Jetpack Compose for Android Automotive

BACHELOR THESIS

Tobias Schmid

Submitted on 30 October 2023

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Christian Schrödel
Prof. Dr. Dirk Riehle, M.B.A.

Friedrich-Alexander-Universität
Faculty of Engineering

Declaration of Originality

License

Acknowledgements

I express my sincere gratitude to Christian for his invaluable guidance, unwavering support, and expert advice throughout this journey.

Additionally, I acknowledge the use of DeepL in improving the language quality of this thesis.

Finally, I would like to thank all the proofreaders for their valuable time and their impact on improving this thesis.

Abstract

The automotive industry has changed in recent years, evolving from conventional vehicles to intelligent, software-driven machines. As a result, Android has released its own automotive operating system. In 2018, Google released the first version of Jetpack Compose. A new UI framework that aims to revolutionize the way Android UI is declared. Jetpack Compose promises a lot of new features and a better development experience. However, migrating a large application or set of applications from one framework to another can be a major undertaking and should be done in a planned, structured, and thoughtful way. This paper discusses the pros and cons of migrating to Jetpack Compose in the context of a large automotive application. As a result, a migration strategy guide will be provided. Additionally, best practices for migrating some specific use cases for automotive software will be covered as well as some common use cases.

Contents

List of Figures

Listings

List of Tables

Acronyms

ADR	Architecture Decision Records
API	Application Programming Interface
APK	Android Package
DSL	Domain-Specific Language
ID	Identification
LOC	Lines of Code
MVVM	Model-View-ViewModel
OEM	Original Equipment Manufacturer
UI	User Interface
UX	User Experience
XML	Extensible Markup Language

1 Introduction

The User Interface (UI) is vital to almost every application, providing a seamless and high-performing User Experience (UX). Traditionally, Android developers have utilized Extensible Markup Language (XML) layouts and the View system to construct UIs. However, as applications increase in complexity and larger systems such as automotive systems running on Android emerge, the challenges associated with UI development also increase.

Therefore, Google has released a new UI toolkit called Jetpack Compose to address these challenges. The goal is to replace XML layouts with a declarative approach. With Compose, UI can be crafted entirely with Kotlin, the most popular (‘Android’s Kotlin-first approach’, 2023) and by Google recommended programming language for Android development (Kotlinlang, 2023). The comprehensive Kotlin feature set allows for facile construction of significant scale application UIs by utilizing for example polymorphism and design patterns.

The automotive industry has transformed in recent years, progressing from conventional vehicles to intelligent, connected and software-driven machines. Software now plays a pivotal role in improving vehicle functionality and safety, while also enhancing the UX by creating feature-rich, user-friendly automotive software that is resilient and meets the unique requirements of the automotive ecosystem (‘Software-Defined Vehicles – A Forthcoming Industrial Evolution’, 2021). This transformation does include in-car systems such as driving assistance and self driving but also the infotainment systems. Their importance has grown and is set to further increase in the future (‘How new infotainment will shape the future customer experience’, 2021). The transition to Jetpack Compose signifies a noteworthy progression in the pursuit of creating highly dynamic, responsive, and sustainable UI. Nonetheless, the process of migration is a complicated undertaking that entails overcoming numerous challenges while effectively utilizing the capabilities of Jetpack Compose.

This thesis investigates the migration process of large scale automotive software systems regarding interoperability, the exertion associated with the transition, and design recommendations for standard use cases.

2 Literature review

2.1 Jetpack Compose

Before delving further into Jetpack Compose, it is important to grasp the idea of declarative UI programming. In a declarative UI framework, developers specify the structure, behavior, and state of the UI using a high-level programming language or Domain-Specific Language (DSL). The framework then converts this declarative code into real UI elements and manages their runtime state. This method provides several benefits over an imperative approach, such as enhanced readability, maintainability, and testability of the code (Varon, 2022). By following an imperative approach the creation and update of the UI elements would be described step by step (Afridi, 2023).

Jetpack Compose employs a concise and expressive syntax to define UI-components. Developers utilize composable functions annotated with *@Composable* to create UI-elements. Typically, these functions only outline the structure of the UI-element without any return value. For instance, the code of a basic button in Jetpack Compose may look like this:

```kotlin
@Composable
fun SimpleButton(text: String, onClick: () -> Unit) {
    Button(onClick = onClick) {
        Text(text = text)
    }
}
```

Listing 2.1: A sample Composable

Compose simplifies the creation of reusable UI components. Composables are conventionally declared as top-level functions, making them reusable throughout the application. This approach facilitates maintenance and promotes UI design consistency ('Thinking in Compose', 2023).

Another intriguing element of Jetpack Compose pertains to state management. In the previously used XML based approach, the empty View has to be manually populated with its data after the XML layout was inflated. Moreover, every data update requires the explicit update of the View's data by keeping a reference to the View and altering its values. This problem has already been addressed by introducing databinding, which enables direct connection of XML layouts to LiveData objects, resulting in automatic updates to the UI when new values are emitted. Jetpack Compose further simplifies state management by fully handling the state internally. A state object can be observed by Jetpack Compose in order to update the UI, whenever the value of the state object changes. This process is called recomposition. To ensure state preservation during recomposition, the state must be saved in a non-composable object or remembered using the *remember()*'-function. If a value is enclosed in a *remember*-function, it will not be executed again during recomposition. Additionally, the *rememberSavable*-function allows the value to persist through configuration changes ('State and Jetpack Compose', 2023).

Compose provides a robust theming and styling system, facilitating the development of consistent user interfaces. Themes can be defined at the application, screen, or component level, offering great flexibility in design. Styles are declared by implementing straightforward style classes and can be instantiated and applied to composable elements. Those styles can be bundled to form a theme (J. R. Castillo, 2022). To uniformly style an entire screen or application, it is recommended to utilize CompositionLocal an Application Programming Interface (API) provided by Compose to declare a value once within the UI-tree. This value can then be accessed from any location in the UI-tree under the node in which it was declared, eliminating the need to pass values using explicit parameters ('Locally scoped data with CompositionLocal', 2023).

The Jetpack Compose developers recognized that a sudden transition from the traditional XML and View framework to Compose was not feasible. As a solution, they provide a broad range of interoperability APIs that enable partial or incremental migration.

2.2 Migration motivation

2.2.1 Challenges with XML and Views

XML-based layouts frequently lead to lengthy code, as UI elements and their attributes are defined in separate XML files. This can pose challenges in comprehending and managing the codebase, especially for intricate UIs. Moreover, developers frequently engage in writing redundant boilerplate code to manually access and manipulate Views. Such code not only clutters the source code but also raises the risk of errors. XML layouts are inherently less reusable, which can hinder the effective encapsulation and reuse of UI components. This limitation may impede productivity and consistency across applications. Additionally, XML layouts lack type safety, resulting in potential runtime errors when incorrect resource references or attribute types are utilized. Such errors can be difficult to catch during the development process.

2.2.2 Advantages of Jetpack Compose

Jetpack Compose employs a declarative syntax, facilitating descriptions of the UI's structure and behavior with concise Kotlin code. Such an approach improves the legibility and maintainability of code. Composables in Jetpack Compose are exceedingly modular and reusable. Developers can encapsulate UI elements into functions, encouraging consistency and decreasing code duplications. Jetpack Compose also furnishes APIs for efficient and anticipatable state management. Composables update automatically when the underlying state changes, simplifying complex UI updates. With Jetpack Compose, UI elements are strongly typed, which reduces the likelihood of runtime errors. Compile-time checks detect many issues before they become runtime problems. Additionally, Jetpack Compose's theming and styling system simplifies the development of visually consistent UIs. Themes can be defined at different levels, facilitating customization of the app's look and feel. The reduction of redundant code, improvement in legibility, and utilization of contemporary development practices augment the efficiency of software engineers, yielding expedited app creation and simplified upkeep.

2.2.3 Preview and LiveEdit

Jetpack Compose introduces powerful developer tools, such as Preview and LiveEdit, that enhance the UI development process significantly. These features provide several essential advantages. Using Preview and LiveEdit, developers can visualize changes they make to the UI instantly and in real-time. This capability speeds up the prototyping phase, enabling quick experimentation and iteration. LiveEdit offers real-time feedback on code modifications, eliminating the requirement for a time-consuming build and deploy cycle. This allows developers to instantly see how their alterations impact the UI, thus reducing development time and increasing productivity ('Iterative code development', 2023). Compose enables the instantiation of multiple Previews with varied configurations simultaneously, allowing for the examination of distinct visual appearances of UIs ('Preview your UI with Composable previews', 2023). The Preview and LiveEdit features also enhance the efficiency of debugging UI issues. The following figure present multiple Previews that preview the same UI code but different preview configurations.

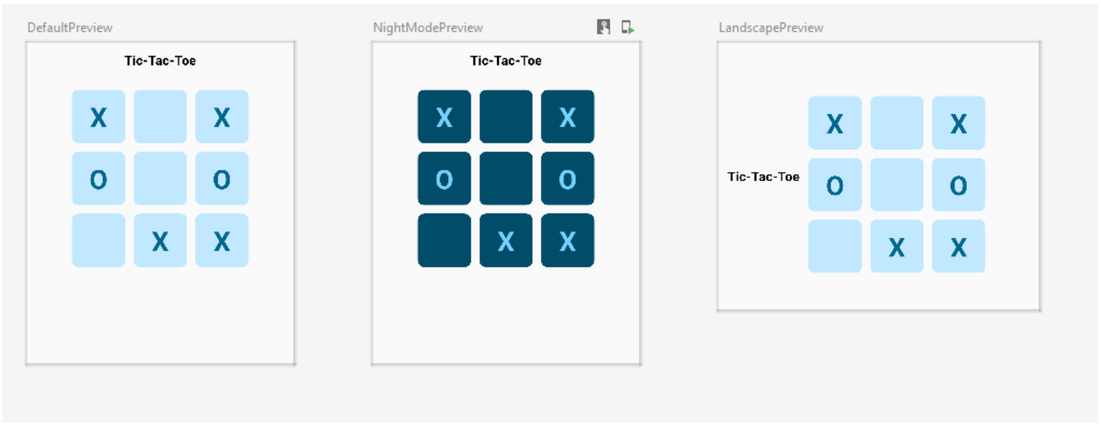


Figure 2.1: Three preview configurations for the same Composable

Developers have the capability to identify problematic components, make necessary adjustments, and instantly observe the results, thereby simplifying the debugging process. The ability to rapidly iterate and visualize changes accelerates the development cycle. Developers can quickly refine the user interface, resulting in a polished and user-friendly interface. The Preview and LiveEdit functionalities transform UI development by providing real-time visualization, immediate feedback, and efficient collaboration. These tools enhance developer productivity and streamline the UI development process.

2.3 Architecture Decision Records

In the field of software architecture and development, it is crucial to maintain clear and documented records of decisions made throughout a project's lifecycle. This is vital for comprehending, managing, and advancing the system. A practice for this is formalized as Architecture Decision Records (ADR)s, which are utilized to capture and communicate significant architectural decisions made during the development of a software system. These records serve as a historical record, offering insight into the reasoning behind crucial design decisions, the circumstances surrounding their creation, and their possible impacts.

ADRs facilitate transparency by providing the entire development team access to architectural decisions. Such transparency ensures that everyone comprehends the reasoning behind decisions, promoting effective contribution to future change-related discussions. Furthermore, ADRs serve as a tool for knowledge preservation, allowing new team members to promptly understand architectural decisions that have shaped the system. They provide a precious resource for onboarding and retaining institutional knowledge. By documenting architectural decisions, ADRs establish accountability. Team members are responsible for their decisions, and ADRs provide clarity on the decision-maker, timing, and rationale. As a project progresses, ADRs facilitate architecture management, providing context for past decisions and aiding in the assessment of proposed changes. After (IcePanel, 2023) a typical ADR includes the following components:

- Title: A concise title that describes the decision being documented.
- Context: The background or context that led to the need for the decision. What problems or challenges were encountered? What factors influenced the decision?
- Decision: The actual decision and details about the proposed solution.
- Consequences: An assessment of the potential consequences of the decision, including both positive and negative impacts. This section helps in understanding the trade-offs involved.
- Status: The current status of the decision, indicating whether it's proposed, accepted, rejected, or superseded by another decision.

Such ADRs will be utilized in the chapter on design and implementation to support and explain the design choices that were made and their corresponding contexts of applicability.

2.4 Arc42

Arc42 (https://arc42.org) is a framework that offers a structured method for documenting software architecture. This framework has been created to resolve the demand for intelligible and consistent documentation of software systems, enabling development teams to understand, communicate, and develop their architecture proficiently. Arc42 is structured with a series of templates and guidelines that support architects and developers in documenting various aspects of their software architecture. It offers a framework for documenting both the technical and conceptual aspects of an architecture, guaranteeing that all relevant parties have access to critical information.

Utilizing arc42 to document software architecture yields several benefits:

- Clarity: arc42 provides a structured and standardized way to document software architecture, making it easier for team members to understand and collaborate on architectural decisions.
- Consistency: The framework ensures that essential aspects of the architecture, such as quality requirements and constraints, are consistently documented.

- Communication: arc42 facilitates effective communication among stakeholders, including architects, developers, testers, and project managers.
- Change management: The documentation serves as a reference for future changes and helps assess the impact of proposed modifications on the architecture.

This framework will be used to document parts of this thesis, but it will not include a complete arc42 documentation because this project only pertains to a migration project and not a full software architecture's documentation.

3 Requirements

The migration of a large-scale application should be meticulously organized, planned, and documented to avoid an untraceable process that could leave errors undetected. An unplanned migration could also result in a lack of a clean architecture. To successfully plan and assess a migration's success, requirements must be established and followed. Some requirements for migrating a large-scale application were collected by conversing with Android UI developers about their aversions to the current UI framework and their expectations for the migration process and resulting software. To clarify and further detail these requirements, dedicated interviews were arranged and carried out. To document the requirements, the first chapter of the arc42 framework "Introduction and Goals" will be used (https://docs.arc42.org/section-1/). This template includes schemes for summarizing and visualizing requirements in a structured and comprehensible manner. This chapter focuses on the requirements for migrating a project to Jetpack Compose.

3.1 Requirements overview

In table 3.1, the requirements for the applications migration process are outlined. This table focuses on the point of view of a software developer.

Requirement	Explanation
Continuous migration	The migration should occur incrementally without interrupting the development of features and the resolution of bugs..
Reduction of boiler-plate code	Writing UI in XML leads to repetitive code, resulting in an increase in lines of code and a decrease in code readability.
Clean architecture	New architecture decisions should be carefully considered to produce a clean and reusable architecture.
Keep separation of concerns	Separating concerns is essential to ensure the code remains both understandable and maintainable.
Improvement of testability of UI code	Testing UI code should be simplified.
Feature usage	Compose should provide developer support features that are equal to or better than those offered by the XML-View framework.

Table 3.1: Requirements

3.2 Quality goals

The following table lists the qualities expected from the application during and after the migration process is executed.

Quality	Description
Performance	At best, the application's performance will improve by avoiding an increase in Android Package (APK) size.
Interoperability	Newly added UI code should be compatible with the rest of the code base.
Transferability	Implemented widgets and support structures should be usable across the entire range of applications.
Visual identity	The application post-migration should maintain the same look and feel as before the migration.

Table 3.2: Quality goals

3.3 Stakeholder

Not only software developers are interested in a switch in the applications UI framework. Therefore table 3.3 lists the most important stakeholders of such a migration project and their corresponding expectations.

Role	Expectations
Customer	Minimal to no decrease in developer productivity occurring in feature development and bug-fixing processes. Development speed should increase as technical debt is reduced.
Software developers	Achieving a streamlined and efficient migration process with a correspondingly enhanced development experience post-migration.
Team lead	The team's productivity must remain stable. Incorporating the latest technologies makes job opportunities more appealing to prospective employers.
Test engineers	Compose should provide improved testability for UI code.
Quality assurance	Improving test coverage can enhance the quality and robustness of the code base. One way to achieve this is by reducing code complexity through the elimination of boilerplate code.
User	The user should not detect any change in the UI framework and any possible improvements should only be reflected in the application's performance.

Table 3.3: Stakeholder expectations

4 Architecture

4.1 State hoisting

State hoisting is a fundamental principle of Jetpack Compose. This principle implies that state is preferably hoisted to the lowest common ancestor. Thus, typically, small Composables do not store their own states. Composables that do not store any state are referred to as stateless, while stateful Composables store state. Additionally, it is advisable to segregate the data from the actual UI widget. Usually, a stateless Composable is wrapped by a stateful Composable, which then provides the stateless Composable with its data through the parameters (J. Castillo, 2022). This approach offers a high level of reusability for the stateless Composables.

4.2 Widget library and application

The UI code for an automobile application typically consists of two primary components. The first project encompasses a widget library, which comprises widgets, styling, and tools that are shared across various applications. The second project involves utilizing the widgets from the shared library to implement the specific application.

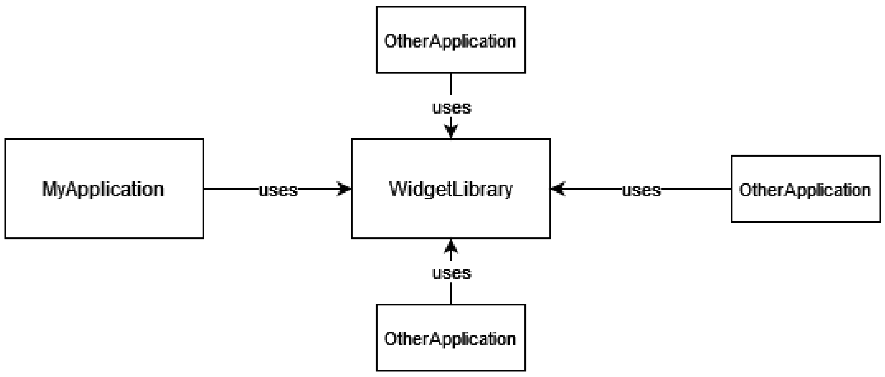


Figure 4.1: Relation between application and widget library (drawn with draw.io)

The separation in projects benefits from state hoisting making it clear which Composables to put inside the library and which to put inside the application. It is recommended that most stateless Composables be integrated into the shared widget library, making them available for use across all applications. Another significant aspect that must be incorporated into the widget library is integrating data structures for applying styling, such as color, dimensions, and commonly used drawables across all Original Equipment Manufacturer (OEM) applications to yield a uniform appearance.

4.3 The MVVM pattern

The application's architecture should adhere to the widely used Model-View-ViewModel (MVVM) design pattern to ensure scalability, maintainability, and testability. MVVM is an architectural design pattern that divides an application into three core components: Model, View, and ViewModel. The Model embodies the data and business logic of the application, which includes manipulation and validation of data and interacting with the data source. The View comprises all UI components of the application and observes the ViewModel. It updates the UI whenever there are changes in the underlying data and propagates user interaction events to the ViewModel. The ViewModel plays the role of an intermediary between the Model and the View. To ensure a clean separation of concerns, the ViewModel conventionally does not have a direct reference to the View (Chugh, 2022).

Using the MVVM pattern comes with many benefits. Using it leads to a clean separation of concerns by separation the data, the UI logic and the UI components, making the codebase more modular and easier to maintain. This modularity also makes it easier to test, since the ViewModels and Models can be tested independently from the UI, ensuring robust and more reliable tests. It also makes the test execution faster, since there is no need to run the tests on an actual or emulated device since they usually do not have any dependency to the UI or the Android system. MVVM also promotes the use of reactive programming principles, making it easier to handle complex UI interactions and asynchronous operations which gets perfected by the usage of state in Jetpack Compose (Chugh, 2022).

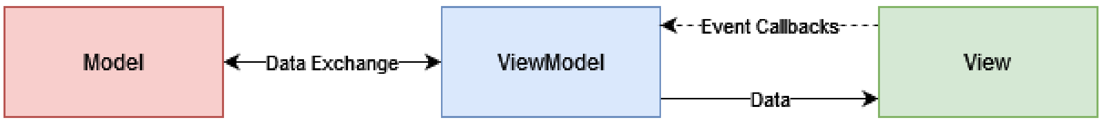


Figure 4.2: The MVVM pattern (drawn with draw.io)

If the MVVM pattern is implemented correctly, the migration to Jetpack Compose can be limited to the Views. The ViewModels and Models can remain untouched since they lack references to the Views. However, ViewModels frequently provide references to LiveData objects for the Views to observe. In contrast, Jetpack Compose works solely with State and not with LiveData. In the long term, it is beneficial to migrate ViewModels from LiveData to State. Meanwhile, Compose provides an interoperability API to convert not only LiveData but also Flows and RxJava Observables to State. Converting a LiveData object to State is as easy as calling the *observeAsState()* method ('State and Jetpack Compose', 2023).

4.4 Build flavors

Automotive applications are frequently designed for multiple OEMs. However, these applications typically only vary in their UI rather than their underlying business logic. To enable the provision of distinct user interfaces for each OEM, the (Android) Gradle plugin features build flavors. These flavors permit the creation of various application versions from the same codebase. Each flavor can possess distinctive code, yet still access code that is shared among all build flavors (Naeem, 2023). This provides an opportunity to share code among OEMs, reducing the number of required tests since the shared code does not need to be tested for each OEM. Additionally, the software becomes more scalable by simply adding a new build flavor for a new OEM. Another significant advantage of this approach is that only the necessary code for each OEM is built into the APK. If various versions were developed using a conditional statement that selects the implementation according to a variable, then there would be a requirement to include code for all OEMs in the APK. This would contribute to an increase in APK size and an elevated risk for bugs.

5 Design and implementation

This section will explore common use cases and challenges in automotive and large-scale software development, following the guidance of chapter 9 in the arc42 framework, specifically the architecture decisions section. To provide context for these use cases and offer suggestions for their successful implementation, this paper will utilize ADRs. It is important to note that while these suggestions are valuable, they are by no means prescriptive or the only viable approach. It is advisable to modify or select an implementation that not only aligns with the particular use case, but also conforms to the subjective programming style preferences.

5.1 Value resources

Context

Google developer's designed Jetpack Compose to be compatible with the old XML resource files. However, with Compose the possibility to store resources have broadened given more flexibility to the developer. An example on how to use XML resources in Compose is given in the following code listing.

```kotlin
@Composable
fun XMLResourceExample() {
    val string = stringResource(R.string.info_home)
    val width = dimenResource(R.dimen.default_square_item)
    val color = colorResource(R.color.primary_blue
    val painter = painterResource(R.drawable.flower_big)
}
```

Listing 5.1: Using XML resources

Decision

The recommended approach is to migrate value resources to Kotlin. This will allow you to take advantage of all the features that Kotlin has to offer. Nested data structures can be created to improve the structure of resources. In addition, resource sets can be extended through inheritance, creating more versatile resource structures. In order to remain interoperable with the traditional Android development while migrating, the Kotlin resources should reference the values from the XML files. The references can be omitted and therefore the value be placed directly into the Kotlin files once the referenced resource is no longer used within XML resources. Those resources should than be provided through a CompositionLocal.

Consequences

Creating nested structures for resources is a departure from the typical one-dimensional resource lists found in XML files, leading to an easier to understand and more organized resource management. By offering the resources as a CompositionLocal they are easily accessible throughout the composition, without requiring additional code, resulting in clean code. Furthermore, utilizing CompositionLocal permits access to the context and configuration within resource classes and enables resources to rely on factors such as the current orientation, UI-mode or language. This replaces the previously utilized resource qualifiers for XML resources.

5.2 Interoperability

Context

As previously mentioned, Google provides numerous interoperability APIs, and this section will focus on the most important ones. The first API is the AndroidView-Composable, which serves as a mediator between the Composable and the View world, allowing developers to reuse Views within a composition. Listing 5.2 will give an example on how to use this API.

```
@Composable
fun ImageView(){
    AndroidView(factory = {
        context ->
        ImageView(context).apply {
            setImageResource(R.drawable.image1)
        }
    })
}
```

Listing 5.2: A Sample of using AndroidView

Integrating Views into Composables is a useful tool during migration, but sometimes Composables may need to be placed within an XML layout. In such cases, the provided ComposeView class can be employed. To use it, insert a ComposeView into the XML layout with an Identification (ID) that can later be referenced after inflation. Then, the *setContent()* method may be used to fill the ComposeView with any desired Composable, as showed in code shown below.

```
//inside XML layout
 <androidx.compose.ui.platform.ComposeView
       android:id="@+id/compose_view"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"/>

//inside Activity or Fragment
val composeView = findViewById<ComposeView>(R.id.compose_view)
composeView.setContent {
    MyComposable()
}
```

Listing 5.3: A sample of using ComposeView

Another key aspect of the Android View system is the utilization of Fragments. Fragments not only organize UI-elements into cohesive UI parts with their own ViewModel but also control their lifecycle and simplify navigation between screens using Jetpack Navigation. To populate a Fragment with a Composable UI tree, XML files can be entirely omitted. By creating a ComposeView object within the Fragment's *onCreateView()*-function, the content of this View can be set to the desired Composable. However, putting a Fragment inside of a Composable requires a more complex approach. To simplify this process, Jetpack Compose offers an interoperability API named AndroidViewBinding that provides an interface for inflating any given XML layout within a Composable function. To achieve this, create an XML layout with a single FragmentContainerView enclosing the desired Fragment. Then, inside of your Composable, inflate the layout to insert the Fragment. To correctly handle the lifecycle of the Fragment, one must detach the FragmentContainer from the NavController when the parent Fragment or Activity is destroyed. An example of inflating such a Fragment in Compose is given in the following listings.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/fragment_container_my_fragment"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:name="com.myApplication.MyFragment" />
```

Listing 5.4: XML needed for inflating the Fragment

```kotlin
@Composable
fun MyComposedFragment(){
    AndroidViewBinding(
        factory = FragmentContainerMyFragmentBinding::inflate)
}
```

Listing 5.5: Inflating a Fragment using AndroidViewBinding

A more detailed guide about interoperability can be found inside Google's Android developer guide under https://developer.android.com/jetpack/compose/migrate/interoperability-apis.

Decision

Interoperability APIs are ultimately crucial for a trouble-free migration. Therefore, interoperability APIs should be used if needed. However, the goal is to remove those framework bridges eventually in order to get a fully migrated application. Especially Fragments should only be put inside of Composables if absolutely necessary, since it can lead to lifecycle problems and it comes with a certain overhead. If the migration follows a bottom-up approach the FragmentContainer should be migrated when the underlying Fragment is already fully migrated making the use of this interoperability API superfluous.

Consequences

Leveraging interoperability enables a smooth migration process. The ability to reuse Views within Composables is beneficial during the migration process as complex custom Views can be reused without requiring migration first. The ability to place Composables inside of XML layouts is especially necessary when migrating larger layouts, where the entire layout cannot be migrated at once, due to time constraints or the need to break the migration into more smaller steps. It is also critical for a bottom-up approach in order to place the newly migrated Composables inside of their parent layout. All of the ways to use interoperability have an overhead. However, the ability to mix the two different frameworks is crucial for a large-scale migration, in order to avoid having to migrate the entire application at once.

5.3 Multiple screen sizes

Context

The idea of accommodating multiple screen sizes in the automotive industry is not a new concept when it comes to developing Android software. It is also applicable to common mobile applications, such as providing distinct layouts for smartphones, tablets, and wearables. One way to handle varying screen sizes is by utilizing XML resource files and assigning resource qualifiers to different XML files, allowing Android to take over from there. However, if the resources are saved within Kotlin files, a different approach must be used. A recommendation is given in the following.

Decision

The configuration, available as a CompositionLocal, provides access to the current screen dimensions. Unlike mobile applications, software for automotive systems anticipates different screen types and can optimize for them. To achieve this, a sealed class can be created to represent each unique screen. The Display can then be passed to the composition tree via the CompositionLocalProvider as shown below.

```kotlin

sealed class Display(val width: Dp, val height: Dp) {
    object: MainDisplaySmall : Display(1440.dp, 720.dp)
    object: MainDisplayBig : Display(1920.dp, 1080.dp)
    object: CoDriverDisplaySmall : Display(1080.dp, 720.dp)
}
@Composable
fun DisplayCompositionLocal(content: @Composable () -> Unit) {
    val configuration = LocalConfiguration.current
    val display = remember(configuration) {
          when (configuration.screenWidthDp) {
               1440 -> Display.MainDisplaySmall
               1920 -> Display.MainDisplayBig
               1080 -> Display.CoDriverDisplaySmall
               else -> error("Display size unrecognized")
          }
    }

    CompositionLocalProvider(LocalDisplay provides display) {
        content()
    }
}

val LocalDisplay = staticCompositionLocalOf {
    Display.MainDisplaySmall
}
```

Listing 5.6: Implementation of a Display CompositionLocal

It is advisable to minimize the differences between these versions, preferably by keeping such distinctions confined within the resource files. In situations where this is not feasible, it is recommended to utilize minimal conditional statements, such as ones that choose between a Row or Column. However, if the situation becomes unmanageable, or if the difference in layout is significant, it is best to build a factory that selects the appropriate implementation based on the Display. It is highly recommended to create all implementations as subclasses of a shared abstract class or interface to prevent duplication of code.

Consequences

If these steps are followed, the resulting code will be free of duplication and is easy to maintain. By eliminating code duplications and implementing as much as possible without differentiating between the screens, testing expenses will be decreased and the code will be more maintainable. Additionally, this implementation can handle changes to the screen configuration during runtime. A common example would be if the screen orientation changes or if an app is moved from the main display to the co-driver display.

Alternative Solution

Alternatively, the Display can be implemented through a publicly accessible composable function that returns a value. This approach presents some minor drawbacks compared to the other solution. First it requires a method call to access the Display, instead of a property access, which is a matter of personal preference. However, by implementing the Display as a remember function, a new instance of the Display will be instantiated each time the function is called, which can result in a small decrease in performance and increase in memory needed. The implementation of the Display as a remember function is shown in listing 5.7.

```
@Composable
fun rememberDisplay() {
    val configuration = LocalConfiguration.current
    return remember(configuration) {
            when (configuration.screenWidthDp) {
                1440 -> Display.MainDisplaySmall
                1920 -> Display.MainDisplayBig
                1080 -> Display.CoDriverDisplaySmall
                else -> error("Display size unrecognized")
            }
    }
}
```

Listing 5.7: Implementation of a remember function for the Display

5.4 Multiple OEMs

Context

As previously noted in the architecture section of this thesis, supporting multiple OEMs can be achieved through the use of Gradle build flavors. However, it is still for debate what should be made flavor-dependent and what should be placed inside the non-flavored directory.

Decision

Similar to managing multiple screens, it is best practice to minimize differences between different build flavors, ideally restricted to resource files. Often, design requirements for different OEMs differ only in terms of styling, such as color choices, dimensions, strings, and drawables. To accommodate this, style classes can be extracted from Composables and passed as a parameter to the implementation that differs between the various flavors. If the designs differ significantly, it may be necessary to create two entirely separate implementations. When using different implementations for different flavors, a shared interface should be declared inside of the unflavored project directory, which then can be implemented by the flavor dependent classes.

Consequences

When implementations only differ in value resources, tests often can be written flavor independent, which ultimately reduces the amount of testing resources required. Declaring shared interfaces for flavor dependant implementations reduces the risk of compilation errors due to unresolved references, which can speed up the development process. Furthermore, it helps to reduce and find bugs by providing a clean structured architecture. Nevertheless, the implementation of the interface takes time, which is ultimately compensated by the time needed to maintain the software in the long run.

5.5 Slot API

Context

The Slot API pattern is a prevalent design pattern in the Jetpack Compose codebase. Its aim is to simplify UI elements despite the UI components' growing complexity. Even minimal data input can result in significant implicit usage of the data to define its display. Composables that use the Slot API present one or more parameters of type *@Composable () -> Unit* that are laid out within the Composable. The Scaffold is a popular example of the Slot API inside of the

Compose codebase. This Composable provides slots for frequently used widgets that are then arranged on the screen comprising the Drawer, TopBar, BottomBar, and FloatingActionButton ('Slot-based layouts', 2023). An example of a Composable layout using Slot API can be seen in listing 5.8 and how the slots will be layed out in figure 5.1.

```kotlin
@Composable
fun SlottedLayout(
    Headline: @Composable () -> Unit,
    Body: @Composable () -> Unit,
    SideBar: @Composable () -> Unit,
) {
    Column(
        modifier = Modifier.fillMaxWidth().height(300.dp)
    ) {
        Box(modifier = Modifier.height(50.dp)) {
            Headline()
        }
        Row(modifier = Modifier.fillMaxSize()) {
            Box(modifier = Modifier.fillMaxWidth(0.8f)) {
                Body()
            }
            Box(modifier = Modifier.fillMaxSize()) {
                SideBar()
            }
        }
    }
}
```

Listing 5.8: A sample of using Slot API

Decision

The Slot API pattern shall be used throughout the entire application, especially in the top section of the composition. When design requirements are inflexible and do not change, the Slot API pattern can be omitted. This also applies to Composables which are located at or near the leaves of the composition tree.

Consequences

This mechanism reveals to the Composable's caller that it's divided into multiple sections, potentially requiring the caller to split the data passed into the sections themselves. This is advantageous in the top sections of your composition tree as it reduces complexity and separates the data passed to the subsections, eliminating the need to pass a large data object or the entire ViewModel. Another advantage

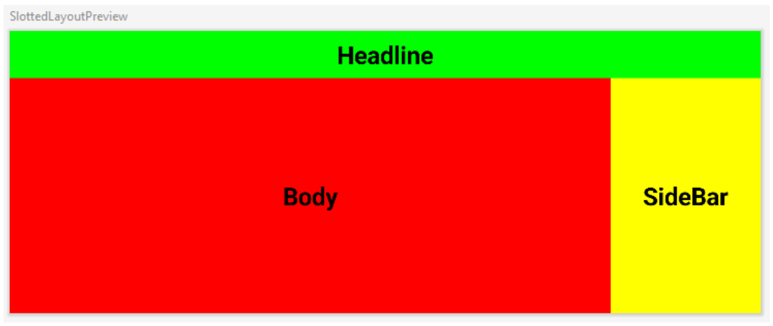


Figure 5.1: Preview of slotted layout

of utilizing this framework is the cohesiveness and lack of coupling in the implemented Composables. It also offers enhanced flexibility and reusability in UI code writing. By reusing a screens layout that has been implemented using Slot API, a consistent look can be created throughout a single or multiple applications without the need for code duplications. However when requirements are fixed, passing data directly at the bottom of the composable tree is likely more efficient, particularly for displaying a single type of information. Using the Slot API in this case may not be entirely unfavorable, although it could result in unnecessary complexity and boilerplate code. Ultimately, it is up to the development team to decide whether to employ this design pattern. However, prioritizing flexibility often results in fewer complications in the future (Banes, 2021).

5.6 Lists

Context

An essential widget found in almost all applications is the list. Especially interesting is dealing with dynamic lists of an unknown size. For dynamic lists, the RecyclerView is the preferred choice within the View system. Although the RecyclerView efficiently creates high-performing dynamic lists, it generates an excessive amount of repetitive boilerplate code. To display items within a RecyclerView, a RecyclerViewAdapter, which supports all types of list items potentially added to the list, must be created. In large scale automotive applications, a list may contain various list items which could be added to a single RecyclerView. This results in complex and bulky adapter classes necessary to accommodate all dataset types and combinations. If a new type of list item is needed not only the layout of the list item must be implemented but also the adapter must support the new list item. Both are typically established within the shared widget library, which leads to a long development cycle if a new list item must be added. First the item must be implemented inside of the widget library which then has to be tested and deployed, in order for the application to be able to use it.

Decision

Jetpack Compose introduces a powerful API for creating dynamic lists called LazyLayout. This API includes different Composables, namely LazyColumn, LazyRow, and LazyGrid, which provide the intuitive functionality associated with their respective names ('Lists and grids', 2023). The key distinction from RecyclerView is that no adapter is necessary, and any type of Composable can be declared within the LazyLayouts, with Compose handling the lazy aspect seamlessly. The use of LazyLists eliminates the need for a central adapter. Consequently, widget libraries can provide better tooling for these types of lists, empowering library developers to grant application developers greater control over list items. Developers can now construct list items more efficiently using powerful list item build tools. This can be achieved using the Slot API pattern. The widget library can now furnish fully-formed list items as well as items containing placeholders that can be populated by any Composable.

Consequences

Providing a comprehensive list item package as well as a toolkit simplifies the support for custom list items in applications, while also allowing the use of shared list items between multiple applications. This reduces the amount of boilerplate code, enhances code reusability, and can lead to a significant reduction in implementation time for new features or change requests. If an unimplemented list items is requested, developers of the application can add it to their own app. This saves time from implementing it within the library, which requires a full deployment cycle for a new version to be released that the application can build against. Additionally, a building kit for various UI-elements can decrease development time due to their high level of reusability. In contrast to the XML-system, only the widget's implementation needs to be written without the need for an adapter, as there is no requirement for one. Furthermore, the concept of separation of concerns is granted, since no application specific list items are implemented inside of the widget library. If the newly requested widget will be used in multiple applications, its implementation can still be refactored to be included in the widget library despite any time constraints. However, LazyLayouts are not directly compatible with old Views, which potentially want to be reused during the migration process. Nevertheless, Compose's developers created an API which handles the reusing aspect of Views inside of LazyLayouts by declaring two recall functions one for when the View will be reused and one for when it leaves the composition. The need for this interoperability API is way less overhead than needed when implementing a RecyclerView, therefore LazyLayouts should be preferred over the RecyclerView, considering the other benefits it entails ('Using Views in Compose', 2023).

5.7 ConstraintLayout

Context

When migrating UI layouts, one might find a lot of ConstraintLayouts, that have gained popularity in traditional Android development. This is because the View system has performance issues with nested hierarchies (‘Performance and view hierarchies’, 2023). Therefore a flat view hierarchy is preferred. Using ConstraintLayout promotes such flat hierarchies by positioning Views based on their relative position to other Views to which they are constrained. However, with Jetpack Compose, Google has addressed the performance issues associated with nested hierarchies, stating in their developerguide (2023) that "Since Compose avoids multiple measurements, you can nest as deeply as you want without affecting performance."

Decision

When migrating it may be tempting to keep the structure of the ConstraintLayout. If desired this is possible since Google offers a ConstraintLayout that behaves quite similarly to the XML version, giving every component that is to be constraint an ID, which can be referenced in a set of constraints, thereby constraining the components to each other. However it is desirable to make use of the standard nesting layouts provided by Compose, meaning Row, Column and Box.

Consequences

Creating ConstraintLayouts in XML is a comparatively simple task because the Drag’n’Drop style editor makes it easy to constrain different Views to each other with the click of a button. But, not using ConstraintLayout can result in smaller Composables because ConstraintLayouts often lay out many components. As with using Row, Column and Box, you can strategically partition the current area, to create more structured UI code. In traditional Android development, LinearLayouts often lead to nested hierarchies. In Compose, this is also the case, but Compose does not have any issues with those. Also, they feel more like Compose, as ConstraintLayout reminds one of XML, where IDs have to be assigned. It is also harder to imagine what a layout might look like given on a potentially large number of constraints. Nested linear and box layouts are easier to visualize in the mind.

5.8 BlockingState

Context

A common use case for automotive software is to limit user interaction in a car based on the current state of the vehicle. This can be achieved through the use of BlockingState, also known as CarUxRestrictions. For instance, if the car is moving at a particular speed, playing videos may be restricted ('CarUxRestrictions', 2023). To obtain the current BlockingState, an Observer must be supplied to the Android's native CarUxRestrictionManager.

Decision

The BlockingState is often accessed from various points within the Composable tree. To address this, the proposed solution is implementing a CompositionLocal that can be accessed from anywhere to provide the BlockingState. The implementation of this CompositionLocal is stated in the following listing.

```kotlin
@Composable
fun CarUxRestrictions(content: @Composable () -> Unit) {
    val context = LocalContext.current
    var carUxRestrictionsManager = remember(context) {
        Car.createCar(context)
        .getCarManager(Car.CAR_UX_RESTRICTION_SERVICE)
        as CarUxRestrictionsManager
    }
    var restrictions
        by remember { mutableStateOf(UX_RESTRICTIONS_BASELINE) }

    LaunchedEffect(carUxRestrictionsManager) {
        carUxRestrictionsManager.registerListener {
            restrictions = it.activeRestrictions
        }
    }

    CompositionLocalProvider(
        LocalRestrictions provides restrictions
    ) {
        content()
    }
}

val LocalRestrictions
    = staticCompositionLocalOf { UX_RESTRICTIONS_BASELINE }
```

Listing 5.9: Implementation of a CarUxRestrictions CompositionLocal

If an action becomes disabled due to the BlockingState, it is common to display a disclaimer text. To make this functionality reusable, a custom Box that displays the content based on the current BlockingState, as seen in the listing below, is an effective option.

```
@Composable
fun BlockableBox(
    blockedStates: Int,
    modifier: Modifier = Modifier,
    content: @Composable () -> Unit
) {
    Box(modifier = modifier.fillMaxSize())
    if (blockedStates and LocalCarUxRestrictions.current != 0) {
        BlockingDisclaimer()
    } else {
        content()
    }
}
```

Listing 5.10: Implementation of a BlockableBox

Consequences

Using a CompositionLocal provides a concise mean of declaring a value that is accessible throughout the composition. This technique is employed for the BlockingState, which is utilized extensively throughout the application. Furthermore, only one observer needs subscribe to the UxCarRestrictionManager. This entails an increase in performance in contrast to adding a new subscription each time the BlockingState is required. The providers implementation conforms to the principle of a single source of truth. This ensures that the value is accessed only once, guaranteeing consistency throughout the entire application. Consequently, the software becomes more robust and easier to maintain. Implementing a common BlockableBox that shows a fixed declaimer leads to an easy use of BlockingState and ensures a consistent look throughout the entire range of applications. It should be noted that this type of provider can provide any value depending on the system context or configuration, such as current speed, fuel type, UI-mode, or selected gear by changing the manager subscribed on.

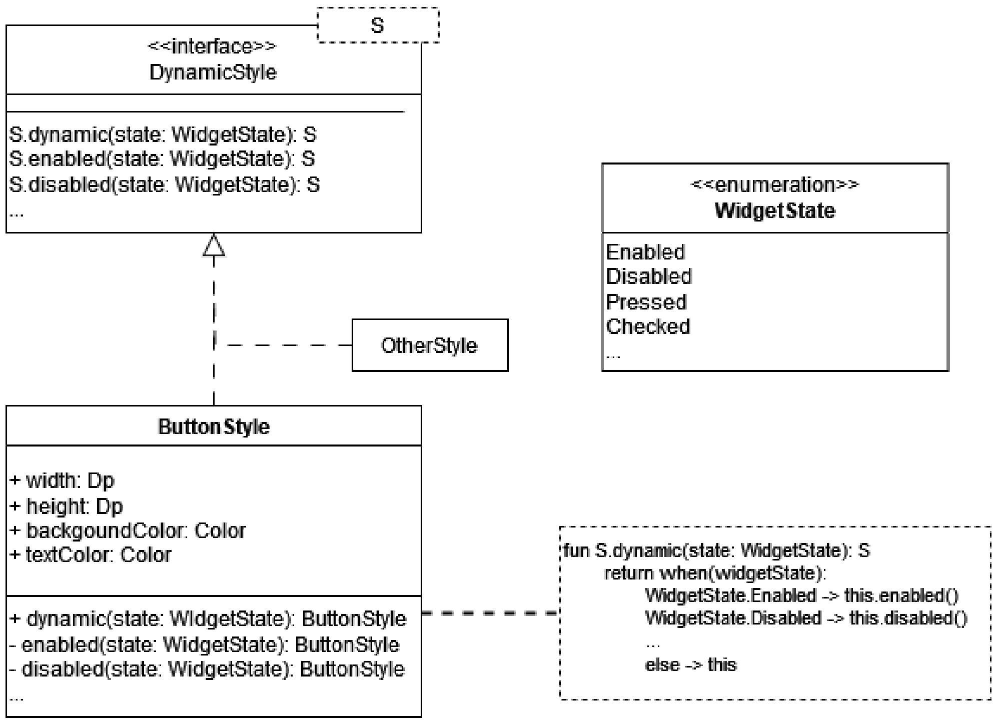
5.9 Selector styles

Context

Another use case is the necessity of incorporating dynamic styles, which rely on the current state of the Composable. In traditional Android development this functionality is achieved through the use of a selector, which determines the appropriate color or drawable based on the View's states, including enabled, disabled, pressed, focused, or checked.

Decision

To create a streamlined implementation for this use case, an enum class that defines the various states a Composable can assume should be implemented. Additionally, an interface called DynamicStyle is created, that includes the functionality required to provide the appropriate style based on the current state. Individual style classes can then implement this interface. This architecture of classes is visualized in the following class diagram .


S
<<interface>>
DynamicStyle
S.dynamic(state: WidgetState): S
S.enabled(state: WidgetState): S
S.disabled(state: WidgetState): S
...
<<enumeration>>
WidgetState
Enabled
Disabled
Pressed
Checked
...
OtherStyle
ButtonStyle
+ width: Dp
+ height: Dp
+ backgoundColor: Color
+ textColor: Color
+ dynamic(state: WIdgetState): ButtonStyle
- enabled(state: WidgetState): ButtonStyle
- disabled(state: WidgetState): ButtonStyle
...

```
fun S.dynamic(state: WidgetState): S
    return when(widgetState):
        WidgetState.Enabled -> this.enabled()
        WidgetState.Disabled -> this.disabled()
        ...
        else -> this
```

Figure 5.2: UML-diagram for DynamicStyle (drawn with draw.io)

```
@Composable
val MyButton(w: WidgetState) {
    val style = remember {
        ButtonStyle(width = 100.dp, height = 100.dp,
            textColor = Color.WHITE, background = Color.GREEN)}
    Button(style = style.dynamic(w))
}
```

Listing 5.11: A sample of using DynamicStyle

Consequences

To ensure consistency throughout the entire code base, it is recommended to have an interface for dynamic styles with a uniform implementation. Additionally, creating top classes for different style categories such as ButtonStyle, TextStyle, or ImageStyle can provide default implementations for dynamic styles. This allows for easy reuse of the default behavior, while also serving as an example for usage.

5.10 Testing

In the world of Android automotive software development, thorough testing plays a vital role in guaranteeing the functionality and security of the application. Compared to typical mobile applications, automotive software necessitates a greater degree of accuracy because of safety regulations governing in-car systems. Effective testing not only helps identify bugs, but also enhances the user experience by preventing crashes, improving performance, and ensuring seamless interactions with vehicle functionalities.

Traditionally, Android tests were created by inflating XML layouts into Views that could be tested, but this method made testing these Views cumbersome. However, thanks to Jetpack Compose, the testing process has been significantly improved. The first enhancement is not actually in the testing process itself, but in the way the Compose code is written. By using a declarative approach, the UI's appearance is now based on its state. Consequently, tests can be easily written to validate the UI based on specific states and scenarios in order to improve the comprehensibility of tests. Additionally, Jetpack Compose includes the incremental build of Composables, allowing every Composable within the hierarchy to be tested. This type of detailed testing not only produces smaller, more understandable tests, but also increases test coverage, resulting in greater confidence in the application's behavior.

The tests are akin to the usual Android tests and are fully interoperable, allowing for the testing of Views within Composables and vice versa. Nonetheless, one contrast is that UI-components cannot be referenced by ID, as is frequent in conventional Android tests. This is due to the fact that Composables do not possess an ID. Composable nodes should be identified using their content description, text, or hierarchical position within the composable tree. Another option is to assign a test tag to the Composables. However, this approach adds additional code to the production code for testing, which is generally not desirable.

6 Evaluation

6.1 Requirement satisfaction

6.1.1 Requirements

Continuous migration

The gathered requirements in this thesis can all be met with the availability of numerous interoperability APIs. The combination of traditional UI development and Jetpack Compose allows for seamless integration at any level, thus enabling continuous migration.

Reduction of boilerplate code

One major inconvenience within the View system is the use of dynamic lists, which now can be implemented with greater flexibility and reduced boilerplate code. It is difficult to quantify exactly how many Lines of Code (LOC)s can be saved without completing the migration, due to overhead in interoperability code and reusable code that may be implemented but not used to its full potential. For the most part, there is a noticeable decline of at least 25%. This decrease is significant and becomes even more apparent when implementing features for multiple OEMs or multiple screen sizes. The increased ability to extract design and style classes also aids in eliminating code duplication. Usually only a few dozen flavor-specific LOCs need to be added, whereas with traditional XML layouts the entire layout had to be duplicated and slightly modified. Additionally, when implementing UI features in Compose, typically only one single file needs to be created, as opposed to the View system where usually an XML file and a new or extended Fragment are typically needed. Overall, the trend in Compose code is leading to a substantial decrease in redundant code through concise declarations.

Clean architecture and separation of concerns

The ability to create highly reusable and well-organized UI code not only enhances separation of concerns but also promotes clean architecture. In the proposed implementations in chapter 5 a consistent focus on these goals was placed, creating scalable and clean solutions.

Improvement of testability of UI code

The ability and encouragement to implement hierarchical, smaller and reusable UI components leads to an improved testability. A clean architecture and separation of concerns will also enhance code testability.

Feature usage

The debate over whether Jetpack Compose is more feature-rich than the traditional framework is ongoing. Although Jetpack Compose offers useful features such as Previews, LiveEdit, and code completion, as well as type safety, it lacks some useful features like a Drag'n'Drop editor. With that being said, Jetpack Compose does offer interoperability support for the old framework, allowing for the utilization of the best of both worlds if desired.

6.1.2 Quality goals

Performance

It is difficult to determine the superior framework based on performance alone. While Jetpack Compose can boast with improved startup time and navigation, XML tends to have better rendering performance for screen scrolling. Nevertheless, Jetpack Compose's performance is continually increasing and may ultimately surpass that of XML in the future (Zaed & Caesar, 2023). When beginning the migration process, expect a minor increase in the size of the APK due to additional libraries needing inclusion. These include both Compose libraries and possibly multiple versions of the widget library for compatibility. Nevertheless, the build APK's size is predicted to significantly decrease once the migration is completed. The effect of Compose on the application's build time behaves quite similar by increasing slightly when Jetpack Compose is first introduced into the project but is expected to noticeably decrease ('Compare Compose and View performance', 2023).

Interoperability

As previously discussed, Compose provides a high level of interoperability, with different APIs to integrate Compose into the traditional Android View system and vice versa.

Transferability

The proposed implementation were created for generic use cases uncoupled from a specific application making them transferable and a context is given to know when the given solution is applicable.

Visual identity

Compose enables developers to create a visually identical twin to an XML-based layout. In cases where a specific View may not exist in Compose, an interoperability API allows integration of the old Android View into the Composable for a matching appearance.

6.1.3 Stakeholders

Customer

The expectation for no decrease in developer productivity is not feasible and unrealistic. A migration process may take time and ultimately result in decreased development speed. However, an increase in development speed can be expected after migration, which will compensate for the temporary decrease.

Software developers

Providing clear documentation for specific architectural decisions and planning the migration ahead of time will lead to a streamlined and efficient migration process, resulting in a well-designed codebase. The increased possibilities developers have when writing Compose code will enhance their experience reinforced by reducing the need for XML code, which is often disliked by developers.

Team lead

As mentioned earlier, the overall productivity of the team may decrease during the migration because the migration takes time. However, Jetpack Compose introduces a new technology that could increase developer motivation. In addition, Jetpack Compose is a much-discussed topic in the Android development community, and adopting it could potentially attract more candidates to your company.

Test engineers and quality Assurance

Using Compose reduces redundant code and generates more reusable elements, thereby enhancing overall testability. This leads to increased test coverage, improving the quality and robustness of the codebase.

User

The ability to create visual twins of the traditional Views and no significant drop in performance makes the migration invisible to the user.

6.2 Disadvantages of migrating to Compose

Employee training

Jetpack Compose represents a paradigm shift in UI development with its declarative approach, in contrast to the imperative nature of XML layouts. Developers trained in XML and Views need to invest time and effort in learning Compose's syntax, concepts, and best practices. This process may slow down development speed initially, but it is necessary for a successful migration.

Programming knowledge

Another issue with Jetpack Compose is that it requires UI developers to possess certain programming skills and knowledge to write UI code. This stands in contrast to the relatively easier task of creating layout files using XML, which can be done even by a non-experienced programmer due to its lower complexity. The effect is accentuated by the design tab when editing XML layouts. The Drag'n'Drop style editor facilitates easy layout creation without requiring code, accommodating a non-programmer audience. Nevertheless, modern UI-developers should possess sufficient programming knowledge to overcome this concern. Notably, Google's introduction of databinding already eliminated the division between XML and programming domains.

New technology

One disadvantage of Jetpack Compose is its relative youth compared to the more established View framework, which has been dominant since its release in 2008. Compose was released less than a decade ago, meaning community support is currently smaller in comparison. Additionally, given Compose's ongoing development, new features and APIs are regularly being published. This could result in instability problems and errors that may still exist within the compose code base, potentially causing crashes or undesirable UX.

6.3 Migration strategies

6.3.1 Ad hoc migration

In all cases, pre-planning the migration process is essential to prevent redundant work and the creation of a low-quality architecture and implementation. Therefore, it is crucial to define precisely when and what to migrate. An ad hoc migration that replaces the current application's View layer in its entirety may reduce work to maintain interoperability with older parts of the code base. Moreover, it might result in a cleaner implementation since the entire user interface can be rebuilt. This approach may also decrease the risk of attempting to replace each old UI element with an equivalent Composable compared to a migration done in numerous small stages. However, in many cases, migrating an extensive application all at once is not a practicable task. This is because it would necessitate a large number of individuals to perform refactoring on a significant portion of the codebase simultaneously, resulting in several potential merge conflicts. Additionally, depending on the application's size, this process could put a hold on feature development and bug fixing for weeks or possibly even months.

6.3.2 Incremental migration

Where to start?

A more feasible strategy would be to utilize available interoperability APIs and undertake the migration incrementally. Before beginning, establish some conventions, such as creating common interfaces for styling or determining whether resources should remain in XML files or be migrated to Kotlin files. Implementation of resources, styling or common interfaces lie within the widget library, hence this subproject is a good point to start the migration process. Once this is done, the application's migration can commence.

Bottom-up migration

A bottom-up approach is often the most efficient method of migration. When done correctly, it minimizes the need for interoperability, which can be a laborious process. This approach involves starting from smaller widgets or features positioned at the bottom of the UI tree, which are typically easier to migrate due to their smaller size. That way, developers can broaden their knowledge on composing without the need to transfer the design of entire screens, which can pose a significant challenge to migrate.

Fragments

For the initial phase of migration, it is advisable to halt at the Fragment level due to some functionality that may be linked to the Fragments. Fragments have their own lifecycle, which includes callback functions that are triggered on lifecycle events, such as *onStart()*, *onStop()*, *onResume()*, or *onDestroy()*. These functions are occasionally utilized to initiate events to the ViewModel. Nonetheless, Composables lack such a lifecycle. They are either part of the composition or not. Therefore, it is necessary to handle these events differently by creating state variables that hold the value indicating whether the composition is currently displayed or not, which can then trigger events upon change. Additionally, Jetpack Navigation often utilizes Fragments, making it more challenging to mix Fragments and Composables at that level. If all Fragments have been migrated, meaning that they only consist of a top-level Composable, then the Fragments can be easily omitted. The Jetpack Navigation can be transformed into Compose Navigation, which behaves similarly.

What to migrate next?

Deciding which components to migrate next can be approached in two ways. The first involves a more in-the-moment planning approach, where developers can select a part of the UI to migrate if there are no higher priorities to attend to at the moment. Alternatively, the confrontation approach entails migrating those features that are to be affected by a current change request or bug fix. If new UI widgets are added during the migration process, it is advisable to use Compose for optimal efficiency and to avoid duplicate feature implementation. In such cases, interoperability APIs might need to be used more extensively than in strictly incremental migration, but it eliminates the need for complete feature implementation repetition.

7 Conclusions

This thesis addressed various aspects to consider when migrating an Android automotive application to Jetpack Compose. It has become apparent that preparing a strategy for a large-scale migration is crucial and the process should not be done impulsively. Nonetheless, Jetpack Compose presents an opportunity to enhance not only the application's quality but also its performance and development time. Many automotive use cases that have been addressed using conventional UI development can also be realized through Jetpack Compose. However, migrating to this new framework requires significant time and resources for employee training and code refactoring. However, discussing possible solutions or initiating the migration process represents a positive step forward. As the saying goes, "Rome wasn't built in a day" and this migration cannot be completed quickly or hastily. A large-scale migration provides an ideal chance to reconsider the architecture and structure of apps for enhancement purposes. In conclusion, although migration is time-consuming, its benefits make it worthwhile. Therefore I can highly recommend migrating any application that is still in development to Jetpack Compose.

Further research on the direct effects of fully migrating a significant automotive application would be intriguing. It would be interesting to observe the app's performance, as well as the actual APK size and total LOCs. Another area that requires further investigation is the resources required to maintain the code base. Jetpack Compose guarantees exceptional reusability and maintainability as a result of its tidy code structure.

Since Jetpack Compose is still under development, we can all look forward to the new features and innovations that Google may introduce in the future.

Appendices

A Recommended resources to learn Jetpack Compose

- Google's Jetpack Compose Pathways: https://developer.android.com/courses/jetpack-compose/course
- Phillip Lackner's Crash Course: https://youtu.be/6_wK_Ud8--0?si=D5XtCLIrn8bS0uQf
- "Android UI Development with Jetpack Compose" by Thomas Künneth

References

Afridi, K. (2023). Declarative vs. imperative ui frameworks: Understanding the key differences. https://medium.com/@afridi.khondakar/declarative-vs-imperative-ui-frameworks-understanding-the-key-differences-7ad2922855ff. (accessed: 29.10.2023 20:42)

Android's kotlin-first approach. (2023). https://developer.android.com/kotlin/first (accessed: 22.10.2023 15:33)

Banes, C. (2021). *Slotting in with compose ui.* https://chrisbanes.me/posts/slotting-in-with-compose-ui/ (accessed: 22.10.2023 15:11)

Caruxrestrictions. (2023). https://developer.android.com/reference/android/car/drivingstate/CarUxRestrictions (accessed: 29.10.2023 22:24)

Castillo, J. (2022). *Stateful vs stateless composables.* https://newsletter.jorgecastillo.dev/p/stateful-vs-stateless-composables (accessed: 30.10.2023 13:01

Castillo, J. R. (2022). Building custom themes in jetpack compose. https://betterprogramming.pub/jetpack-compose-custom-themes-b1836877981d. (accessed: 22.10.2023 16:28)

Chugh, A. (2022). Android mvvm design pattern. https://www.digitalocean.com/community/tutorials/android-mvvm-design-pattern. (accessed: 22.10.2023 18:10)

Compare compose and view performance. (2023). https://developer.android.com/jetpack/compose/migrate/compare-performance (accessed: 22.10.2023 17:49)

developerguide. (2023). *Compose layout basics.* https://developer.android.com/jetpack/compose/layouts/basics (accessed: 23.10.2023 10:46)

How new infotainment will shape the future customer experience. (2021). https://cariad.technology/de/en/news/stories/infotainment-customer-experience.html. (accessed: 22.10.2023 15:56)

IcePanel. (2023). Architecture decision records (adrs). https://icepanel.medium.com/architecture-decision-records-adrs-5c66888d8723. (accessed: 22.10.2023 18:16)

Iterative code development. (2023). https://developer.android.com/jetpack/compose/tooling/iterative-development (accessed: 22.10.2023 18:00)

Kotlinlang. (2023). *Kotlin for android.* https://kotlinlang.org/docs/android-overview.html (accessed: 22.10.2023 15:33)

Lists and grids. (2023). https://developer.android.com/jetpack/compose/lists (accessed: 29.10.2023 21:50)

Locally scoped data with compositionlocal. (2023). https://developer.android.com/jetpack/compose/compositionlocal (accessed: 22.10.2023 16:30)

Naeem, M. (2023). Android gradle (build types, product flavors, build variants, source sets). https://medium.com/@naeem0313/android-gradle-build-types-product-flavors-build-variants-source-sets-4c9631e6fb30. (accessed: 22.10.2023 18:02)

Performance and view hierarchies. (2023). https://developer.android.com/topic/performance/rendering/optimizing-view-hierarchies (accessed: 23.10.2023 10:39)

Preview your ui with composable previews. (2023). https://developer.android.com/jetpack/compose/tooling/previews (accessed: 22.10.2023 17:59)

Slot-based layouts. (2023). https://developer.android.com/jetpack/compose/layouts/basics#slot-based-layouts (accessed: 29.10.2023 21:32)

Software-defined vehicles – a forthcoming industrial evolution. (2021). *Deloitte.* https://www2.deloitte.com/cn/en/pages/consumer-business/articles/software-defined-cars-industrial-revolution-on-the-arrow.html. (accessed: 22.10.2023 15:52)

State and jetpack compose. (2023). https://developer.android.com/jetpack/compose/state (accessed: 22.10.2023 16:20)

Thinking in compose. (2023). https://developer.android.com/jetpack/compose/mental-model (accessed: 22.10.2023 16:20)

Using views in compose. (2023). https://developer.android.com/jetpack/compose/migrate/interoperability-apis/views-in-compose (accessed: 26.10.2023 18:18)

Varon, A. (2022). Declarative ui — what, how, and why? https://medium.com/israeli-tech-radar/declarative-ui-what-how-and-why-13e092a7516f. (accessed: 22.10.2023 16:04)

Zaed, N., & Caesar, E. (2023). *Ui performance comparison of jetpack compose and xml in native android applications.*