JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 26 (2023) 13–17
Received:14 September 2022
Accepted: 7 November 2022

Performance analysis of user interface implementation methods in mobile applications

Analiza wydajności metod implementacji interfejsów użytkownika w aplikacjach mobilnych

Jakub Szczukin*

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract
The purpose of this article is to analyze the impact of Jetpack Compose on user interface performance in mobile applications. A relatively new technology, Jetpack Compose, has not seen much research on its performance. The study used applications written in Kotlin, using the Jetpack Compose toolkit and views. The applications were tested with performance tests, using the Macrobenchmark tool, UI Automator 2 and JUnit 5. A literature review examining the impact of many factors on UI and Android performance was performed. In the end, upon completion of the testing, it was concluded that Jetpack Compose is slightly inferior in performance compared to interfaces built with views, in return it offers faster and easier code development.

Keywords: testing; user interface; jetpack compose; performance

Streszczenie
Celem artykułu jest analiza wpływu zastosowania Jetpack Compose na wydajność interfejsu użytkownika w aplikacjach mobilnych. Jetpack Compose jest stosunkowo nową technologią, przez co nie ma wiele badań na temat jej wydajności. Do badania zostały wykorzystane aplikacje napisane w języku Kotlin, przy użyciu zestawu narzędziowego Jetpack Compose oraz widoków. Aplikacje były testowane przy użyciu testów wydajnościowych, przy użyciu narzędzia Macrobenchmark, UI Automator 2 oraz JUnit 5. Został wykonany przegląd literatury badającej wpływ wielu czynników na wydajność interfejsu użytkownika i systemu Android. Po zakończeniu badań wywnioskowano, że Jetpack Compose jest nieznacznie gorszy wydajnościowo w porównaniu do interfejsów zbudowanych za pomocą widoków, w zamian oferuje szybszą i łatwiejszą pracę nad kodem.

Słowa kluczowe: testowanie; interfejs użytkownika; jetpack compose; wydajność

*Corresponding author
Email address: jakub.szczukin@pollub.edu.pl (J. Szczukin)

1. Wstęp

Historia urządzeń mobilnych jest stosunkowo krótka – pierwsze telefony komórkowe zostały pokazane w 1973r. w Nowym Jorku. Telefony zostały wykonane przez firmę Motorola, były dużych rozmiarów i ważyły około 2 kilogramy [1]. W 1979 r. Japonia uruchomiła pierwszą na świecie sieć telefonii komórkowej. Parę lat później,
w roku 1983 został wypuszczony na rynek pierwszy, masowo produkowany telefon komórkowy DynaTAC 8000x.

W następnych latach technologia urządzeń mobilnych szybko się rozwijała, głównie w kierunku rozmów głosowych oraz wiadomości tekstowych, do momentu prezentacji iPhone z systemem iOS oraz opublikowania pierwszej wersji systemu operacyjnego Android w roku 2007. Od tego momentu, rozwój urządzeń mobilnych kierował się w stronę wielofunkcyjnych urządzeń multimedialnych. Obecnie, obydwa systemy wyparły konkurencję i zajmują odpowiednio 29,24% oraz 70,01% urządzeń na rynku światowym [2].

Wraz z rozwojem technologii i rozpowszechnieniem urządzeń mobilnych wśród gospodarstw domowych, wzrosło zapotrzebowanie na coraz wydajniejsze urządzenia. Rozwiązaniem wydają się nowoczesne podzespoły, a także zoptymalizowane sposoby pisania nowych aplikacji. W tym artykule, starano się pokazać wpływ dwóch sposobów wdrażania interfejsu użytkownika na wydajność urządzenia mobilnego. Została postawiona teza, że interfejs użytkownika stworzony za pomocą Jetpack Compose jest wydajniejszy od dotychczasowych alternatyw. Potwierdzono wpływ zastosowania odpowiednio zoptymalizowanego kodu oraz techniki wykonania interfejsu użytkownika na jego wydajność.

2. Przegląd literatury

Autorzy artykułu „How resource utilization influences UI responsiveness of Android software” [3] prowadzali testy wytrzymałościowe oraz testy na aplikacji na urządzeniach z systemem Android w celu zrozumienia, jak wykorzystanie zasobów wpływa na responsywność systemu Android. Z przeprowadzonych badań wywnioskowano, że pojemność urządzenia oraz wielkość pamięci wpływają na responsywność UI tylko wtedy, gdy ich użycie zbliża się do 100%, natomiast najważniejszym aspektem dla wydajności aplikacji jest odpowiednie zarządzanie priorytetami wątków.

Następnym wybranym artykułem jest „Performance of Hybrid Mobile Application UI Frameworks” [4], skupiającym się na porównaniu najczęściej stosowanych, hybrydowych interfejsach użytkownika dla aplikacji mobilnych. Na zbudowanych aplikacjach przeprowadzono testy wydajnościowe, gdzie szczególną uwagę zwracano na szybkość ładowania elementów, płynność przewijania i płynność przechodzenia między stronami. W badaniu wzięło udział 4 ekspertów, którzy ocenili kryteria w stosunku od najmniej ważnych do najważniejszych. Do wyników wykorzystano obliczone wagi kryteriów. W otrzymanych wynikach zauważono korelację pomiędzy prostotą używanej biblioteki, a jej wydajnością.

Artykuł „The Effect of Representational UI Design Quality of Mobile Shopping Applications on Users’ Intention Shop” [5] bada wpływ zwięzłości i spójności interfejsu użytkownika na użyteczność aplikacji i użytkowników. W badaniu wzięło udział ponad 230 studentów,
a otrzymane wyniki potwierdziły założone hipotezy – zwięzły i spójny UI pozytywnie wpływa na użyteczność aplikacji oraz intencję użytkowników do ponownego korzystania z aplikacji do zakupów.

Możliwość dostosowania interfejsu użytkownika jest również ważnym aspektem, który został zbadany w artykule pt. „Enhancing user experience through customization of UI design” [6]. W eksperymencie wzięło udział 50 studentów w wieku 18-23 lat, którzy odpowiadali na pytania z kwestionariusza dot. Komfortu korzystania
z UI. Na podstawie otrzymanych wyników, wywnioskowano, że możliwość dostosowania interfejsu przez użytkownika wpływa na komfort jego użytkowania.

Ostatnim artykułem odnoszącym się do niniejszej pracy jest „User Interface Matters: Analysing the Complexity of Mobile Applications from a Visual Perspective” [7]. Autorzy zaproponowali analizę interfejsu poprzez wprowadzenie odpowiednich wskaźników. W celu pokazania ich użyteczności, oceniono za ich pomocą aplikacje mobilne napisane podczas kursów programowania. Pozwoliły w prosty i przejrzysty sposób pokazać złożoność interfejsu, logiki i wydajności.

W powyższej literaturze, nie znaleziono porównań takich jak w niniejszym artykule, jednakże, stanowiły one bazę, na której zostały opracowane oraz zinterpretowane badania użyte w niniejszym artykule.

3. Plan i kryteria badań

Badania zostały przeprowadzone na aplikacji mobilnej napisanej w języku Kotlin [8] na system operacyjny Android. Aplikacja została podzielona na wiele aktywności, w skład których wchodzi lista z elementami (Listingi 1 - 3) oraz krótka animacja (Rysunek 1). Lista elementów została napisana przy użyciu Jetpack Compose [9], widoków oraz ich kombinacji. Do uruchomienia animacji jest wymagana interakcja użytkownika z wyświetlaczem urządzenia. Polega ona na przemieszczeniu w losowe miejsce 10 kolorowych kółek o identycznej średnicy. Do stworzenia animacji, ponownie wykorzystano zestaw Jetpack Compose oraz widoki wraz z elementami zapisanymi w formacie XML.

Listing 1: Część kodu odpowiedzialna za wyświetlanie listy elementów za pomocą widoków

```
@Composable
fun List(listContent: List<DataRow>){
    LazyColumn{ this: LazyListScope
        items(listContent.size){ index ->
            ListRow(listContent[index])
        }
    }
}

@Composable
fun ListRow(obj: DataRow) {
    Row{ this: RowScope
        Column(modifier = Modifier.padding(10.dp)){ this: ColumnScope
            Text(text = obj.id.toString(), fontSize = 30.sp)
        }
        Column{ this: ColumnScope
            Row{ this: RowScope
                Text(text = "test", fontSize = 18.sp)
            }
            Row{ this: RowScope
                Text(text = obj.content, fontSize = 18.sp)
            }
        }
    }
}
```

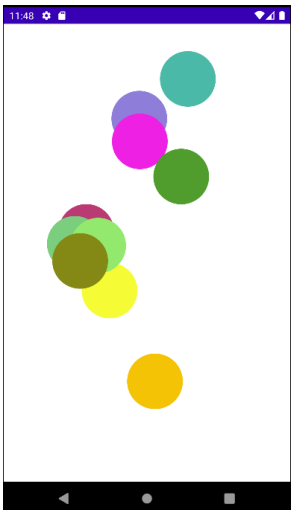
Listing 2: Część kodu odpowiedzialna za wyświetlenie listy elementów za pomocą widoków

```
    val linearLayoutManager = LinearLayoutManager( context: this)
    val adapter = ListAdapter(SampleData)
    val recyclerView = findViewById<RecyclerView>(R.id.itemList)
    recyclerView.layoutManager = linearLayoutManager
    recyclerView.adapter = adapter
```

Listing 3: Część kodu odpowiedzialna za wyświetlenie listy elementów za pomocą kombinacji Jetpack Compose i widoków

```
val composeView = findViewById<ComposeView>(R.id.compose_view)
val sampleData = SampleData.listContent

composeView.setContent{
    LazyColumn{ this: LazyListScope
        items(sampleData.size){ this: LazyItemScope
            ListRow(sampleData[it])
        }
    }
}
```

Do odpowiedniego przeprowadzenia eksperymentu zastosowano narzędzie Macrobenchmark [10] zintegrowany z IDE Android Studio do przeprowadzania testów wydajnościowych na platformę Android. Przy pomocy bibliotek JUnit oraz UIAutomator zostało przeprowadzone 30 testów mierzących czas uruchomienia aplikacji oraz 9 testów zliczających czas potrzebny procesorowi do wyrenderowania pojedynczej klatki. Każda aplikacja, została poddana identycznej liczbie testów obciążających interfejs jednakowymi działaniami, z których wyliczono średnie statystyki dotyczące wydajności.



Rysunek 1: Przykładowe położenie kulek po uruchomieniu animacji.

Przeprowadzone testy brały pod uwagę różne tryby uruchomienia aplikacji. Pierwszym z nich jest „tryb zimny" (ang. Cold), dla którego wpierw został tworzony proces aplikacji, a następnie utworzona nowa aktywność. Następnie badano „tryb ciepły" (ang. Warm), dla którego proces aplikacji istnieje, lecz jest potrzebne stworzenie aktywności. Ostatnim sposobem uruchamiania aplikacji jest „tryb gorący" (ang. Hot), gdzie istniejąca aktywność jest przenoszona na pierwszy plan, w już działającym procesie.

Badania zostały powtórzone na fizycznym urządzeniu, działającym pod kontrolą systemu Android. Wspomniane urządzenia mobilne posiadały różne parametry, tj. wersja systemu operacyjnego, procesor, procesor graficzny, pamięć RAM, rozdzielczość oraz liczba pikseli przypadających na cal długości wyświetlacza (Tabela 1).

Tabela 1: Parametry urządzeń mobilnych

	Xiaomi Redmi 9	Samsung Galaxy A3	Motorola Moto G50
Model	M2003J15SG	SM-A300FU	Moto G(50)
CPU	Octa-core (2x2.0 GHz Cortex-A75 & 6x1.8 GHz Cortex-A55)	Quad-core 1.2 GHz Cortex-A53	Gualcomm Snapdragon 480 8x2.00 GHz
GPU	Mali-G52 MC2	Adreno 306	Adreno 619
RAM	4 GB	1,5 GB	4 GB
System operacyjny	Android 11 RP1A.200720.011 (API level 30)	Android 6.0.1 (API level 23)	Android 12 S1RFS32.27-25-1 (API level 31)
Wyświetlacz	1080x2340 px, 395 ppi, 60Hz	720x1280px, 312 ppi, 60Hz	720x1600px, 270ppi, 90Hz

Przeprowadzone testy skupiały się głównie na zmierzeniu czasu potrzebnego urządzeniu, na przeprowadzenie danej operacji na interfejsie użytkownika, w danej technologii. Czasy wykonania testów zostały zmierzone za pomocą narzędzia Macrobenchmark, zintegrowanego z IDE Android Studio. Pierwszy z przeprowadzonych testów miał za zadanie zmierzyć czas potrzebny urządzeniu na uruchomienie aplikacji oraz przejście do aktywności z badanym interfejsem. Przykładowa część kodu została przedstawiona poniżej (Listing 4).

Następne przeprowadzone testy miały za zadanie zmierzyć czas zbudowania pojedynczej klatki przez procesor. W tym wypadku, ostatecznym wynikiem jest suma dwóch wątków, pracujących nad budową interfejsu graficznego. Do badania przygotowano dwie, testowane aktywności. Pierwsza z nich zawierała dynamiczną listę z elementami, druga – animację wykonaną w danej technologii. Na tych aktywnościach, zasymulowano aktywność użytkownika za pomocą biblioteki UIAutomator (Listingi 5 – 6).

Listing 4: Część kodu odpowiedzialna za mierzenie czasu startowego aplikacji

```
@LargeTest
@RunWith(Parameterized::class)
class StartupBenchmark(private val startupMode: StartupMode) {
    @get:Rule
    val benchmarkRule = MacrobenchmarkRule()

    @Test
    fun startup() = benchmarkRule.measureRepeated(
        packageName = TARGET_PACKAGE,
        metrics = listOf(StartupTimingMetric()),
        compilationMode = CompilationMode.DEFAULT,
        startupMode = startupMode,
        iterations = ITERATIONS,
        setupBlock = { this: MacrobenchmarkScope
            pressHome()
        }
    ){ this: MacrobenchmarkScope
        val intent = Intent()
        intent.action = "$packageName.ComposeActivity"
        startActivityAndWait(intent)
    }
```

Listing 5: Część kodu odpowiedzialna za symulację przewijania listy

```
){ this: MacrobenchmarkScope
    val column = device.findObject(By.scrollable( isScrollable: true))

    scrollList(column, device)
}
```

Listing 6: Część kodu odpowiedzialna za symulację kliknięć na ekranie

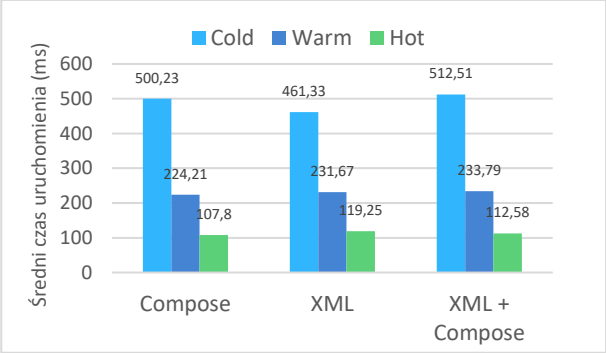
```
){ this: MacrobenchmarkScope
    val canvas = device.findObject(By.clazz( className: "android.view.View"))

    for(i in 0 .. 4){
        canvas.click()
        sleep( millis: 2000)
    }
}
```

4. Wyniki

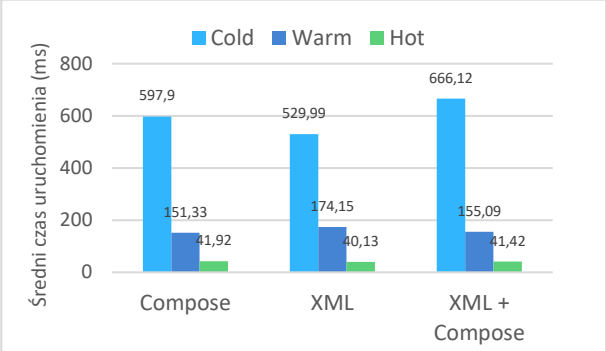
Początkowo zostały przeprowadzone testy mierzące czas startowy aplikacji. Badanie przeprowadzono na liście ze statycznymi elementami. Otrzymane wyniki zostały przedstawione poniżej. Na podstawie otrzyma-

nych wyników, zostały przeprowadzone odpowiednie obliczenia (Rysunki 2 – 4).

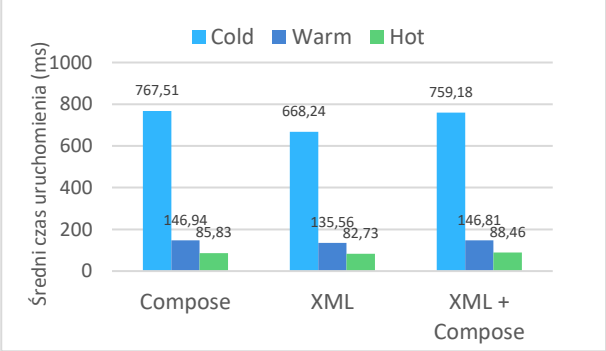
Biorąc pod uwagę poniższe wyniki, można zauważyć, że najlepsze, ogólne wyniki uzyskała aktywność zbudowana przy pomocy widoków i plików XML. Niezależnie od typu urządzenia, posiada również najniższy czas uruchomienia pierwszego, „zimnego" startu aplikacji. Jetpack Compose osiąga nieco gorsze wyniki, oscylując w granicach 5% - 10% wyników poprzednika. Najgorzej w całym zestawieniu poradziła sobie kombinacja obu technologii, osiągając największy czas uruchomienia dla większości trybów startowych.



Rysunek 2: Wyniki pomiarów czasu startowego aplikacji dla telefonu Xiaomi Redmi 9.



Rysunek 3: Wyniki pomiarów czasu startowego aplikacji dla telefonu Samsung Galaxy A3.



Rysunek 4: Wyniki pomiarów czasu startowego aplikacji dla telefonu Motorola Moto G50.

Następnym przeprowadzonym badaniem jest zliczenie czasu budowania pojedynczej klatki przez procesor. W tym celu badania zostały przeprowadzone na liście elementów, która podczas testów była automatycznie przewijana. Każdy test został wykonany na „zimnym" trybie startowym. Ponieważ interfejs użytkownika jest generowany na dwóch wątkach MainThread oraz RenderThread, zaprezentowane niżej wyniki, są sumą czasów działania tych wątków. Zostały one przedstawione w formie średniej arytmetycznej, odchylenia standardowego, percentyli wielkości 50, 90, 95 i 99 oraz wartości minimalnych i maksymalnych (Tabele 2-4).

Z otrzymanych wyników wynika, że technologia, w której wykonany został interfejs użytkownika nie ma wielkiego wpływu na jego płynność. Różnica pomiędzy wynikami z danych metod jest niewielka. Największe odchyły od pomiarów wykazał jednak Jetpack Compose, co może powodować niestabilność pracy w wybranych przedziałach czasowych. Najmniejszym czasem budowania klatki podczas animacji przewijania listy, cechuje się kombinacja obu badanych technologii. Osiągnęła ona najstabilniejsze i najszybsze wyniki spośród badanych.

Tabela 2: Wyniki czasu potrzebnego do zbudowania pojedyńczej klatki przez procesor dla telefonu Xiaomi Redmi 9

	Compose	XML	XML + Compose
P50	23,94 ms	29,13 ms	13,66 ms
P90	30,82 ms	32,71 ms	30,06 ms
P95	32,51 ms	33,85 ms	31,27 ms
P99	61,53 ms	61,27 ms	48,82 ms
Śr.	22,61 ms	27,19 ms	17,34 ms
Odch. St.	10,99 ms	9,44 ms	8,42 ms
Min	10,32 ms	8,45 ms	8,73 ms
Max	125,58 ms	90,05 ms	64,15 ms

Tabela 3: Wyniki czasu potrzebnego do zbudowania pojedyńczej klatki przez procesor dla telefonu Samsung Galaxy A3

	Compose	XML	XML + Compose
P50	9,20 ms	10,11 ms	8,95 ms
P90	16,07 ms	16,45 ms	15,94 ms
P95	21,24 ms	20,85 ms	21,25 ms
P99	34,01 ms	28,61 ms	34,32 ms
Śr.	11,12 ms	11,12 ms	10,82 ms
Odch. St.	5,49 ms	5,05 ms	5,55 ms
Min	4,83 ms	4,83 ms	2,66 ms
Max	48,56 ms	51,41 ms	54,74 ms

Tabela 4: Wyniki czasu potrzebnego do zbudowania pojedyńczej klatki przez procesor dla telefonu Motorola Moto G50

	Compose	XML	XML + Compose
P50	6,13 ms	5,59 ms	6,12 ms
P90	8,83 ms	7,53 ms	8,66 ms
P95	10,65 ms	9,13 ms	10,20 ms
P99	24,80 ms	13,39 ms	21,22 ms
Śr.	6,57 ms	5,94 ms	6,45 ms
Odch. St.	4,02 ms	1,83 ms	3,54 ms
Min	2,32 ms	2,38 ms	2,63 ms
Max	46,33 ms	16,84 ms	39,63 ms

Powyższe badanie, zliczające czas zbudowania pojedynczej klatki, zostało również wykonane na aktywnościach zawierających ok. 10-sekundową animację, poruszających się kół. Zebrane wyniki przedstawiono poniżej (Tabele 5-7).

Tabela 5: Wyniki czasu potrzebnego do zbudowania pojedynczej klatki przez procesor dla telefonu Xiaomi Redmi 9

	Compose	XML
P50	11,55 ms	7,39 ms
P90	14,60 ms	8,29 ms
P95	15,64 ms	8,72 ms
P99	17,49 ms	10,48 ms
Śr.	12,04 ms	7,23 ms
Odch. St.	6,34 ms	1,17 ms
Min	5,06 ms	2,59 ms
Max	143,95 ms	14,14 ms

Tabela 6: Wyniki czasu potrzebnego do zbudowania pojedynczej klatki przez procesor dla telefonu Samsung Galaxy A3

	Compose	XML
P50	10,02 ms	7,34 ms
P90	16,39 ms	10,58 ms
P95	20,63 ms	12,96 ms
P99	28,49 ms	15,54 ms
Śr.	11,26 ms	7,92 ms
Odch. St.	5,33 ms	2,14 ms
Min	4,48 ms	4,22 ms
Max	84,67 ms	24,41 ms

Tabela 7: Wyniki czasu potrzebnego do zbudowania pojedynczej klatki przez procesor dla telefonu Motorola Moto G50

	Compose	XML
P50	6,85 ms	5,57 ms
P90	8,38 ms	7,29 ms
P95	9,12 ms	7,72 ms
P99	13,60 ms	8,83 ms
Śr.	6,63 ms	5,57 ms
Odch. St.	2,83 ms	1,36 ms
Min	2,75 ms	1,62 ms
Max	55,62 ms	10,72 ms

Wyniki ostatniego badania ponownie pokazują niewielką niestabilność pracy podczas animacji wykonanych w Jetpack Compose. Osiąga zarówno najwyższy, maksymalny jak i minimalny czas budowania klatki od alternatywnej opcji. Pomimo tego, obydwie metody wykonały płynne animacje, z małą różnicą pomiędzy ich średnim czasem wykonania zadania.

6. Wnioski

Przeprowadzone badania pozwalają na wyciągnięcie następujących wniosków:

Zastosowanie widoków w aplikacji jest efektywniejszą opcją od zestawu narzędziowego Jetpack Compose, oraz związanymi z nim opcjami. Zarówno czas startowy, jak i czas potrzebny do wyrenderowania pojedynczej klatki okazał się mniejszy w aktywnościach zbudowanych przy pomocy widoków. Nie dotyczy to aplikacji działających głównie w tle, gdzie przywrócenie ich na pierwszy plan zajmuje krócej, gdy interfejs użytkownika jest zbudowany w Jetpack Compose.

Stabilność animacji wydaje się problemem w obecnej wersji 1.1.1 Jetpack'a Compose. Starsze urządzenia osiągają znacznie gorsze wyniki korzystając z animacji Compose. Wpływ na to mogą mieć na to zarówno parametry testowanych urządzeń fizycznych, jak i obecne w testowanej wersji Jetpack Compose, eksperymentalne animacje. W pozostałych przypadkach, obie metody osiągają podobne rezultaty.

Literatura

[1] Wywiad z Martinem Cooperem, wynalazcą telefonów komórkowych, http://news.bbc.co.uk/1/hi/programmes/click_online/8639590.stm, [21.09.2022].

[2] Udział w rynku mobilnych systemów operacyjnych na świecie, https://gs.statcounter.com/os-market-share/mobile/worldwide, [26.01.2021].

[3] J. Fu, Y. Wang, Y. Zhou, X. Wang, How resource utilization influences UI responsiveness of Android software, Information and Software Technology 141 (2022)1-11, https://doi.org/10.1016/j.infsof.2021.106728.

[4] R. Vala, R. Jasek, Performance of Hybrid Mobile Application UI Frameworks, in: V. Mladenov, I.Rudas, O. Martin, G. Tsenov, P. M. Pardalos, M. Hromada, Proceedings of the 2014 Intertnational Conference on Applied Mathematics, Computational Science & Engineering (AMCSE 2014), Varna, Bulgaria, September 13-15, 2014.

[5] W. Jung, The Effect of Representational UI Design Quality of Mobile Shopping Applications on Users Intention to Shop, Procedia Computer Science 121 (2017) 166-169.

[6] S. L. T. Hui, S. L. See, Enhancing user experience through customisation of UI design, Procedia Manufacturing 3 (2015) 1932–1937.

[7] L. Corrala, I. Fronzab, T. Mikkonenc, User Interface Matters: Analysing the Complexity of Mobile Applications from a Visual Perspective, Procedia Computer Science 191 (2021) 9-16.

[8] Informacje o języku kotlin, https://kotlinlang.org/docs/home.html, [21.09.2022].

[9] Informacje o narzędziu Jetpack Compose, https://developer.android.com/jetpack/compose/documentation, [21.09.2022].

[10] Dokumentacja narzędzia Macrobenchmark, https://developer.android.com/topic/performance/benchmarking/macrobenchmark-overview, [21.09.2022].