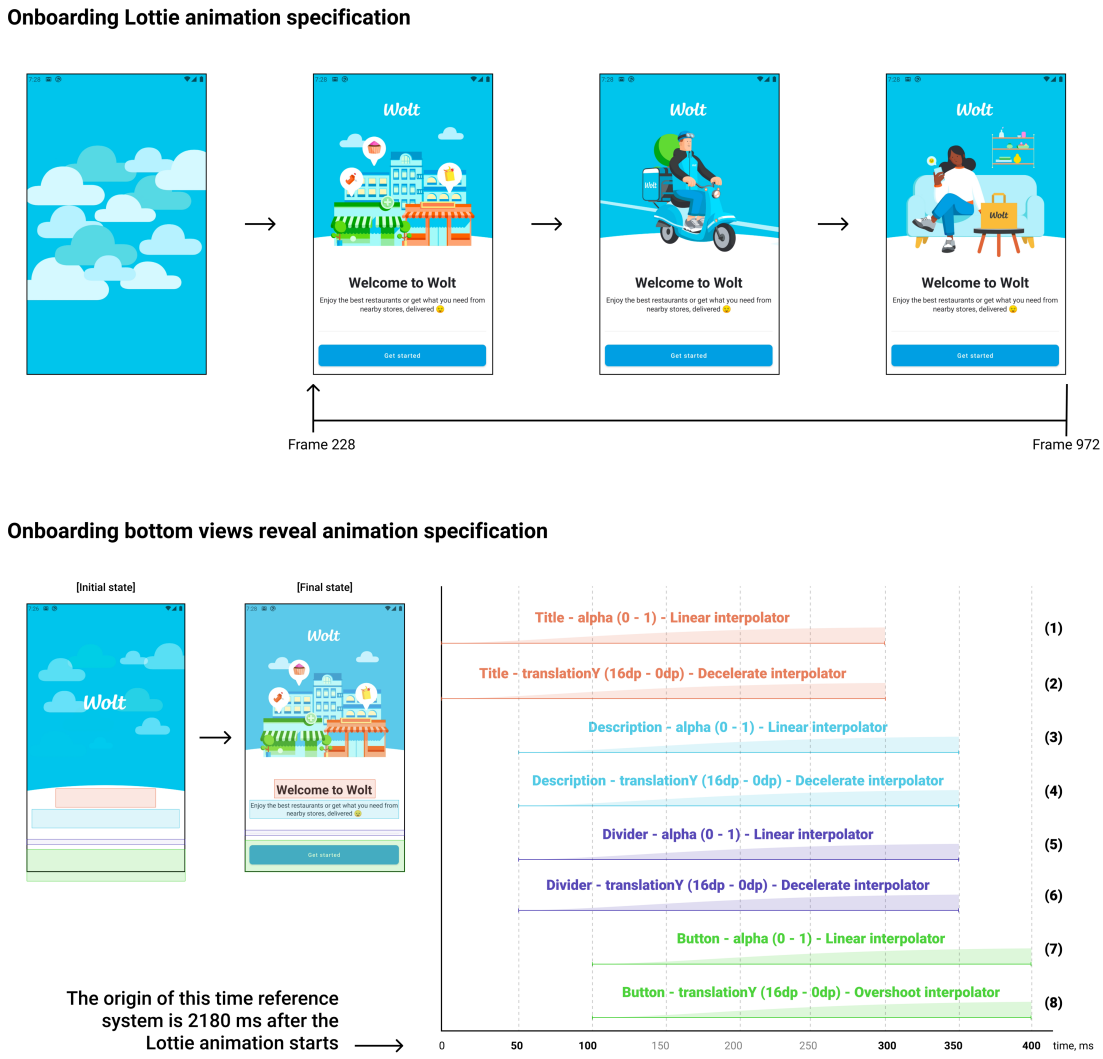


Figure 15. Onboarding screen animation specification.

According to the specification, two main animations are happening on the screen.

The first one is Lottie animation, which has 4 main scenes. The first scene only appears once before the other three takes turn to be shown. In other words, the frames between

frame 0 and frame 227 of the Lottie animation are only rendered once while the frames between frame 228 and frame 972 are rendered repeatedly in order.

The second animation runs after the first one has started for 2180 milliseconds, and it is a combination of 8 smaller motions. The time graph in the specification indicates their components. For example, the motion (1) runs on the title view, which is marked in the reddish-orange box, and changes its alpha (opacity) from 0 to 1 in 300 milliseconds without any delay.

3.3.2 Implementation

A fragment called **OnboardingFragment**, defined in the **onboarding** module, presents the onboarding screen. When the screen is shown, the texts, the button, and the divider are hidden. Therefore, the implementation should prepare the UI before the animation happens. It is enough to make those views transparent and disable the button, as the users should not be able to click before it is visible. This can be done with script 13.

```
private fun prepareUiBeforeAnimation() {
    btnDone.isClickable = false
    tvTitle.alpha = 0f
    tvDescription.alpha = 0f
    vDivider.alpha = 0f
    btnDone.alpha = 0f
}
```

Script 13. Implementation of *prepareUiBeforeAnimation*.

The animations for the views at the bottom of the UI can be performed in a group by using **AnimatorSet**. Because there are several animations created, it is a good practice to create a function, called *constructAnimator*, that is responsible for the creation. Its implementation is placed in the *core-animation* module as in script 14.

```
# core-animation/src/main/java/.../animator/AnimatorFactory.kt

fun constructAnimator(
    duration: Int, interpolator: Interpolator? = null, delay: Long = 0,
    onUpdate: (Float) -> (Unit), onStart: (() -> (Unit))? = null,
    onEnd: (() -> (Unit))? = null
): ValueAnimator {
    val animator = ValueAnimator.ofFloat(0f, 1f)
        .setDuration(duration.toLong())
    interpolator?.let { animator.interpolator = it }
    animator.addUpdateListener { onUpdate(it.animatedFraction) }
    if (onStart != null || onEnd != null) {
        animator.addListener(object : AnimatorListenerAdapter() {
```

```
            override fun onAnimationStart(animation: Animator?) {
                onStart?.invoke()
            }

            override fun onAnimationEnd(animation: Animator?) {
                onEnd?.invoke()
            }
        })
    }
    animator.startDelay = delay
    return animator
}
```

Script 14. Implementation of *constructAnimator*.

In this case, the function should return an animator so that it can be added to an **AnimatorSet**. That means **ViewPropertyAnimator** should not be used because it is not a subclass of **Animator**. The output of the function should be either **ObjectAnimator** or **ValueAnimator**. It is mentioned in section 2.4.1 that **ValueAnimator** is a superclass of **ObjectAnimator**, and it is capable of replacing **ObjectAnimator** as well as creating more customized animations. Another advantage of **ValueAnimator** is that it allows developers to animate multiple properties of different views as long as their motions possess the same duration, delay, and easing curves. Therefore, the superclass is preferred. The function takes 3 components of animation*, duration,* easing curve *(interpolator),* and *delay*, as inputs. It also provides additional callbacks. *onUpdate* is a mandatory callback, which is called whenever the progress value of the **ValueAnimator** changes. The progress value is between 0 to 1. Typically, the API consumer calculates the value of the animated property based on the progress value and adjusts UI elements in this callback. *onStart* and *onEnd* are optional callbacks that give the developer the ability to take actions when the animation starts or ends. With the implemented **constructAnimator** function, it is straightforward to implement the animations for those views at the bottom follow the specifications as shown in script 15.

```
private fun createBottomViewsAnimatorSet(): AnimatorSet {

    val animations = listOf(
        constructAnimator(300, LinearInterpolator(),
            onUpdate = { tvTitle.alpha = it }                    // motion (1)
        ),
        constructAnimator(300, DecelerateInterpolator(),
            onUpdate = { tvTitle.translationY = 16 * (1 - it) } // motion (2)
        ),
        constructAnimator(300, LinearInterpolator(), delay = 50,
            onUpdate = {
                tvDescription.alpha = it                         // motion (3)
                vDivider.alpha = it                              // motion (5)
            }
        ),
        constructAnimator(300, DecelerateInterpolator(), delay = 50,
```

```
            onUpdate = {
                tvDescription.translationY = 16 * (1 - it)      // motion (4)
                vDivider.translationY = 16 * (1 - it)           // motion (6)
            }
        ),
        constructAnimator(300, LinearInterpolator(), delay = 100,
            onUpdate = { btnDone.alpha = it },                  // motion (7)
            onEnd = { btnDone.isClickable = true }
        ),
        constructAnimator(300, OvershootInterpolator(), delay = 100,
            onUpdate = { btnDone.translationY = 16 * (1 - it) } // motion (8)
        )
    )

    val animatorSet = AnimatorSet()
    animatorSet.playTogether(animations)
    animatorSet.startDelay = 2180
    return animatorSet
}
```

Script 15. Implementation of *createBottomViewsAnimatorSet*.

The function *createBottomViewsAnimatorSet* belongs to **OnboardingFragment**. It creates all the animations according to the given specification. Even though there are 8 motions according to the requirement, there are only 6 animators created because the description text and the divider have the same motions. Therefore, motion (3) and motion (5) can be created with only one animator while another one achieves motion (4) and motion (6). All of the created animators are played together under an **AnimatorSet**. It also adds a delay of 2180 milliseconds to the returned animator. One important thing is that the button is re-enabled after it is fully visible by using the *onEnd* callback.

The next step is to show the Lottie animation on the UI. Needless to say, the Lottie JSON file, *onboarding.json*, has to be placed in the *app-resource* module. After the resource file is ready, the function that prepares the configuration for the Lottie view is written in **OnboardingFragment** as in script 16.

```
private fun setupLottieAnimation(bottomAnimation: Animator) {
    lottieView.setAnimation(R.raw.onboarding)
    lottieView.setRepeatCount(INFINITE)
    lottieView.setMaxFrame(972)
    lottieView.addAnimatorListener(object : AnimatorListenerAdapter() {

        override fun onAnimationStart(animation: Animator?) = bottomAnimation.start()

        override fun onAnimationRepeat(animation: Animator?) =
lottieView.setMinFrame(228)

    })
}
```

Script 16. Implementation of *setupLottieAnimation*.

In the requirement, the frames within frame 0 and frame 227 are shown once before the frames between 228 and 972 are looped. Therefore, *setMinFrame* is called when the animation repeats to prevent the view from replaying scene 1. The animations of bottom views are also played when the Lottie animation starts.

The last preparation is to cancel the animator when the fragment is destroyed. This is important because uncleared animators may cause memory leaks. This action may happen in another place because an application has multiple fragments, and some of those fragments may also have animators running. Therefore, creating a function to stop an animator in the *core-animation* module for reusable purpose is necessary.

```
fun Lifecycle.cancelAnimatorOnDestroy(animator: Animator) {
    addObserver(object : LifecycleObserver {
        @OnLifecycleEvent(Lifecycle.Event.ON_DESTROY)
        fun onDestroyed() {
            removeObserver(this)
            animator.cancel()
        }
    })
}
```

Script 17. Implementation of *cancelAnimatorOnDestroy*.

In script 17, *cancelAnimatorOnDestroy* is an extension function of the **Lifecycle** class. It attaches an observer to the lifecycle (of a fragment or an activity) to determine when it is destroyed and cancel the animator accordingly.

The final step is to start those created animators after the fragment's view hierarchy is created as in script 18.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    prepareUiBeforeAnimation()
    val bottomAnimator = createBottomViewsAnimatorSet()
    lifecycle.cancelAnimatorOnDestroy(bottomAnimator)
    setupLottieAnimation(bottomAnimator)
    lottieView.playAnimation()
}
```

Script 18. Start onboarding animation.

From this implementation, it is noticeable that the animations of the texts, the divider, and the button are dependent on the Lottie animation. One may argue that these dependencies between those animations are not needed because the animations for the

views at the bottom of the UI can start immediately after the screen is loaded with a delay of 2180 milliseconds. However, that argument is invalid because it does not count the Lottie animation loading time. With that approach, the motions of the texts and buttons are not synchronized with the Lottie animation and appear earlier than expected as the loading time may take up to 0.3 seconds to 0.5 seconds on weak devices.

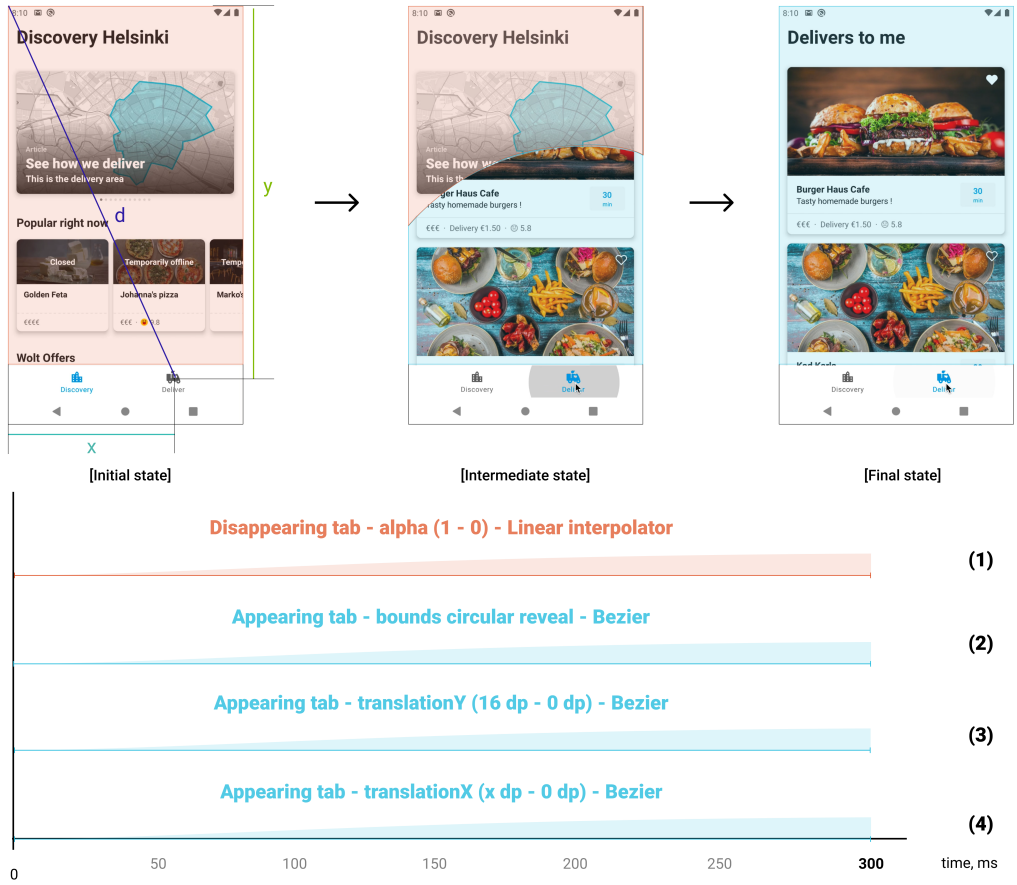
After accomplishing the first animation, the *core-animation* module has its first two methods *constructAnimator* and *cancelAnimatorOnDestroy*. They lay the first foundation for this project's future. Without *constructAnimator*, the implementation of *createBottomViewsAnimatorSet* will be gigantic, and so will the future animations.

3.4 Transition between tabs

3.4.1 Specification

In the application, there is a screen that has a bottom navigation bar that allows users to switch the content on the screen by selecting one of the icons in it. Each of those icons is called a tab. For simplicity purpose, this project only shows only two tabs on the screen. They are the discovery tab and the deliver tab. When the user switches to a new tab, the content also changes. To reduce friction for that change, a transition is needed. Its specification is described in figure 16.

According to it, both disappearing and appearing contents are animated. While the disappearing content only fades out gradually in 300 milliseconds, the appearing content has three motions in total. Unlike the onboarding animation, where the motions have constants for their animated values, the animation for the appearing area is screen-size-specific, meaning that the animated values differ between screen sizes. For example, while the *translationX* motion requires *x*, the bounds circular reveal motion depends on three dynamic parameters *x*, *y,* and *d*. Those three values are calculated at runtime. The specification defines *x* as the distance between the left bound of the screen to the center point of the tab, *y* as the distance between the top bound to that same point, and *d* as the distance between the top-left corner of the screen and the tab's center.



Notes:
- x, y, d are defined in the initial state screen shot
- Bounds circular reveal animation has center at (x, y), and its radius animates from 0 to d
- Bezier specification: P1 (0.25, 0.1) and P2 (0.25, 1)

Figure 16. Specification of the transition between tabs.

3.4.2 Implementation

The whole UI of this screen, including the bottom navigation bar and the content, can be hosted inside a fragment, **TabsFragment** that belongs to the **tabs** module. The content of each tab can be encapsulated into child fragments to reduce the parent's responsibility. All tabs may be hosted under **TabsFragment**, but it is not scalable because, in the future, there might be more tabs introduced. Having all logic under one fragment makes the code less comprehensive and less maintainable.

Since each tab's content is under one fragment, it is reasonable to use fragment transition APIs in this case. The first transition is the disappearing tab's exit transition whereas the other three can be the appearing tab's enter transition. In reality, one project

is likely to have several fragment transitions. Therefore, creating a base class for fragment transition in **core-animation** is necessary. Script 19 is the implementation of that abstract class.

```
# ./core-animation/src/main/.../core_animation/transition/FragmentTransition.kt
abstract class FragmentTransition : Transition() {

    protected lateinit var fragment: Fragment

    override fun captureEndValues(transitionValues: TransitionValues) {}
    override fun captureStartValues(transitionValues: TransitionValues) {}

    override fun isTransitionRequired(startValues: TransitionValues?, endValues:
TransitionValues?) = true

    @CallSuper override fun createAnimator(sceneRoot: ViewGroup, startValues:
TransitionValues?, endValues: TransitionValues?): Animator? {
        val fragmentRoot = fragment.view
        return fragmentRoot?.let { createFragmentAnimator(it) } ?: return null
    }

    abstract fun createFragmentAnimator(fragmentRoot: View): Animator

    @CallSuper
    open fun integrateWithFragment(fragment: Fragment) {
        this.fragment = fragment
    }

}
```

Script 19. Implementation of **FragmentTransition**.

By default, a fragment transition does not require any value capturing. Thus, the implementations of *captureEndValues* and *captureStartValues* are empty. The subclass may change this if necessary. However, because of those empty functions, the Android system may understand that the transition can be skipped. As a result, the transition does not visually happen. **FragmentTransition** can prevent that behavior by enforcing *isTransitionRequired* to always return true. The *createAnimator* function will create the animator that runs when the transition happens. It passes the root view of the fragment to *createFragmentAnimator*, which is overridden by subclasses. This function is essential for creating customized transitions. Last but not least, integrateWithFragment is an open function that determines the relation between the transition and the fragment. The default implementation is simply setting the fragment as a property of the transition.

Since enter transition and exit transition have distinctive characteristics, it is also possible to create base classes for them in the **core-animation** module. They should directly inherit from **FragmentTransition** as shown in script 20.

```
# ./core-animation/src/main/.../core_animation/transition/FragmentEnterTransition.kt
abstract class FragmentEnterTransition : FragmentTransition() {
    override fun createAnimator(sceneRoot: ViewGroup, startValues: TransitionValues?,
        endValues: TransitionValues?): Animator? {
        val animator = super.createAnimator(sceneRoot, startValues, endValues) ?: return
null
        animator.addListener(
            onStart = { fragment.view?.bringToFront() },
            onEnd = { fragment.enterTransition = null }
        )
        return animator
    }

    @CallSuper override fun integrateWithFragment(fragment: Fragment) {
        super.integrateWithFragment(fragment)
        fragment.enterTransition = this
    }
}

# ./core-animation/src/main/.../core_animation/transition/FragmentExitTransition.kt
abstract class FragmentExitTransition : FragmentTransition() {
    override fun createAnimator(sceneRoot: ViewGroup, startValues: TransitionValues?,
        endValues: TransitionValues?): Animator? {
        val animator = super.createAnimator(sceneRoot, startValues, endValues) ?: return
null
        animator.addListener(
            onStart = { fragment.view?.isVisible = true },
            onEnd = { fragment.exitTransition = null }
        )
        return animator
    }

    @CallSuper override fun integrateWithFragment(fragment: Fragment) {
        super.integrateWithFragment(fragment)
        fragment.exitTransition = this
    }
}
```

Script 20. Implementation of **FragmentEnterTransition** and **FragmentExitTransition**.

Both classes override *integrateWithFragment* to set up the transition for the fragment and override *createAnimator* to add certain callbacks to the animator. For **FragmentEnterTransition**, it brings the fragment's root to the front so that the disappearing fragment does not overlap the appearing fragment. It also clears the enter transition of the fragment after the transition finished to prevent memory leak due to cyclic reference between the transition and the fragment. On the other hand, **FragmentExitTransition** forces the fragment's root view to be visible before running the animation. By default, it is hidden by the fragment manager.

At this stage, the code for switching between tabs' fragments is still not ready. This block of code is another candidate to go to **core-animation** because it applies transitions to fragments and can be reutilized. An extension function on **FragmentManager** is rational in this case.

```
# ./core-animation/src/main/.../core_animation/fragment/FragmentNavigationHelper.kt
fun FragmentManager.navigateToTab(
    tag: String,IdRes rootId: Int, appearingFragmentFactory: () -> Fragment,
    enterTransition: FragmentEnterTransition, exitTransition: FragmentExitTransition
) {
    val (appearingFragment, newlyCreated) = findFragmentByTag(tag)?.let { it to false }
        ?: enterFragmentFactory() to true
    enterTransition.integrateWithFragment(appearingFragment)
    val disappearingFragment = fragments.firstOrNull { it.isVisible }
    if (disappearingFragment != null)
        exitTransition.integrateWithFragment(disappearingFragment)
    commit {
        if (newlyCreated) add(rootId, appearingFragment, tag)
        else show(appearingFragment)
        disappearingFragment?.let { hide(it) }
    }
}
```

Script 21. Implementation of *navigateToTab*.

The function takes a *tag* as a parameter to determine if the *appearingFragment* exists or not. If it does not, it is created using the *appearingFragmentFactory* callback and added to the view with the id of *rootId*. Otherwise, the fragment manager can just show the existing fragment instead of creating a new one. In either case, the *enterTransition* is applied to the *appearingFragment*. About the *disappearingFragment*, it is easily found because it is the only visible fragment before the transition happens. The extension function integrates the *exitTransition* to the *disappearingFragment* and simply hides it. It is important to hide the fragment instead of removing it because hiding the fragment helps the fragment manager find and reuse it while removing enforces creating a new tab's fragment every time the user switches back to that tab, which is wasteful.

With the implementation of base classes in place, it is straight forward to create custom transitions by inheriting the correct class. For example, the exit transition of the disappearing's tab can be written and placed in the **tabs** module as in script 22.

```
# ./tabs/src/main/java/la/me/leo/tabs/transition/MainTabsPopTransition.kt
internal class MainTabsPopTransition : FragmentExitTransition() {

    override fun createFragmentAnimator(fragmentRoot: View): Animator {
        val animator = constructAnimator(
            duration = 300,
            onUpdate = { fragmentRoot.alpha = 1f - it },
            onEnd = { fragmentRoot.alpha = 1f }
        )
        fragment.lifecycle.cancelAnimatorOnDestroy(animator)
        return animator
    }
}
```

Script 22. Implementation of **MainTabsPopTransition**.

However, the enter transition for the appearing tab is more complicated because it has dynamic parameters. The parameters *x* and *y* should be provided by the **TabsFragment** as it has information of the tab's position. The parameter *d* can be calculated in the transition class by applying the Pythagorean Theorem: $d = \sqrt{x^2 + y^2}$. When all parameters are resolved, three motions are written and played together using **AnimatorSet**.

```
# ./tabs/src/main/java/la/me/leo/tabs/transition/MainTabsPushTransition.kt
internal class MainTabsPushTransition(private val x: Int, private val y: Int) :
         FragmentEnterTransition() {
    override fun createFragmentAnimator(fragmentRoot: View): Animator {
        val animator1 = createCircularRevealAnimator(fragmentRoot)
        val animator2 = createTranslationAnimator(fragmentRoot)
        val animatorSet = AnimatorSet()
        animatorSet.playTogether(animator1, animator2)
        animatorSet.interpolator = PathInterpolatorCompat.create(0.25f, 0.1f, 0.25f, 1f)
        fragment.lifecycle.cancelAnimatorOnDestroy(animatorSet)
        return animatorSet
    }

    private fun createCircularRevealAnimator(fragmentRoot: View): Animator {
        val d = sqrt(fragmentRoot.height.toFloat().pow(2f) + x.toFloat().pow(2f))
        return ViewAnimationUtils.createCircularReveal(fragmentRoot, x, y, 0f, d)
            .setDuration(300L)
    }

    private fun createTranslationAnimator(fragmentRoot: View): Animator {
        return constructAnimator(
            duration = 300,
            onUpdate = { animatedValue ->
                with(fragmentRoot) {
                    translationX = (1 - animatedValue) * x
                    translationY = (1 - animatedValue) * 16f
                }
            }
        )
    }
}
```

Script 23. Implementation of **MainTabsPushTransition**.

Finally, the **TabsFragment** utilizes *navigateToTab* and the created transitions to bring the desired animations to consumers. As mentioned, it only needs to calculate the *x* and *y* values for the **MainTabsPushTransition**, and those calculations are clearly shown in the following script.

```
# ./tabs/src/main/java/la/me/leo/tabs/TabsFragment.kt
private fun setUpTabs() {
    bottomNavigationView.setOnNavigationItemSelectedListener f@{
        when (it.itemId) {
            R.id.item_discovery -> {
                showFragment(TAG_DISCOVERY, 0) { DiscoveryFragment() }
                return@f true
            }
            R.id.item_deliver -> {
                showFragment(TAG_PROFILE, 1) { DeliveryFragment() }
```

```
                return@f true
            }
        }
        return@f false
    }
}

private fun showFragment(tag: String, tabIndex: Int, fragmentFactory: () -> Fragment) {
    val tabCount = bottomNavigationView.menu.size()
    val x = ((tabIndex + 0.5f) / tabCount * bottomNavigationView.width).toInt()
    val y = (bottomNavigationView.bottom + bottomNavigationView.top) / 2
    childFragmentManager.navigateToTab(
        tag, R.id.fragmentRoot, fragmentFactory,
        MainTabsPushTransition(x, y), MainTabsPopTransition()
    )
}
```

Script 23. Tabs transition usage in **TabsFragment**.

By extracting the navigation code into a function in the core-animation module, the code in TabsFragment looks shorter and more self-descriptive. More valuable, all future screens with tabs take the same benefit and take less time to implement.

After finishing this transition, **core-animation** has abstract code related to fragment transition. It might take much effort at first to write all these base classes, but as the project grows, it helps to add custom transitions seamlessly and reliably because all necessary setup work is bundled inside the developed base classes. Developers only focus on the animation and transition logic inside the sub-classes. On rare occasions, when they need extra setup work for the transitions, they simply add more code to the base classes and all existing animations shall remain the same.

3.5 Venue detail page scrolling behavior

3.5.1 Specification

Figure 17 demonstrates the scrolling process of the venue detail page. When being scrolled, the UI elements inside the header moves to form a united visual effect. This process is complex and is divided into 6 phases. In each phase, there are different motions that happen simultaneously. Some motions even happen across multiple phases. Therefore, it is essential to write the behavior with carefulness so that it is easy to follow.

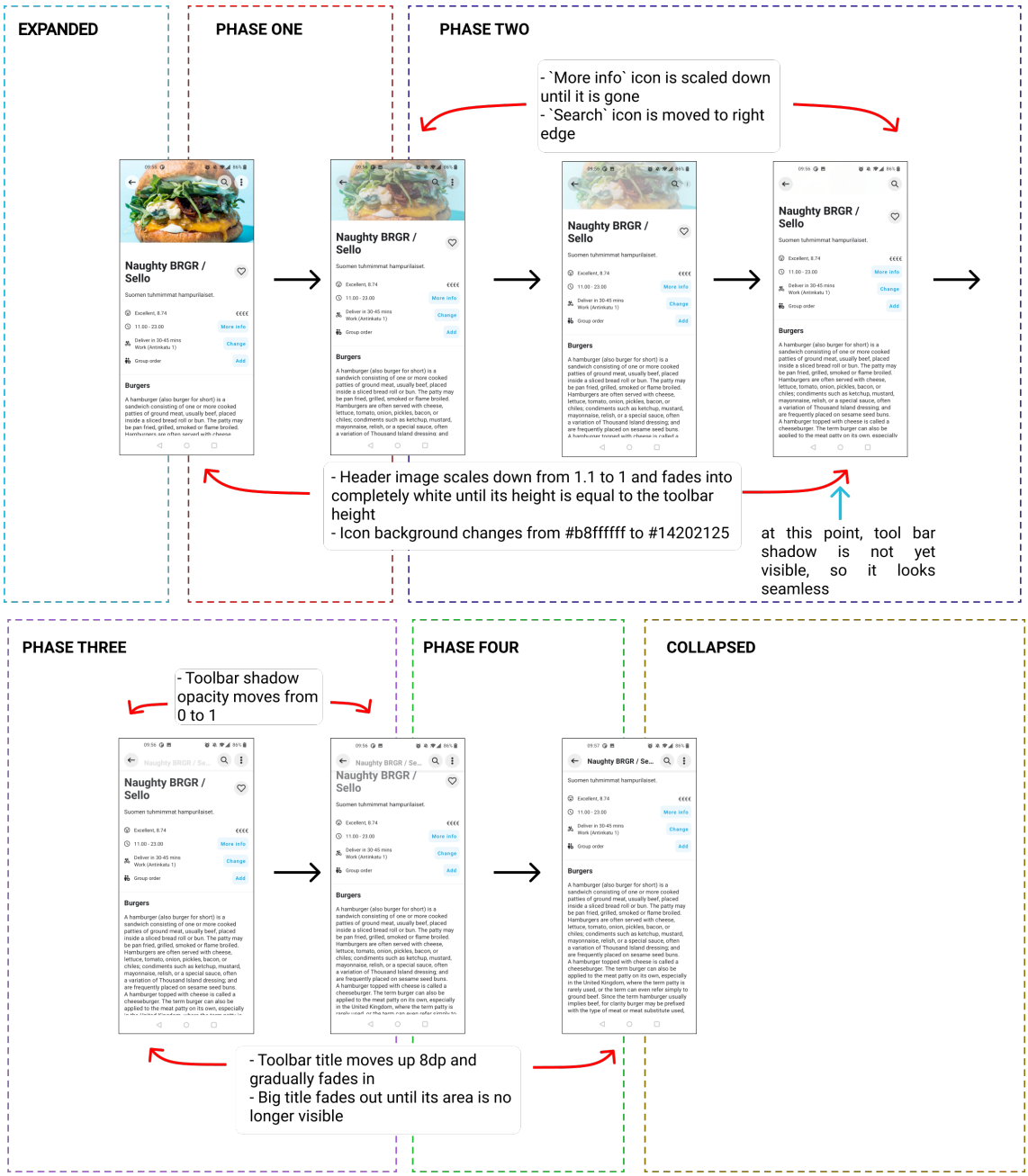


Figure 17. Specification of scrolling behavior in venue page detail.

The first phase and the last phase are self-descriptive. In the expanded phase, the scrolling process has not started, thus, none of the motions run at this phase. During the collapsed phase, the header is fully collapsed, the positions of UI elements settle, thus, the only motion that exists is the moving content.

The four middle phases involve multiple motions. Phase one takes place before the scrolling position reaches half of the image's height. Phase two occurs right after phase

one until the content is scrolled up by exactly the whole image's height. Phase three immediately starts after phase two is finished until the big title of the page meets the toolbar bottom. Last but not least, phase four ends when the big title is completely hidden.

From the technical implementation perspective, it might be easier to analyze the specification based on each motion than analyze it based on each phase. There are exactly 10 motions that can be taken from the requirement:

The image's opacity changes from 1 to 0 during phase 1 and phase 2 (1).

The image's scale changes from 1.1 to 1 during phase 1 and phase 2 (2).

The background color of toolbar icons (*back* icon, *search* icon, and *more info* icon) changes from #B8FFFFFF to #14202125 during phase 1 and phase 2 (3).

The *more info* icon's scale changes from 1 to 0 during phase 2 (4).

The *more info* icon's opacity changes from 1 to 0 during phase 2 (5).

The *search* icon moves to the right and takes the *more info* icon's position during phase 2 (6).

The toolbar background's opacity changes from 0 to 1 during phase 3 (7).

The toolbar title's opacity changes from 0 to 1 during phase 3 and phase 4 (8).

The toolbar title moves up by 8dp during phase 3 and phase 4 (9).

The big title's opacity changes from 1 to 0 during phase 3 and phase 4 (10).

3.5.2 Implementation

This visual effect is special since it does not have a duration. Its appearance completely depends on the scrolling position of the content. Because of that reason, it is impossible to use Android Animator and Transition APIs to achieve the goal. However, the Android

framework allows developers to subscribe to the change in position of scrolling content. It helps render the elements correctly based on the scrolling position. A simple solution is to write the custom header view by inheriting the class **View** and subscribe to the position change inside the implementation. However, browsing the original Wolt application shows that there are several collapsing elements with different graphical effects; it is wiser to create a base class for all collapsing elements. Script 24 contains the simple implementation of that base class called **CollapsingWidget**. This class, of course, belongs to the **core-animation** module for reusability.

```
# ./core-animation/src/main/.../core_animation/CollapsingWidget.kt
abstract class CollapsingWidget(context: Context, attrs: AttributeSet)
         : ConstraintLayout(context, attrs) {

    protected lateinit var scrollView: NestedScrollView


    fun bind(scrollView: NestedScrollView) {
        this.scrollView = scrollView
        scrollView.setOnScrollChangeListener { _, _, scrollY, _, _ ->
            onScrollPositionChanged(scrollY.toFloat())
        }
    }

    protected abstract fun onScrollPositionChanged(scrollY: Float)
}
```

Script 24. Implementation of **CollapsingWidget**.

The *bind* function is exposed so that an activity or a fragment can tell the collapsing view, which scroll view's scrolling position, is interested and react accordingly. The function also stores the scroll view as a property of the **CollapsingWidget** in case subclasses need to access it.

The base class also has one abstract function that requires all children to override called onScrollPostionChanged. This function determines how the elements transform for a particular scrolling position as **CollapsingWidget** attached it to the *scrollView*'s listener.

At this point, the developer can start implementing the custom header element to achieve the required scrolling behavior. In this example, it is called **VenueCollapsingImageWidget**, a class that inherits **CollapsingWidget**. Its skeleton is included in script 25. Since this view is only used in the venue detail page, it should be placed in the **venue-detail-page** module.

```
# ./venue-detail-page/src/.../venue_detail_page/widget/VenueCollapsingImageWidget.kt

internal class VenueCollapsingImageWidget(context: Context, attrs: AttributeSet)

        : CollapsingWidget(context, attrs) {


    private var currentPhase: Phase = Phase.EXPANDED


    override fun onScrollPositionChanged(scrollY: Float) {
        val collapsingDistance = calcCollapsingDistance()
        determineCurrentPhase(scrollY, collapsingDistance)
        renderImage(scrollY, collapsingDistance)
        renderIconBackground(scrollY, collapsingDistance)
        renderMoreInfoIconSize(scrollY, collapsingDistance)
        renderSearchIconPosition(scrollY, collapsingDistance)
        renderTitles(scrollY, collapsingDistance)
        renderToolbarBackground(scrollY, collapsingDistance)
    }

    private fun determineCurrentPhase(scrollY: Float, collapsingDistance: Float) {}

    // Handle motion (1) and (2)
    private fun renderImage(scrollY: Float, collapsingDistance: Float) {}

    // Handle motion (3)
    private fun renderIconBackground(scrollY: Float, collapsingDistance: Float) {}

    // Handle motion (4) and (5)
    private fun renderMoreInfoIconSize(scrollY: Float, collapsingDistance: Float) {}

    // Handle motion (6)
    private fun renderSearchIconPosition(scrollY: Float, collapsingDistance: Float) {}

    // Handle motion (7)
    private fun renderToolbarBackground(scrollY: Float, collapsingDistance: Float) {}

    // Handle motion (8), (9) and (10)
    private fun renderTitles(scrollY: Float, collapsingDistance: Float) {}

    enum class Phase { EXPANDED, ONE, TWO, THREE, FOUR, COLLAPSED }

}
```

Script 25. Structure of **VenueCollapsingImageWidget**.

Inside the class, there is an enum class defined, **Phase**. It represents the phases in the specifications and is stored inside the view under the property *currentPhase*. As mentioned, since **VenueCollapsingImageWidget** is a subclass of **CollapsingWidget**, it has to override *onScrollPositionChanged* and adjust its element inside the implementation of the function. Basically, what happens there is that the view tries to determine the current phase based on the scrolling position and the collapsing distance, or the image's height, by calling *determineCurrentPhase* function. After that step, it renders the elements based on that phase, the scrolling position, and the collapsing distance by calling all render methods. Each render method will handle the motions that

are related to at least one element. For instance, *renderImage* handles motion (1) and motion (2) because they both require changes in the image element.

The most important logic inside this view is determining phase logic. If the phase is wrongly determined, the elements are all rendered incorrectly. That logic is written in script 26 following the specification analysis in the previous section.

```
private fun determineCurrentPhase(scrollY: Float, collapsingDistance: Float) {
    val phase3ScrollRange = context.getDimen(R.dimen.u3)
    currentPhase = when {
        scrollY == 0f -> Phase.EXPANDED
        scrollY < collapsingDistance / 2 -> Phase.ONE
        scrollY < collapsingDistance -> Phase.TWO
        scrollY < collapsingDistance + phase3ScrollRange -> Phase.THREE
        scrollY < collapsingDistance + phase3ScrollRange + (bigTitle?.height ?: 0) ->
Phase.FOUR
        else -> Phase.COLLAPSED
    }
}
```

Script 26. Determining phase logic for **VenueCollapsingImageWidget**.

Thanks to Kotlin *when* expression, the function looks short and precise. If the project is written in Java, there will be many *if-else* chains, which reduce readability.

The last step in this implementation is to write the code for rendering UI elements inside the header based on phase and scrolling position. With the specification analysis step, this becomes relatively straight-forward. Script 27 shows how to follow the design requirements for those elements.

```
// Handle motion (1) and (2)
private fun renderImage(scrollY: Float, collapsingDistance: Float) {
    val progress = when(currentPhase) {
        Phase.EXPANDED -> 0f
        Phase.ONE, Phase.TWO -> scrollY / collapsingDistance
        else -> 1f
    }
    val ivImageScale0 = 1.1f
    val ivImageScale1 = 1.0f
    val ivImageScale = (ivImageScale0 - ivImageScale1) * (1f - progress) + ivImageScale1
    binding.ivImage.scaleX = ivImageScale
    binding.ivImage.scaleY = ivImageScale
    binding.ivImage.alpha = 1f - progress
}

// Handle motion (3)
private fun renderIconBackground(scrollY: Float, collapsingDistance: Float) {
    val bgProgress = when(currentPhase) {
        Phase.EXPANDED -> 0f
        Phase.ONE, Phase.TWO -> scrollY / collapsingDistance
        else -> 1f
    }
    val iconBgColor =
```

```
        argbEvaluator.evaluate(bgProgress, expandedIconBgColor, collapsedIconBgColor) as
Int
    binding.leftIconWidget.backgroundCircleColor = iconBgColor
    binding.rightIconWidget1.backgroundCircleColor = iconBgColor
    binding.rightIconWidget2.backgroundCircleColor = iconBgColor
}

// Handle motion (4) and (5)
private fun renderMoreInfoIconSize(scrollY: Float, collapsingDistance: Float) {
    val progress = when(currentPhase) {
        Phase.EXPANDED, Phase.ONE -> 1f
        Phase.TWO -> 1 - (scrollY - collapsingDistance / 2) / (collapsingDistance / 2)
        else -> 0f
    }
    binding.rightIconWidget1.alpha = progress
    binding.rightIconWidget1.size = (binding.rightIconWidget2.size * progress).toInt()
}

// Handle motion (6)
private fun renderSearchIconPosition(scrollY: Float, collapsingDistance: Float) {
    val progress = when(currentPhase) {
        Phase.EXPANDED, Phase.ONE -> 1f
        Phase.TWO -> 1 - (scrollY - collapsingDistance / 2) / (collapsingDistance / 2)
        else -> 0f
    }
    val marginEnd0 = context.getDimen(R.dimen.u8)
    val marginEnd1 = context.getDimen(R.dimen.u2)
    val marginEnd = (marginEnd0 - marginEnd1) * progress + marginEnd1
    binding.rightIconWidget2.updateLayoutParams<LayoutParams> {
        updateMarginsRelative(end = marginEnd.toInt())
    }
}

// Handle motion (7)
private fun renderToolbarBackground(scrollY: Float, collapsingDistance: Float) {
    val alpha = when(currentPhase) {
        Phase.EXPANDED, Phase.ONE, Phase.TWO -> 0f
        Phase.THREE -> (scrollY - collapsingDistance) / context.getDimen(R.dimen.u3)
        else -> 1f
    }
    binding.flToolbarBgContainer.alpha = alpha
}

// Handle motion (8), (9) and (10)
private fun renderTitles(scrollY: Float, collapsingDistance: Float) {
    val progress = when(currentPhase) {
        Phase.EXPANDED, Phase.ONE, Phase.TWO -> 0f
        Phase.THREE, Phase.FOUR -> (scrollY - collapsingDistance) / phase34ScrollRange
        Phase.COLLAPSED -> 1f
    }
    val translationY0 = context.getDimen(R.dimen.u1)
    val translationY = (1f - progress) * translationY0
    binding.tvTitle.translationY = translationY
    binding.tvTitle.alpha = progress
    bigTitle?.alpha = 1f - progress
}
```

Script 27. Render elements logic for **VenueCollapsingImageWidget**.

The code that renders elements are well-structured and split into small functions. Complex calculations become understandable by using meaningful variable names. These actions are done so that future adjustments in this class can be done fast. The implementation of this custom view is done at this point. There is still code that is used

to assign the image to render, the venue's title, or click listeners for the icons, but they are minor and out of scope as this paper focuses mainly on animation code.

This class is used in **VenueDetailPageFragment** as shown in script 28.

```
# venue-detail-page/src/main/java/la/me/leo/venue_detail_page/VenueDetailPageFragment.kt
class VenueDetailPageFragment : Fragment() {

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        setupToolbar()
    }

    private fun setupToolbar() = venueCollapsingImageWidget.bind(scrollView)
}
```

Script 28. A usage of **VenueCollapsingImageWidget**.

By exercising this scrolling behavior, **core-animation** gains an abstract class for creating collapsing views, which is common in mobile applications. The current implementation is only compatible with **NestedScrollView**. Whenever needed, the developer should extend the functionality to also support **RecyclerView**, which is another view with scrolling behavior in Android development.

4 Analysis

After the implementation, the repository gains a special module, **core-animation** that helps writing animations easier and cleaner. It acts as a base of all animation code in the project. Nevertheless, a perfect codebase does not exist. Therefore, it is worth analyzing the result from different aspects to benchmark it precisely.

The result shows several benefits. As mentioned, reusable animation components are created and encapsulated inside **core-animation**. These components may take time to implement at first, but once they are finished, it is possible to reuse them across the repository. In a company with some mobile applications, the module can also be reused across different projects. The module is technically reliable since it is tested before; it saves time and effort to create similar animations for both the development process and the testing process in the future. Reusability also brings another advantage for engineers. They write shorter code, which is more readable and more comprehensible. It is significant especially when others review the code or adjust the code later. Thirdly, this module is also extendable. In section 3.4, when building the transition, a hierarchy tree of transition classes is created. Starting at the root is the **FragmentTransition**, the superclass of two direct children, **FragmentEnterTransition** and **FragmentExitTransition**. They are two abstract ancestors of **MainTabsPushTransition**, **MainTabsPopTransition**, and other transitions that will be introduced. However, in the future, if the two ancestors cannot satisfy a requirement, for example, reenter transition, a transition happens on the previous screen when the user goes back from the current screen, it is simple to introduce another superclass in **core-animation**, called **FragmentReenterTransition** that extends **FragmentTransition** and fulfill the use case as shown in figure 18.

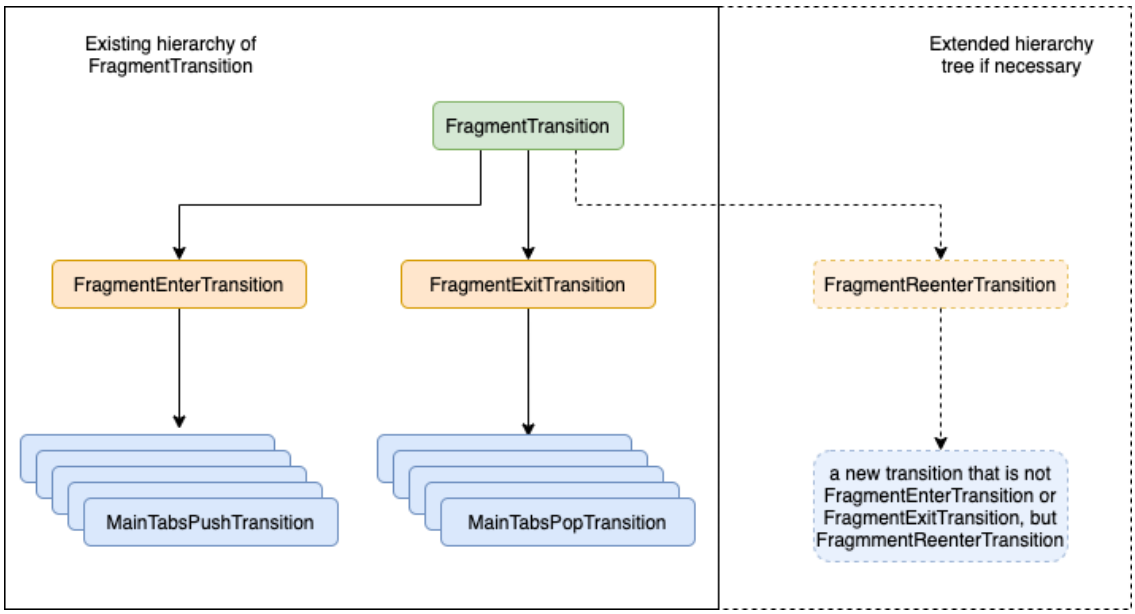


Figure 18. Visualized diagram that describes the extendability of **FragmentTransition**.

The last benefit of encapsulating the base animation code inside a dedicated module is that it can be a candidate for an open-source project. Exposing the module to the community means the code gains the attention of interested developers. With more contributors, bugs are spotted more effortlessly and faster. They also add more features to the repository and enhances its completeness.

On the contrary, there exist minor problems in this approach. Firstly, when new members join the team, it takes time for them to familiarize themselves with the module **core-animation** and its feature. Even though the module is built on top of Android animator APIs and transition APIs, it is still not a standard Android API that is widely known among the developers. One possible solution is having short but precise documentation for the module so that it is neither overwhelming to absorb nor useless to understand. The document needs to be updated whenever a developer adds a new function or updates a function inside the module. Secondly, having common functions in **core-animation** does not guarantee a completely maintainable codebase. It is only the prerequisite for such a codebase as it simplifies and shortens the creation of animations and the control of their callbacks. Nevertheless, developers are required to organize their code tidily in the call sites. In the example of onboarding animation (section 3.3), *createBottomViewsAnimatorSet* and *setupLottieAnimation* still stays in a specific

fragment. Those functions are ones that require engineers to construct nicely at the UI level.

Occasionally, difficulties arise when attempting to generalize code or wrap the existing animation API inside a simpler programming contract. In the case of venue detail page scrolling behavior (section 3.5), the developer needs to perform perplexing calculations to determine different phases of the motion. This is unavoidable because animation is partly based on mathematics. Those calculations are intricate that they make the code less understandable. During the implementation of that behavior, the motion is divided into as small as possible phases so that the mathematic formulas are found easier. The code is also broken into corresponding phases to increase the obviousness. Another challenge comes when *navigateToTab* (section 3.4) is implemented. The function is designed to take enter transition and exit transition as parameters; otherwise, all call sites need to assign the transitions to fragment, which is error-prone when a developer forgets to do that. Therefore, **FragmentTransition** class is introduced to build transitions. Its instances are directly passed to *navigateToTab*. However, to build the necessary animator, it needs to know the fundamental component of animation, the animated element. In this case, it is the fragment. Giving a fragment's reference to the transition is manageable, but the problem is that the fragment also has the reference of the transition. In Kotlin, this is called a circular reference, which caused memory leaks (Dehghani, 2020). Technically, this is unacceptable for any application. To overcome this bug, the transition's reference in the fragment is removed after the animation finishes by using the animator's callback in **FragmentTransition**'s direct children.

Figure 19. References between **Fragment** and **FragmentTransition** throughout animation lifecycle.

Despite the existence of minor issues, the approach significantly helps to achieve the goal of building a constructive and maintainable project. The source code is simple, reusable, understandable, and divided into specialized modules. In addition to that cleanliness, the biggest success is that all three complex animations are replicated, which proves the usefulness and practicality of the method in the Android development area. Ultimately, the outcome meets the goal of the project.

5 Conclusion

The priority of the project is to build an Android codebase from scratch that highly supports customized and maintainable animations. By recreating a few animations from a real-life application, Wolt: Delivery Food, this thesis has shown how animation code can be written efficiently and cleanly to achieve motion designer's requests and decrease later difficulties in maintenance work. The work covers numerous native Android animation APIs' usages and other ways to construct animations. Most importantly, after these animations are rebuilt, they leave the project with convenient functionalities in the **core-animation** module to accelerate the development of incoming animations in the project and a perspicuous animation codebase for collaborators.

Although the three animations require the application of multiple techniques, it is impossible to achieve 100% coverage for all kinds of animations because the imagination and the creativity of humans do not have any limitations. It is normal if the existing code cannot fulfill a specification afterward. When that time comes, the engineer has to think of another way.

Even if animating UI elements in Android is hard due to the framework's 11 years of evolution, the basic understanding of Android animations APIs is a prerequisite. When the vital core is mastered, it is recommended to evaluate maintainability thoroughly, and the concrete implementation can come later. Code that might be reused should be extracted to a common module, similar to **core-animation** in this project. Not only does it apply to animation work, but that attitude should also be considered in the daily work of a software engineer. It strengthens software-designing skills and helps developers transform complex logic such as animation logic into comprehensible source code.

References

Airbnb.io. (2020). Lottie Docs. [online] Available at: https://airbnb.io/lottie/ [Accessed 9 Dec. 2020].

Andreas, B., Tor-Morten, G. & Ghinea, G. (2019). Animations in Cross-Platform Mobile Applications: An Evaluation of Tools, Metrics and Performance. Sensors (Basel, Switzerland), 19.

Android Developers. (2019a). Animations and Transitions. [online] Available at: https://developer.android.com/training/animation [Accessed 23 September 2020].

Android Developers. (2019b). Create a custom transition animation. [online] Available at: https://developer.android.com/training/transitions/custom-transitions [Accessed 17 Nov. 2020].

Android Developers. (2019c). Fragments. [online] Available at: https://developer.android.com/guide/componentsb/fragments [Accessed 7 Nov. 2020].

Android Developers. (2019d). Reveal or Hide a View Using Animation. [online] Available at: https://developer.android.com/training/animation/reveal-or-hide-view [Accessed 1 October 2020].

Android Developers. (2020a). Activity. [online] Available at: https://developer.android.com/reference/android/app/Activity [Accessed 11 Oct. 2020].

Android Developers. (2020b). Animate layout changes using a transition. [online] Available at: https://developer.android.com/training/transitions [Accessed 21 Nov. 2020].

Android Developers. (2020c). Jetpack Compose. [online] Available at: https://developer.android.com/jetpack/compose [Accessed 6 Dec. 2020].

Android Developers. (2020d). Navigate between fragments using animations. [online] Available at: https://developer.android.com/training/basics/fragments/animate [Accessed 21 Nov. 2020].

Android Developers. (2020e). Optimize Your Build Speed. [online] Available at: https://developer.android.com/studio/build/optimize-your-build#optimize [Accessed 3 Dec. 2020].

Android Developers. (2020f). Start an activity using an animation. [online] Available at: https://developer.android.com/training/transitions/start-activity [Accessed 16 Nov. 2020].

Android Developers. (2020g). View. [online] Available at: https://developer.android.com/reference/android/view/View?hl=en [Accessed 7 Nov. 2020].

Android Developers. (2021a). Kotlin and Android. [online] Available at: https://developer.android.com/kotlin [Accessed 16 Jan. 2021].

Android Developers. (2021b). Reduce your app size. [online] Available at: https://developer.android.com/topic/performance/reduce-apk-size [Accessed 13 Mar. 2021].

Barclay, C. (2019). 5 Steps for Systematizing Motion Design. [online] DesignSystems.com. Available at: https://www.designsystems.com/5-steps-for-including-motion-design-in-your-system/ [Accessed 6 October 2020].

Carullo, G. (2020). Implementing Effective Code Reviews : How to Build and Maintain Clean Code. Apress.

Dehghani, A. (2020). Garbage Collection and Cyclic References in Java | Baeldung. [online] Baeldung. Available at: https://www.baeldung.com/java-gc-cyclic-references [Accessed 16 Mar. 2021].

Dragicevic, P., Bezerianos, A., Javed, W., Elmqvist, N. & Fekete, J.-D. (2011). Temporal distortion for animated transitions. Proceedings of the 2011 annual conference on Human factors in computing systems - CHI ’11.

Edwards, S. (2018). Gradle Dependency Management: Using Kotlin and buildSrc for build.gradle Autocomplete in Android Studio. [online] Caster.io. Available at: https://caster.io/lessons/gradle-dependency-management-using-kotlin-and-buildsrc-for-buildgradle-autocomplete-in-android-studio [Accessed 5 Dec. 2020].

Elye (2020). Which Android Animator to Use? - Mobile App Development Publication - Medium. [online] Medium. Available at: https://medium.com/mobile-app-development-publication/which-android-animator-to-use-ced54e21d317 [Accessed 8 Nov. 2020].

Gradle.org. (2020). Declaring Dependencies between Subprojects. [online] Available at: https://docs.gradle.org/current/userguide/declaring_dependencies_between_subprojects.html [Accessed 10 Dec. 2020].

Grafia.fi. (2015). Vuoden Huiput 2015 on valittu - Grafia. [online] Available at: https://www.grafia.fi/ajankohtaista/vuoden-huiput-2015-on-valittu/ [Accessed 12 Dec. 2020].

Hashimi, S. Y., Komatineni, S. & McLean, D. (2010). Pro Android 2. California: Apress.

Hugo (2017). SeeWeather. [online] GitHub. Available at: https://github.com/xcc3641/SeeWeather [Accessed 19 Jan. 2021].

Izdebski, Ł. & Sawicki, D. (2016). Easing Functions in the New Form Based on Bézier Curves. Computer Vision and Graphics, pp.37–48.

Kantola, T. (2017). Transition animations in mobile applications. Master’s thesis. Aalto University. Available at: https://tuukka.is/static/sci_2017_tuukka_kantola.pdf [Accessed 03 October 2020].

Kiat, H.C. (2019). FabMenu. [online] GitHub. Available at: https://github.com/cheekiat/FabMenu [Accessed 7 Oct. 2020].

Klimczak, E. (2013). Design for Software. New York: John Wiley & Sons Inc.

Lake, I. (2018). Single Activity: Why, When, and How (Android Dev Summit '18). [online] YouTube. Available at: https://www.youtube.com/watch?v=2k8x8V77CrU [Accessed 7 Nov. 2020].

Liu, D., Butcher, N., Roard, N. & Hoford, J. (2018). Get Animated (Android Dev Summit '18). YouTube. Available at: https://www.youtube.com/watch?v=N_x7SV3I3P0&list=PLmEM3-F7iVMlm936VEVxlITwdn60XJQX0&t=337s [Accessed 8 Nov. 2020].

Material Design (2020a). Speed. [online] Material Design. Available at: https://material.io/design/motion/speed.html [Accessed 5 October 2020].

Material Design (2020b). Understanding Motion. [online] Material Design. Available at: https://material.io/design/motion/understanding-motion.html [Accessed 30 September 2020].

Martin, R.C. (2010). Clean Code a Handbook of Agile Software Craftsmanship. Upper Saddle River [Etc.] Prentice-Hall.

Mathis, L. (2016). Designed for Use, 2nd ed. Texas: The Pragmatic Bookshelf.

Microsoft (2018). Animations and Transitions - Win32 apps. [online] Microsoft.com. Available at: https://docs.microsoft.com/en-us/windows/win32/uxguide/vis-animations [Accessed 6 October 2020].

Nielsen Norman Group. (2020). Executing UX Animations: Duration and Motion Characteristics. [online] Available at: https://www.nngroup.com/articles/animation-duration/ [Accessed 7 Oct. 2020].

Penner, R.C. (2002). Robert Penner's Programming Macromedia Flash MX. New York: McGraw-Hill/Osborne.

PROTOIO Inc. (2014). Easings. [online] Available at: https://support.proto.io/hc/en-us/articles/115001466372-Easings [Accessed 3 October 2020].

Spitsin, M. (2017). How simple animation can be a big problem. [online] Medium. Available at: https://programmerr47.medium.com/how-simple-animation-can-be-a-big-problem-ca206f6a8059 [Accessed 7 Nov. 2020].

MaaS Global (2011). Whim - All your journeys. [application] Google Play Store. Available at: https://play.google.com/store/apps/details?id=global.maas.whim [Accessed 7 Oct. 2020].

Ramotion (2020). paper-onboarding-android. [online] GitHub. Available at: https://github.com/Ramotion/paper-onboarding-android [Accessed 13 Mar. 2021].

Visser, J., Rigal, S., van der Leek, R., Wijnholds, G. & van Eck, P. (2016). Building Maintainable Software, Java Edition: Ten Guidelines for Future-proof Code. O'Reilly.

Willenskomer, I. (2017). Creating Usability with Motion: The UX in Motion Manifesto. [online] Medium. Available at: https://medium.com/ux-in-motion/creating-usability-with-motion-the-ux-in-motion-manifesto-a87a4584ddc [Accessed 7 Oct. 2020].

Wolt Enterprises Oy (2015a). Wolt: Food delivery. [application] App Store. Available at: https://apps.apple.com/us/app/wolt-food-delivery/id943905271 [Accessed 24 September 2020].

Wolt Enterprises Oy (2015b). Wolt: Food delivery. [application] Google Play Store. Available at: https://play.google.com/store/apps/details?id=com.wolt.android [Accessed 24 September 2020].

Yuen, S. (2017). Mastering Windows Presentation Foundation. Packt Publishing.

Zalando SE (2011). Zalando Lounge - Shopping Club. [application] Google Play Store. Available at: https://play.google.com/store/apps/details?id=de.zalando.lounge [Accessed 7 Oct. 2020].