JETPACK COMPOSE AND XML LAYOUT RENDERING PERFORMANCE COMPARISON

JETPACK COMPOSE UN XML LIETOTĀJA SASKARNES IZVIETOJUMA ATSPOGUĻOŠANAS ĀTRUMA SALĪDZINĀJUMS

Autors: **Iļja FJODOROVS**, e-pasts: ilja_latvia@mail.ru
Zinātniskā darba vadītājs: **Dr.sc.ing., doc. Sergejs KODORS**,
Rēzeknes Tehnoloģiju akadēmija,
Atbrīvošanas aleja 115, Rēzekne, Latvija

Abstract. *The aim of the work is to find out the rendering performance of new Google Android user interface framework "Jetpack Compose". Author has built two applications for Android platform with identical user interfaces: one uses classic approach with Kotlin + XML layout file, another application is developed using Jetpack Compose. In the results, the performance comparison of each approach is provided.*

Keywords: *Android, Jetpack Compose, user interface.*

Introduction

The expectations around UI development have grown. Today, we can't build an application and meet the user's needs without having a polished user interface including animation and motion. These requirements are things that didn't exist when the Android UI toolkit was created.[1]

To address the technical challenges of creating a polished UI quickly and efficiently Google Development Team have introduced Jetpack Compose, a modern UI toolkit that lets developers write user interface for Android OS using Kotlin programming language.

One of the fundamental things that developers like is the separation of concerns, as it is a well-known software design principle. Despite being well known, it is often difficult to grasp whether or not this principle is being followed in practice. It can be helpful to think of this principle in terms of "Coupling" and "Cohesion".[2]

When we write code, we create modules that consist of multiple units. Thus, *coupling* is the dependency among units in different modules and reflects the ways in which parts of one module influence parts of other modules. Meanwhile, *cohesion* is relationship among units within one module, it indicates how well the units are grouped in the module (see *Fig. 1*).When maintainable software is developed, it is important to minimize coupling and maximize cohesion[1].

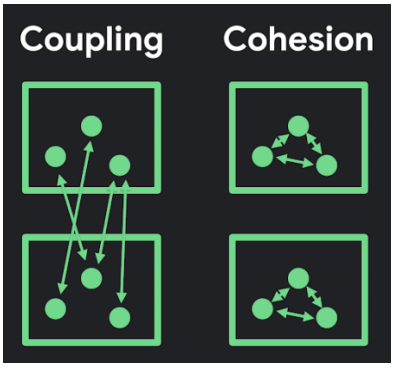

Fig.1. **Coupling and cohesion principles** [1]

https://doi.org/10.17770/het2021.25.6779

An UI development using Kotlin+XML represents the coupling paradigm. In this case, a code changes in one module require making changes in another. The coupling can often be implicit, because changes appear to be entirely unrelated.

On the other hand, Kotlin+Jetpack Compose toolkit application represents cohesion, because development is completed using the same language (Kotlin). In the result, the dependencies, that were implicit, start to become more explicit.

Materials and methods

The experiment environment is presented in Table 1.

Table 1. **Experiment environment**

Host OS	Windows 10 Home 20H2
IDE	Android Studio Arctic Fox | 2020.3.1 Canary 14
Compose Version	1.0.0-beta04 | April 7, 2021
Emulator Version	30.4.5 (February 23, 2021)
Virtual Device	Pixel 2 XL
OS on Virtual Device	Android 10 | API 29
Virtual Device RAM	2Gb of 8Gb DDR4 on the system
Virtual Device Cores	2 cores of 4 (Intel i5-8265U)

Two Android applications with similar user interface were developed for the experiment. The version with Kotlin+XML is depicted inFig.2, but the version based on application of Kotlin+Jetpack Compose is depicted in Fig.3. User interface contains an image of Android OS logo followed by a Lorem Ipsum paragraph. Style of text may vary, but it does not impact on the experiment.

Fig.2. **Test application with Kotlin+XML**

Fig.3. **Application with Jetpack Compose**

The source code of both applications is provided in Fig.4 and Fig. 5, the variables *start* and *end* record execution time to measure UI content preparation. Firstly, we save current time in *start* variable, after that we place corresponding layout code, and immediately after that we save current time in *end* variable. Now we can calculate the performance of each approach subtracting the *end* time from *start* time.

```
private var start = 0L
private var result = 0L

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        window.decorView.post {
            start = System.currentTimeMillis()
            setContentView(R.layout.activity_main)
            val end = System.currentTimeMillis()
            val result = end - start
            println("Rendering time XML = $result ms")
        }
    }
}
```

Fig.4. **Source code of application with Kotlin+XML**

```
private var start = 0L
private var result = 0L

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        window.decorView.post {
            start = System.currentTimeMillis()
            setContent {
                Column(
                    modifier = Modifier.padding(16.dp)
                ) { this: ColumnScope
                    Image(painter = painterResource(R.drawable.android_logo),
                        contentDescription = null)
                    Text(
                        text = stringResource(R.string.Lorem_ipsum)
                    )
                }
                val end = System.currentTimeMillis()
                val result = end - start
                println("Rendering time compose = $result ms")
            }
        }
```

Fig.5. **Source code of application with Kotlin+Jetpack Compose**

Results

Measurements were completed 10 times for each application. Console outputs are depicted in Fig.6 and Fig.7.

```
tem.out: Rendering time XML = 28 ms
tem.out: Rendering time XML = 25 ms
tem.out: Rendering time XML = 35 ms
tem.out: Rendering time XML = 29 ms
tem.out: Rendering time XML = 30 ms
tem.out: Rendering time XML = 29 ms
tem.out: Rendering time XML = 43 ms
tem.out: Rendering time XML = 23 ms
tem.out: Rendering time XML = 34 ms
tem.out: Rendering time XML = 33 ms
```

Fig.6. **Measurement results of Kotlin+XML version**

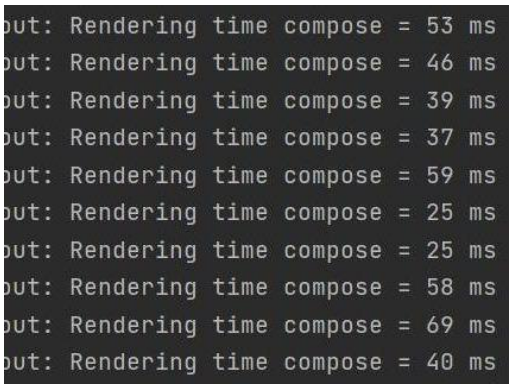

Fig.7. **Measurement results of Kotlin+Jetpack version**

Results of experiment are presented in Table 2.

Table 2. **Experiments results**

	XML	Compose
1	28	53
2	25	46
3	35	39
4	29	37
5	30	59
6	29	25
7	43	25
8	23	58
9	34	69
10	33	40

Min	23	25
Max	43	69
Average	30,9	45,1
%	100,00%	145,95%
d%	0,00%	45,95%

As can be seen in Table 2 average rendering time for XML is 30,9ms , and average rendering time for Compose is 45,1ms. The increase in rendering time is 45,1 – 30,9 = 14,2ms, or if we take XML time as 100% , we get a 45,95% rendering time increase for Compose version.

Conclusions

Results show approximately 46% Jetpack Compose rendering time increase comparing to XML layout file rendering. This fact can be explained by early development stage of the Jetpack Compose (by april of 2021, version: beta04)[3], so we can't really tell how it will perform in the final release. Research could be repeated after Jetpack Compose stable release.

Summary

Darba autoram bija interese uzzināt cik ātri Google Android platformas jaunais Jetpack Compose lietotāja saskarnes izstrādes rīks veic lietotāja saskarnes attēlošanu uz ekrāna salīdzinājuma ar „klasisko" pieeju ar XML failiem.

Darba autors uzskata ka 46% ātruma samazinājums ir kompromiss kuru gala lietotājs praktiski nepamanīs lietojot aplikāciju. Kā arī tā ir „cena kuru ir jāsamaksā" par aplikācijas lietotāja saskarnes izstrādes atvieglošanu.

Darba autors vēlētos uzsvērt uzmanību ka testētais Jetpack Compose lietotāja saskarnes izstrādes rīks ir izstrādes cikla „beta" stadijā. Tālākos izstrādes posmos var tik uzlabota ātrdarbība.

Pēc Jetpack Compose stabilās versijas jeb 1.0 versijas publiskošanas pētījumu varētu atkārtot.

Bibliography

1. Understanding Jetpack Compose https://medium.com/androiddevelopers/understanding-jetpack-compose-part-1-of-2-ca316fe39050
2. Software Engineering | Coupling and Cohesionhttps://www.geeksforgeeks.org/software-engineering-coupling-and-cohesion/
3. Announcing Jetpack Compose Beta!https://android-developers.googleblog.com/2021/02/announcing-jetpack-compose-beta.html