Discussion-2 What are the common challenges faced when designing complex layouts, and how can they be overcome – Jetpack compose?

Discussion Topic:

Please choose one of the following questions to discuss in your initial post:

What are the best practices for designing user-friendly interfaces in Android applications?
How can you handle different screen sizes and orientations in Android?
What are the common challenges faced when designing complex layouts, and how can they be overcome?

My Post:

Hello Class,

For my initial post, I chose the following topic: What are the common challenges faced when designing complex layouts, and how can they be overcome?

In the context of Android app development, designing complex layouts presents a set of common challenges. This is true when using both XML and Jetpack Compose. For my portfolio, I chose to use Jetpack Compose because Google encourages it. Google's parent company, Alphabet, owns Android, and Google manages it as a subsidiary. Google acquired Android in 2005 for just $50 million (Reynolds, 2017). Since then, Google has invested heavily in Android and guided the evolution of its ecosystem, making it the most popular mobile system in the world. This post provides a brief overview of Jetpack Compose and explores common challenges one can face when designing complex layouts in Jetpack Compose.

Jetpack Compose Overview

XML (Extensible Markup Language) has been the cornerstone of Android UI design since its early days in 2008 (Angelo, 2024). XML is a markup language used to structure and organize data, it supports information exchange between computer systems such as websites, databases, and third-party applications (AWS, n.d.). In the Android ecosystems, it is used to define User Interface (UI) layouts and elements. Developers use XML to specify the arrangement of buttons, text views, images, and other UI components. However, since declaring Kotlin the official programming language for Android app development in 2017, Google endorsed Jetpack Compose, a toolkit written in Kotlin, as the preferred solution for building complex Android app UIs, starting in early 2019 (Samojło, 2024; Angelo, 2024). See Table 1 for a list of the main differences between XML and Jetpack Compose.

Table 1

XML vs Jetpack Compose

	XML	Jetpack Compose
Syntax	Verbose, declarative and too much text	Declarative, programmatic and concise
Language	Java or Kotlin	Kotlin only
Code structure	Each layout has a separate XML file. Sometimes complex and hard to maintain	Inline UI Kotlin code and easy to maintain
Reactivity	Non-reactive	Reactive
Learning curve	Easy to learn	Easier to learn than XML
Customization	Customizable	Highly customizable than XML
Adoption	Widely used in old/earlier projects	Widely used in recent projects
Community	Extensive	Rapidly growing
Age	Old	Modern
Owner	W3C	Google Inc.

Note: The table lists the main differences between XML and Jetpack Compose. From "XML vs. Jetpack Compose: A Comparison" by Angelo (2024)

Compose toolkit is part of the Android Jetpack developer library from Google, a suite of toolkits that are used to create and manage UI, manage data, schedule background tasks, and more. The toolkit uses a declarative programming paradigm (Android Developers, n.d.a). This means that developers describe what your UI should contain, and the Jetpack Compose toolkit does the rest not needing to write anything in XML, everything is done through Kotlin code (Android Developers, 2022b).

Compose layouts are built using Kotlin composable functions, which are functions that emit *Units* describing elements of a UI (Android Developers, n.d.c). In Kotlin, a *Unit* is a type that represents no meaningful value, similar to *void* in Java or C++. Additionally, Almost everything in Compose is a *Layout* such as Box, Column, Row, and BoxWithConstraints. The examples below illustrate how to use a Compose basic function and the *Column Layout.*

Code Snippet 1

A Basic Composable Function

```
@Composable
fun ArtistCard() {
    Text("Alfred Sisley")
    Text("3 minutes ago")
}
```

Output

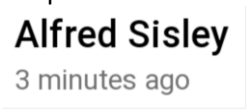
Note: The code Snippet is an example of a basic Kotlin composable function. From "Compose layout basics" by Android Developers (n.d.c).

Code Snippet 2

A Basic Composable Function Using A Column Layout

```
@Composable
fun ArtistCardColumn() {
    Column {
        Text("Alfred Sisley")
        Text("3 minutes ago")
    }
}
```

Output



Note: The code Snippet is an example of a basic Kotlin composable function using a C*olumn Layout*. From "Compose layout basics" by Android Developers (n.d.c).

Note that just by adding the *@Composable* annotation the compiler will treat the following function as a Composable function, and by encapsulating the *Text* element within the *Column Layout,* the text outputs automatically aligned vertically. The Compose function code could be translated to XML and Kotlin as follows:

Code Snippet 3

Compose Code Translated to Kotlin and XML

Kotlin

```
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity

class ArtistCardColumnActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.artist_card_column)
    }
}
```

XML

```
<?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="wrap_content"
    android:orientation="vertical">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Alfred Sisley"
        android:textSize="16sp"
        android:padding="8dp"/>

    <TextView
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="3 minutes ago"
        android:textSize="14sp"
        android:padding="8dp"/>

</LinearLayout>
```

Note: The XML and Kotlin code snippets are the translations of the Compose code from Code Snippet 2.

Note that just by adding the *@Composable* annotation the compiler will treat the following function as a Composable function, and by encapsulating the *Text* element within the *Column Layout,* the text outputs automatically aligned vertically. This shows one of the reasons why Jetpack Compose is generally preferred over XML as it is less verbose. However, XML integrates with Java and C++, whereas Jetpack Compose is exclusively Kotlin-based, so it's important to consider the application's needs when choosing between these two approaches. Additionally, large applications are more likely to adopt a hybrid approach, using both XML and Jetpack Compose; for example, an AAA or AA mobile video game is most likely to combine Kotlin, Java, and C++ with both XML and Jetpack Compose for UI/UX.

Common challenges When Designing Complex Layouts — Jetpack Compose

Whether using XML or Jetpack Compose within an Android application or a PC desktop application for that matter, designing complex UI/UX layouts has common challenges. Complex UIs/UXs need to balance functionality with usability, making the task of designing layouts a daunting one. Nonetheless, developers need to understand what those challenges are and how to address them.
The table below is the results of my research on the matter, it lists the most common challenges in designing UI/UX and general solutions for any app and specific solutions for those built with Jetpack Compose.

Table 2

Challenges and Solutions for Designing Complex App Layouts with Jetpack Compose

Challenge	Description	Solution	Solution Specific to Jetpack Compose
Balancing Complexity and Simplicity	The app needs to handle complex tasks without overwhelming users with a cluttered UI/UX.	Use progressive disclosure, that is show essential UI/UX elements first, revealing more as needed.	*ExpandableCard* in Jetpack Compose, as inspired by *Headspace*
Information Overload	Presenting too much information can confuse users and potential abandonment.	Organize content with clear headings, subheadings, and visual hierarchy (e.g., using Column and Text in Jetpack Compose with consistent typography).	Column with *Text* in Jetpack Compose, following *Headspace*'s approach
Navigation Difficulty	Complex layouts with too many features can overwhelm the users and make it difficult for them to navigate the interface.	Implement intuitive and easy-to-navigate UIs/UXs with standard and recognizable patterns.	*BottomNavigation* in Jetpack Compose, inspired by Instagram's centered design
Performance Issues	Complex layouts can slow down apps, and this can affect user experience, especially on lower-end devices.	Implement optimizations that promote fast rendering and coding that minimize unnecessary recompositions.	*LazyColumn* in Jetpack Compose for lazy loading, implements smooth graphics and animations

Cross-Platform Consistency	Layouts need to be responsive and adaptable, meaning that they need to adapt to different device platforms and screen sizes.	Use responsive design with adaptive UI/UX layouts and test on all possible devices	*Box* or *ConstraintLayout* in Jetpack Compose, which are tested across devices like iPhone and Samsung Galaxy

Note: The table provides a list describing the most common challenges in designing UI/UX their general solutions for any app and specific solutions for those built with Jetpack Compose. From several sources (TechAffinity, 2023; Blair, 2024; Sakthivel, 2022, Android Developers, n.d.d).

As shown in Table 1, Jetpack Compose provides modern solutions to common challenges in designing UI/UX, challenges like balancing complexity and simplicity, information overload, navigation difficulty, performance issues, and cross-platform consistency by offering a variated of tools. This affirms Google's strategy for pushing Jetpack Compose as the ideal tool for building complex UI/UX layouts within the Android ecosystem.

To summarize, Jetpack Compose is a powerful tool for Android UI development; it is concise, reactive, and Kotlin-based. Jetpack Compose offers more modern features than XML and is significantly less verbose, saving time, reducing code lines, and minimizing errors. Additionally, it addresses common challenges in designing UI/UX with effective solutions. Overall, Jetpack Compose a well-developed toolkit that reflects Google's vision for Android as a developer-friendly platform meant to guide the future of mobile app design.

-Alex

References:

Android Developers (n.d.a). Thinking in Compose. Android. https://developer.android.com/develop/ui/compose/mental-model

Android Developers (2022b, September 13*). Intuitive: thinking in compose - MAD skills* [Video]. YouTube. https://www.youtube.com/watch?v=4zf30a34OOA

Android Developers (n.d.c). *Compose layout basics*. Android. https://developer.android.com/develop/ui/compose/layouts/basics#:~:text=Composable%20functions%20are%20the%20basic,each%20other%2C%20making%20them%20unreadable:

Android Developers (n.d.d). *Jetpack Compose Tutorial*. Android. https://developer.android.com/develop/ui/compose/tutorial

Angelo, A. (2024, April 29). *XML vs. Jetpack Compose: A Comparison*. https://blog.openreplay.com/xml-vs-jetpack-compose--a-comparison/

AWS (n.d.). *What is XML?* Amazon. https://aws.amazon.com/what-is/xml/#:~:text=Extensible%20Markup%20Language%20(XML)%20lets,might%20give%20suggestions%20like%20these:

Blair, I. (2024, September 21). *The best app layout ideas for amazing user experience*. Buildfire. https://buildfire.com/best-app-layout-ideas/

TechAffinity (2023, March 15*). 7 common app design challenges and ways to overcome them*. TechAffinity Blog. https://techaffinity.com/blog/7-common-app-design-challenges-and-ways-to-overcome-them/

Sakthivel, K. (2022, April 11). Common app design challenges and their solutions. UXmatter. https://www.uxmatters.com/mt/archives/2022/04/common-app-design-challenges-and-their-solutions.php