Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Vývoj Android aplikací s Jetpack Compose

Bakalářská práce

Autor: Daniel Pitropovski
Studijní program: Aplikovaná informatika

Vedoucí práce: doc. Mgr. Tomáš Kozel, Ph.D

Hradec Králové Duben 2024

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

vlastnoruční podpis

V Hradci Králové dne 19.04.2024 Daniel Pitropovski

Poděkování:

Rád bych poděkoval panu doc. Mgr. Tomáši Kozlovi, Ph.D. za jeho odborné vedení a cennou pomoc při zpracování této práce.

Anotace

Název: Vývoj Android aplikací s Jetpack Compose

Tématem této bakalářské práce je vývoj nativní Android aplikace pomoci programovacího jazyku Kotlin a frameworku Jetpack Compose. Práce se snaží prozkoumat přednosti a výhody použití této sady nástrojů pro vytváření UI komponent.

Teoretická část právě definuje jednotlivé komponenty a funkce které jsou potřebné pro vývoj jedné aplikace. Účelem této časti je poskytnout čtenářům důkladný přehled o funkcionalitě Jetpack Compose a definici každého z jeho části uživatelského rozhraní.

Na základě teoretického základu, který bude uveden první části bakalářské práce, se bude v praktické části vytvářet aplikace s názvem Yamb, která zkombinuje některé komponenty a funkce které Jetpack Compose poskytuje.

Vývojáři pro vývoj mobilních aplikací, kteří se obávají přechodu na Jetpack Compose a chtějí se lépe seznámit s jeho výhodami a přínosy, mohou výsledky této práce využít jako základ.

Klíčová slova: Jetpack Compose, Kotlin, API, Uživatelské rozhraní, Deklarativní UI, Android, Mobilní aplikace.

Annotation

Title: Android Application Development using Jetpack Compose

The topic of this bachelor's thesis is the development of a native Android application using the Kotlin programming language and the framework Jetpack Compose. The objective of the thesis is to investigate the advantages and benefits of using this toolkit to create UI components.

The theoretical part defines the individual components and functions that are needed for the development of an application. The purpose of this section is to provide readers with a thorough overview of Jetpack Compose's functionality and a definition of each part of its user interface.

Based on the theoretical basis that will be presented in the first part of this bachelor's thesis, in the practical part there will be instructions on how to create such an application called Yamb, which will combine some components and functions that are provided by Jetpack Compose.

Mobile app developers who are apprehensive about switching to Jetpack Compose and want to better understand its advantages and features can use the results of this work as a baseline.

Keywords: Jetpack Compose, Kotlin, API, User interface, Declarative UI, Android, Mobile application.

Obsah

Seznam obrázků

1 Úvod

Mobilní aplikace se v současné digitální éře stály nezbytnou částí naších životů a pokrývají řadu funkcí od komunikace a zábavy až po produktivitu a nakupování. Aplikace zjednodušují lidem život tím, že jím šetří čas, úsilí a zároveň neustále lidem pomáhají vykonávat každodenní činnosti.

Kvůli své mobilitě a velké komunitě vývojářů, cesta pro vývoj aplikací začala s programovacím jazykem Java, který byl vybrán jako oficiální programovací jazyk. Java, jazyk, který již existuje přes 25 let, sloužil jako základ pro vývoj aplikací a dával programátorům pevnou základnu. Java tak byla oblíbenou volbou kvůli objektovému orientovanému programování a její známosti. Bohužel od doby, kdy se Android rozšiřoval, se objevilo mnoha problémů. Jeden z problémů byl boilerplate kód, což znamenalo že vývojáři museli psát stále opakující se kód, tudíž kód v projektu byl velice nepřehlední a zároveň to mělo špatný vliv na optimalizaci. Další zásadní problém byl v tom že vývojáři museli definovat rozvržení uživatelského rozhraní pomocí souboru XML, které byly oddělené od kódu Java. Oddělení UI komponent od kódu Java zkomplikovalo vývojový proces. Proto postupem času se hledal alternativní přistup, jak efektivněji vytvářet aplikace.

Vzhledem k tomu, že se technologie výrazně denně zlepšuje, je vývoj aplikací čím dál tím složitější. Navíc, protože mnoho API a funkcí není zcela zpětně kompatibilní, je stále obtížnější vytvořit kód který podporuje všechny verze Androidu. To by znamenalo, že kód, který by byl vytvořen v nové verzi Androidu by nemusel fungovat na starší verzi. Tohle jsou důvody pro vývoj nového programovacího jazyku a frameworku jako je Kotlin a Jetpack Compose. Tyto nástroje jsou navrženy tak, aby zjednodušily proces vytváření aplikací a pomohly vývojářům držet krok s neustále se měnícími požadavky na trhu. V současném rychle se rozvíjejícím technologickém prostředí poskytují tyto špičkové nástroje a jazyky vývojářům vyšší produktivitu, flexibilitu a efektivitu, což v konečném důsledku pomáhá vytvářet lepší a spolehlivější programy. Jetpack Compose má také jednu zásadní věc, a to je jeho deklarativní přístup, který usnadňuje udržování uživatelského rozhraní v průběhu času a přispívá ke konzistenci uživatelského rozhraní.

2 Cíl práce

Cílem této bakalářské práce je, aby se její čtenáři seznámili s revolučním frameworkem Jetpack Compose. Také má za cíl poskytnout začínajícím Android vývojářům znalosti a dovednosti potřebné k plnému využití možností tohoto frameworku tím, že v teoretické části se poskytne důkladný přehled Jetpack Compose. Se znalosti, které čtenář získá, bude moci s jistotou vytvářet vizuálně přitažlivé a funkcionální aplikace pro Android. Práce obnáší nejen teorii, ale také praktický příklad, jak takovou aplikaci vytvořit.

Teoretická část

3 Kotlin

Oblast vývoje softwaru zaznamenala významný růst použitím pragmatického programovacího jazyka Kotlin, který organizace JetBrains začala vyvíjet v roce 2010 a oficiální oznámený bylo v červnu 2011 jako open-source programovací jazyk, pod názvem „Project Kotlin“. [6] Kotlin navazuje na Javu v tradici pojmenování programovacích jazyků po ostrovech, což znamená, že tento jazyk je pojmenován po ostrově v Baltském moří. Kotlin je navržen tak, aby byl bezpečnější než mnoho jiných programovacích jazyků, je snadnější ke čtení, zápisu a obojí zároveň. Hlavním cílem jazyka Kotlin je vytvořit bezpečný a stručný kód. Kód je obvykle považován za stručný, když je snadno pochopitelný a srozumitelný a taky kód, který lze psát rychleji a efektivněji. Kotlin má řadu funkcí, které zvyšují pravděpodobnost, že potenciální problémy budou nalezeny během psaní kódu spíše, než že by vedli k run-time pádům. [1, s. 87]

Je důležité poznamenat, že tato bakalářská práce nezahrnuje vše, co obsahuje programovací jazyk Kotlin, protože předpokládá že čtenář má již nějaké základy o programování. Zde je však několik zásadních témat, která budou zmíněna, které čtenář bude muset pochopit, před tím, než se seznámí s Jetpack Compose. Rozhodnutí zaměřit se na základy Kotlinu před ponořením se do Jetpack Compose, je záměrné, protože pokládá silný základ pro pochopení pokročilých témat, která budou probrána později v této práci.

3.1 Proměnné

Datové hodnoty jsou uloženi v kontejnerech nazývaných proměnné. V programovacím jazyku Kotlin existují dva typy proměnných na základě jejich vlastností a jak se chovají v programu, a to jsou proměnné typu „val“ a „var“.

```
val jmeno:String="Daniel"
var vek:Int=22
```

Proměnné typu „var“ jsou takzvané „Mutable“ (proměnlivé). To znamená že hodnota v průběhu aplikace se muže změnit. Na druhou stranu proměnné typu „val“ jsou „Immutable“ (neměnné), a chovají se jako konstanty. Tohle indikuje že se

hodnota v průběhu aplikace nemůže změnit. Pokud je hodnota používaná opakovaně v kódu aplikace, jsou tyhle neměnné proměnné obzvláštně užitečné. [1, s. 94] Kotlin podporuje používaní neměnných proměnných (val) před proměnnými (var), kdykoli je to praktické, protože zlepšují bezpečnost, čitelnost a předvídatelnost kódu. Neměnné proměnné jsou ve výchozím nastavení Thread-safe[1], pomáhají s redukcí chyb a zjednodušují kódovaní. Navíc poskytují specifické optimalizace kompilátoru a jsou v souladu se současnými programovacími standardy. [7]

3.2 Nullable Types

Funkce typu s možností „null“ je pro Kotlin unikátní a skoro jedinečná a chybí ve většině programovacích jazyků. Nullable Type jsou navrženy tak, aby nabízely bezpečnou a spolehlivou metodu řešení okolností, ze kterých může být proměnné přidělena nulová hodnota. Jinými slovy, cílem je zabránit rozšířenému problému s výjimkami nulového ukazatele (NullPointerException) ke kterým dochází, když kód naběhne na hodnotu „null“, když se to nepředpokládalo. [1, s. 96] Tahle výjimka je jedním z nejčastějších problémů s programovacími jazyky a není vždycky lehké ji vyřešit. Java sloužící jako předchůdce Kotlinu ve světě programovacích jazyků, hrála klíčovou roli při inspirování implementace této funkce. Vývojáři si uvědomili, jaké výzvy mohou „null“ hodnoty přinést. Kotlinův vývoj tak položil pevný základ pro řešení těchto problémů.

```
var priklad:String?=null //přidělení null hodnoty

priklad="priklad 1" //přidělení hodnoty
```

V Kotlinu je možné přiřadit proměnné nulovou hodnotu tím že ji deklarujeme s typem, který umožnuje „null“ hodnoty. K tomu slouží přiřazení znaku „?“ za typem. Pokud proměnná nebude obsahovat znak „?“ tak se nemůže „null“ hodnota přidělit.

```
var priklad2:String="priklad2" //přidělení hodnoty

priklad2=null//nejde přidělit null hodnotu-chyba při kompilaci
```

[1] Thread-safe znamená, že program nebo systém dokáže zpracovat současný přístup a úpravu sdílených dat několika vlákny, aniž by došlo k poškození dat nebo neočekávanému chování.

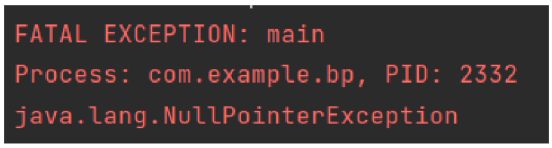
Proměnnou, která má hodnotu „null“, lze ale použít stejným způsobem jako proměnnou bez hodnoty „null“, bez toho, aby nastaly komplikace při kompilaci. Tohle lze dosáhnout pomoci implementace dvojitých vykřičníků (!!). [1, s. 97]

```
var priklad3:String?=null

var velikost= priklad3!!.lenght()

print(velikost)
```

I když tento příklad na začátku vypadá v pořádku, problém vznikne při třetím řádku, kde používáme „print(velikost)“. V této chvíli může dojít k výjimce typu „NullPointerException“, což je za důsledek manipulace s „null“ hodnotami touto metodou.



Obrázek 1 Výjimka typu „NullPointerException“
Zdroj: Autor

3.3 Lambda & Trailing lambda

Klíčovou součástí programovacího jazyka Kotlin a Jetpack Compose, jsou výrazy lambda, včetně trailing lambda (koncových lambd). Jetpack Compose využívá výraznou syntaxi Kotlin, aby nabídl uživatelsky přívětivější a efektivnější způsob vytváření uživatelských rozhraní. Výrazy lambda, včetně koncových lambd, jsou často využívány frameworkem Jetpack Compose k vytváření komponent uživatelského rozhraní a specifikaci jejich chování. Toto připojení kromě zefektivnění procesu vývoje uživatelského rozhraní zlepšuje čitelnost kódu a jeho udržovatelnost. Použitý těchto výrazů umožnuje vývojářům vyjádřit logiku uživatelského rozhraní přirozenějším a intuitivnějším způsobem. Předtím než dojde k ukázce implementace lambda funkce je potřeba nejdřív pochopit, jak by vypadala původní funkce. V následujícím kódu je definována funkce s názvem „soucet“, která

přebírá dva parametry „a“ a „b“ typu „Integer“ (celočíselný typ) a vrací jejich součet.

```
//Definování normalní funkce,která sčítá dvě čísla
fun soucet(a: Int, b: Int): Int {
    return a + b
}

//výsledek, který bude uložen do této proměnné
val vysledek = soucet(5, 5)

// Tisk výsledku
println("Výsledek: $vysledek")
```

```
// Definování lambda funkce,která sčíta dvě čísla
val soucet: (Int, Int) -> Int = { a, b -> a + b }

//výsledek, který bude uložen do této proměnné
val vysledek = soucet(5, 5)

// Tisk výsledku
println("Výsledek: $vysledek")
```

Funkce lambda jsou stručné, anonymní funkce, což znamená že se ne definuje jméno funkce. Funkce lambda jsou definovány na místě, často pro specifické, krátké úlohy, pomoci kompaktní syntaxe. Normální funkce na druhou stranu jsou pojmenovány a mají podrobnější syntaxi, a tak jsou vhodné pro delší, složité a opakovaně použitelné operace.

„*V Kotlinu „trailing lambda“ označuje syntaktickou funkci, která vám umožňuje přesunout výraz lambda mimo závorky volání funkce, pokud je to poslední argument této funkce.*“ [8] Z definice je nutné podotknout že jako poslední parametr funkce je také funkce, pokud by funkce nebyla poslední parametr tak by nešlo přesunout výraz lambda mimo závorky. Příklad, jak by vypadala implementace trailing lambdy:

```
fun soucetCisel(a: Int, b: Int, operace: (Int, Int) -> Int): Int {
    return operace(a, b)
}

val vysledek = soucetCisel(5, 5) { a, b ->
    a + b
}
// Tisk výsledku
println("Výsledek: $vysledek")
```

Tato část kódu definuje funkci s názvem „soucetCisel“, která přebírá tři argumenty: dvě celá čísla, „a“ a „b“ a „operace“ jako lambda funkce, která přijímá dvě celá čísla a vrací celé číslo. Tato funkce vezme dva stupy, provede s nimi operaci a poté vrátí výsledek.

Protože výrazně zvyšují čitelnost a stručnost kódu, jsou trailing lambdy nezbytnou částí Jetpack Compose. Trailing lambdy umožnují vývojářům pochopení a údržbu jejich kódu uživatelského rozhraní deklarativnějším způsobem. V podstatě trailing lambdy tvoří základ snahy Jetpack Compose zefektivnit a posunout vytváření uživatelských rozhraní pro aplikací Android.

3.4 Coroutines

Nejdříve je potřeba zmínit a definovat co je synchronní a asynchronní programování. V synchronním programování se příkazy provádějí postupně, jeden po druhém a nelze je zastavit, dokud nejsou hotové. Každá úloha musí před spuštěním počkat na dokončení provedení úlohy před ní. Co se týče v asynchronním programování, lze úlohy provádět v libovolném pořadí nebo dokonce souběžně. Pokud je úkol zdlouhavý, složitý, nebo vyžaduje několik zdrojů, tato strategie funguje dobře. Funguje to tak, že pro ten úkol se vytvoří nové vlákno, které nepřekáží hlavnímu vláknu, a tak nepřekáží operacím které se tam vykonávají. [9]

Vzhledem k tomu že úlohy lze provádět v určitém pořadí, je jednoduší sledovat synchronní programy. Toto je však ale důvod, proč jsou tyhle programy pomalejší a méně efektivní. Na druhou stranu asynchronní programy lze provádět souběžně a bez čekání na sebe, znamená to, že jsou rychlejší a efektivnější. Hlavní nevýhodou je že jsou složitější na implementaci a jsou náchylné k chybám. Jako vždy by měl programátor vybrat optimální programovací model na základě požadavků. Správný výběr mezi synchronním a asynchronním programováním vyžaduje pečlivé zvážení rozdílů mezi oběma přístupy.

Běhový systém vytvoří při prvním spuštění aplikace pro Android jedno vlákno, na kterém se ve výchozím nastavení spustí všechny součásti programu. Primární vlákno je obvykle označováno jako toto. Hlavní odpovědností hlavního vlákna je řídit událostí uživatelského rozhraní a interakce s jeho pohledy. Pokud některý kód v aplikaci používá hlavní vlákno k provádění časově náročné činnosti, bude se zdát, že je celá aplikace zablokována, dokud úloha nebude dokončena. V důsledku toho se uživateli obvykle zobrazí oznámení „Application *is not responding*“, neboli že aplikace neodpovídá. Pro jakoukoli aplikaci je to daleko od

požadovaného chování. Dobrou zprávou je, že Coroutines v Kotlinu nabízí jednoduchou náhradu. [1, s. 261]

Coroutines v oficiální Kotlin dokumentaci jsou definovány tímto způsobem: „Coroutines *jsou lehká vlákna, která vám umožňují psát asynchronní neblokující kód. Kotlin poskytuje knihovně řadu primitiv s podporou coroutines na vysoké úrovni.*“ [10] Aniž by se programátor musel obtěžovat vytvářením složitých multitaskingových řešení nebo přímým řízením mnoha vláken, lze implementovat coroutines, které jsou mnohem efektivnější z hlediska zdrojů a času než použití konvenčních více vláknových metod díky způsobu, jakým jsou konstruovány. Coroutines také umožnují sekvenční zápis kódu bez nutnosti zpětných volání pro zpracování událostí a výsledku souvisejících vlákny, takže výsledný kód je podstatně jednodušší na psaní, pochopení a také pro údržbu. [1, s. 261]

3.4.1 Porovnání vláken a coroutines

Po velmi dlouhou dobu byla vlákna nezbytnou součástí paralelního programování. Umožnují paralelní provádění mnoha procesů, což umožnuje programátorům dokončit časově náročnou práci bez blokování hlavního vlákna, které řídí uživatelské interakce a udržuje odezvu uživatelského rozhraní. [11]

Počet vláken je omezený a jsou drahá z hlediska použití zdrojů procesoru a systémové režie, což představuje problém. Proces spuštění, plánování a ukončování vlákna vyžaduje hodně práce na pozadí. Počet jader procesoru určuje skutečný počet vláken, která mohou být spuštěna paralelně v kteroukoli chvíli. Pokud je potřeba více vláken než jader procesoru, musí systém použít plánovaní vláken, aby určil, jak rozdělit provádění vláken mezi dostupná jádra. Namísto vytváření nového vlákna pro každou coroutine a následného ukončení, když coroutine skončí, Kotlin udržuje fond aktivních vláken a řídí, jak jsou coroutines k těmto vláknům přiřazovány, aby se tato režie minimalizovala. Runtime Kotlin uloží aktivní coroutine, když je pozastavena, a začne ji nahrazovat jiná. Coroutine se jednoduše obnoví do již existujícího prázdného vlákna v rámci fondu, aby pokračovala v běhu, dokud buď neskončí, nebo nebude pozastavena, když bude obnovena. Tato metoda umožnuje efektivní využitý malého počtu vláken k provádění velkého počtu souběžných asynchronních úloh bez přirozeného snížení výkonu, které by bylo

důsledkem použitý běžného více vláknového zpracování. [1, s. 261] [1, s. 262] Běžnější metodou pro zpracování asynchronních operací jsou vlákna, zatímco coroutines nabízejí aktuálnější, organizovanější a moderní metodu. Mezi výhody coroutine patří: lepší zpracování výjimek, nenáročné provádění a strukturovaná souběžnost. [11] Jednou ze základních podmínek pro funkce coroutine je, že je třeba je spouštět v rámci „coroutine scope“.

3.4.2 Coroutine Scope

„*Pro coroutines, které jsou vytvořeny a spuštěny z CoroutineScope, je definován životní cyklus a životnost.*“ [12] Existují tři hlavní typy, které lze shrnout takto:

1. **GlobalScope.**

Jednou z metod pro zahájení coroutine je pomocí „Global Scope“. Když jsou spuštěny v rámci globálního rozsahu, coroutines pokračují tak dlouho, dokud program běží. Pokud ale coroutine dokončí svůj úkol, bude zničena a nebude existovat, i když aplikace bude běžet. Pokud ale coroutine stále provádí nějakou práci, tak coroutine zanikne, když je aplikace náhlé ukončena, protože její maximální životnost je stejná jako živostnost aplikace. [13]

2. **ViewModelScope.**

Při použitý komponenty „ViewModel“ Jetpack architektury je poskytována výhradně pro použití v instancích „ViewModel“. Běhový systém Kotlin automaticky zruší všechny coroutines spuštěné v tomto rozsahu, když je tato instance zničena. [1, s. 262] Třída s názvem „ViewModel“ se používá ke správě a ukládaní dat souvisejících s uživatelským rozhraní. [13]

3. **LifecycleScope**

Tenhle rozsah je skoro shodný s GlobalScope. Jediný rozdíl je v tom, že když se použije LifecycleScope, všechny coroutines spuštěné v rámci aktivity také zaniknout. [13] (Aktivitu lze tady definovat jako jednu obrazovku uživatelského rozhraní.)

Je důležité porozumět těmto pojmům, protože jsou základem pro zpracování asynchronních akcí a zachování integrity aplikace. Poskytují nezbytnou strukturu pro správu souběžných úkolů, zjištění, efektivity zdrojů a dodržování osvědčených postupů při vývoji aplikací pro Android. Později v této práci, po ponoření do frameworku Jetpack Compose, se zmíní funkce „LaunchedEffect“ které s těmito rozsahy souvisí.

4 Jetpack Compose

„*Jetpack Compose je doporučená moderní sada nástrojů pro Android pro vytváření nativního uživatelského rozhraní. Zjednodušuje a urychluje vývoj uživatelského rozhraní na Androidu. Rychle oživte svou aplikaci s menším množstvím kódu, výkonnými nástroji a intuitivními rozhraními Kotlin API.*“ [14] Toto je oficiální definice od Google, která je hlavním iniciátorem vytvoření tohoto frameworku. Vydání první verze 1.0 tohoto frameworku se zaznamenalo v červenci 2021. [15] Byl to velký krok vpřed ve vytváření současných uživatelských rozhraní aplikací pro Android, když byl tento framework zpřístupněn jako stabilní verze. Od svého uvedení na trh si Jetpack Compose mezi vývojáři Androidu získává stále větší oblibu a neustále se vylepšuje prostřednictvím nových funkcí a aktualizací. V úvodu bakalářské práce je zmíněn jeho deklarativní přístup, ale co to vlastně znamená? V oblasti vývoje aplikací je standardní postup vytváření uživatelských rozhraní založen na imperativní metodice. Nástup technologií jako Jetpack Compose však přinesl posun ve vývoji uživatelského rozhraní směrem k deklarativnímu přístupu.

4.1 Imperativní přistup.

Imperativní programování je programovací paradigma, ve kterém se poskytuje počítači explicitní instrukce, jak provést úkol. [16] Protože tyto pokyny krok za krokem specifikujeme abychom dosáhli konkrétních cílů, může být obtížné řídit imperativní programování ve velkých a komplikovaných aplikacích, přestože nabízí přesné ovládání. Zde je příklad definice tlačítka, kde každá z jeho vlastností je explicitně stanovena:

```
val tlacitko = Button(this)
tlacitko.text = "Klikni na mě"
tlacitko.setOnClickListener {
    // Akce při stisknutí tlačítka
}
```

4.2 Deklarativní přístup

Deklarativní programování na druhou stranu znamená definování výsledku, který chceme, aby náš kód produkoval. Toto je většinou dosaženo pomocí jedinečných funkcí a nástrojů, které jsou k dispozici prostřednictvím různých frameworků (jako například Jetpack Compose), programovacích jazyků a knihoven. [16] Zde je stejný příklad, který ale používá deklarativní přistup frameworku Jetpack Compose:

```
Button(onClick = {
    // Akce při stisknutí tlačítka
} ) {
    Text(text = "Klikni na mě")
}
```

Je důležité poznamenat, že to musí být voláno z funkce Composable, která je základní součástí frameworku Jetpack Compose. Tyto funkce budou podrobněji vysvětlena v další kapitole.

4.3 Composable funkce

Funkce Kotlin, ke které je připojena anotace @Composable, se nazývá Composable funkce. Tato anotace je vyžadována pro všechny Composable funkce, protože sděluje kompilátoru Compose, že funkce transformuje data na prvky uživatelského rozhraní. [2, s. 45] Obvykle, když je volána Composable funkce, tak obdrží některé vlastnosti a data, která specifikují, jak by měla související část uživatelského rozhraní vypadat ve spuštěné aplikaci. [1, s. 143] V této ukázce je definice jednoduché Composable funkce která nebude nic obsahovat. Je důležité poznamenat, že každá Composable funkce by měla začínat velkým písmenem, na rozdíl od Kotlin funkce, která začíná malým písmenem.

```
@Composable
fun PrikladFunkce() {

}
```

4.4 Stav v Jetpack Compose a rekompozice

Composable funkce jsou kategorizovány jako stavové nebo bezstavové, ale co ten stav vlastně reprezentuje? Stav v kontextu Compose je definovaná jako jakákoli hodnota, která se může změnit během spouštění aplikace. [1, s. 143] Jeden z hlavních důvodů pro implementace stavu je protože jakmile aplikace nakreslí komponentu Composable, nelze ji v Compose změnit, což znamená že pokud se změní hodnota tak se ta změna neprojeví na obrazovce zařízení. Nicméně lze změnit stavy, které jsou přiřazeny každé Composable funkce, aby se změnili hodnoty, které jsou jim poskytovány, a tak se změny projeví na obrazovce zařízení. [3, s. 59] Pokud programátor chce aby Compose vědělo o změnách stavu, musí být hodnota zabalena do objektu „State" pomoci funkce „mutableStateOf()" a také do funkce „remember", která se používá pro to aby se tato stavová proměnná uchovala mezi různými cykly rekompozice. Díky tomu Compose bude sledovat změny a aktualizovat uživatelské rozhraní. [17] Předtím než proběhne ukázka jak vypadá funkce která obsahuje stav proměnné, je potřeba si nejdřív definovat co taková rekompozice znamená.

Pomocí Compose jsou aplikace navrhovány uspořádáním Composable funkcí do hierarchií. Informace přenášené mezi této funkce jsou obvykle specifikovány jako stavové proměnné v nadřazené funkci. To znamená že jakákoliv úprava hodnoty stavu nadřazené položky se musí projevit také ve všech podřízených kompostovatelných položkách na které byl stav aplikován. Compose toto řeší provedením operace rekompozice. [1, s. 149]

Rekompozice je pouze opětovné volání metody a předání aktualizované hodnoty stavu. Pokaždé když se hodnota stavu v rámci hierarchie Composable funkcí změní, dojde k rekompozici, což znamená že se funkce znovu zavolá. Samozřejmě bylo by extrémně nehospodárné vykreslovat a aktualizovat uživatelské rozhraní pokaždé když se změní hodnota stavu, proto Compose používá inteligentní techniku rekompozice, aby se zabránilo této režii. Řečeno jinak, když se hodnota změní, tak rekompozice nastane pouze v těch funkcí, ve kterých se ta hodnota změnila. [1, s. 149] Zde je přiklad jak definovat proměnu s názvem „textPriklad" se měnitelným stavem:

```
var textPriklad = remember { mutableStateOf("") }
```

Popis toho, co tento řádek kódu provádí:

- „mutableStateOf“ je používaná k vytvoření stavové proměnné a sleduje její aktuální hodnotu a změny. Počáteční hodnota proměnné je prázdný řetězec (“”).
- Účelem funkce „remember“ je, aby zaručila, že hodnota bude zapamatována během cyklu rekompozice (opětovné volání funkce, kdykoli se hodnota změní). Rekompozice uživatelského rozhraní se může stát vícekrát během používání aplikace, ale funkce „remember“ zajišťuje, že stavová proměnná není vytvořena znovu od základu.

Pomoci kombinací těchto dvou funkcí je možné vytvořit stavovou proměnu, kterou je možné nazvat jakkoliv (jako třeba v příkladu „textPriklad“). Změny této hodnoty automaticky spustí proces rekompozice, který upraví prvky uživatelského rozhraní závislé na této proměnné. To umožnuje přizpůsobit uživatelské rozhraní aktuální hodnotě této proměnné.

S definicí základní struktury funkce Composable, stavu proměnné a principu fungování rekompozice, je čas vydat se hlouběji do světa Jetpack Compose a ukázat tak předpřipravené komponenty uživatelského rozhraní tohoto frameworku. Pomocí těchto předpřipravených komponent lze zjednodušit vytváření uživatelského rozhraní a zvýšit efektivitu a zábavu procesu vytváření aplikací.

4.5 Předdefinované komponenty Jetpack Compose

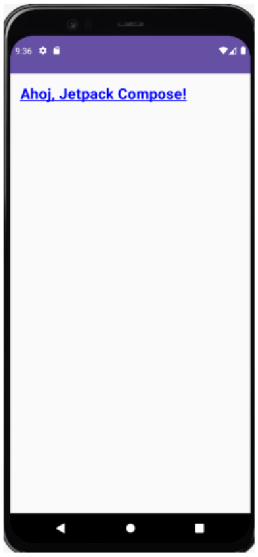
4.5.1 Text

V dynamickém prostředí Jetpack Compose je Text Composable funkce velmi důležitá a často používaná pro vytváření textu pro aplikace Android. Jedná se o základní stavební prvek, který umožňuje vývojářům efektivně a vysoce flexibilně vytvářet, stylovat a zobrazovat text. Tato funkce poskytuje možnost manipulovat s atributy, jako je font, velikost, barva, zarovnání a další.

```
@Composable
fun Text(
    text: String,
    modifier: Modifier = Modifier,
    color: Color = Color.Unspecified,
    fontSize: TextUnit = TextUnit.Unspecified,
    fontStyle: FontStyle? = null,
    fontWeight: FontWeight? = null,
    fontFamily: FontFamily? = null,
    letterSpacing: TextUnit = TextUnit.Unspecified,
    textDecoration: TextDecoration? = null,
    textAlign: TextAlign? = null,
    lineHeight: TextUnit = TextUnit.Unspecified,
    overflow: TextOverflow = TextOverflow.Clip,
    softWrap: Boolean = true,
    maxLines: Int = Int.MAX_VALUE,
    onTextLayout: (TextLayoutResult) -> Unit = {},
    style: TextStyle = LocalTextStyle.current
)
```

Jak je vidět, tato funkce nabízí řadu atributů pro přizpůsobený. Další příklad se však zaměřuje na použití těch nejpodstatnějších.

```
@Composable
fun TextComposablePriklad() {
    Text(
      text = "Ahoj, Jetpack Compose!", // Skutečný text, který bude
zobrazen.
      modifier = Modifier.padding(16.dp), // Modifikace rozložení a
stylu aplikované na text.
      color = Color.Blue, // Nastavuje barvu textu na modrou.
      fontSize = 34.sp, // Určuje velikost písma textu.
      fontWeight = FontWeight.Bold, // Určuje tučnost textu.
      textDecoration = TextDecoration.Underline // Přidává podtržení
k textu.
       )}
```



Obrázek 2 Výsledek přikladu „TextComposablePriklad“
Zdroj: Autor

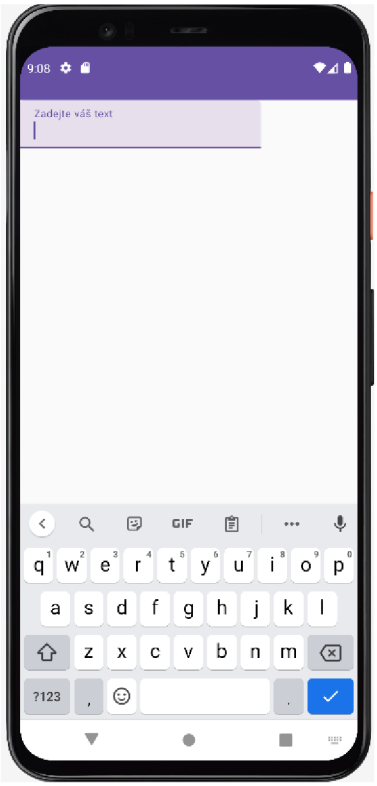
4.5.2 TextField

„TextField“ je klíčovou komponentou pro shromažďování textového vstupu od uživatelů snadno použitelným a přizpůsobitelným způsobem. Vývojáři mohou snadno navrhovat interaktivní pole pro zadávání textu pomocí této funkce.

```
@Composable
fun TextField(
    value: String,
    onValueChange: (String) -> Unit,
    modifier: Modifier = Modifier,
    enabled: Boolean = true,
    readOnly: Boolean = false,
    textStyle: TextStyle = LocalTextStyle.current,
    label: @Composable (() -> Unit)? = null,
    placeholder: @Composable (() -> Unit)? = null,
    leadingIcon: @Composable (() -> Unit)? = null,
    trailingIcon: @Composable (() -> Unit)? = null,
    isError: Boolean = false,
    visualTransformation: VisualTransformation =
VisualTransformation.None,
    keyboardOptions: KeyboardOptions = KeyboardOptions.Default,
    keyboardActions: KeyboardActions = KeyboardActions.Default,
    singleLine: Boolean = false,
    maxLines: Int = Int.MAX_VALUE,
    interactionSource: MutableInteractionSource = remember {
MutableInteractionSource() },
    shape: Shape = FilledTextFieldTokens.ContainerShape.toShape(),
    colors: TextFieldColors = TextFieldDefaults.textFieldColors()
)
```

Zde je příklad použití některých vlastností:

```
@Composable
fun TextFieldPriklad() {
    // Definice proměné s měnitelným stavem pro uložení textu
zadaného uživatelem.
    var text by remember { mutableStateOf("") }
    TextField(
// Atribut 'value' váže hodnotu textového vstupu na proměnnou'text'.
        value = text,
        // 'onValueChange' je zpětné volání, které aktualizuje
'text', když se text v TextFieldu změní.
        onValueChange = { zmenaTextu ->
            text = zmenaTextu
        },
        // 'label' je popisek zobrazený nad TextFieldem, který
naznačuje očekávaný vstup.
        label = { Text("Zadejte váš text") },
        // 'singleLine' je atribut určující, zda má být TextField
jednořádkový (true) nebo víceřádkový (false).
        singleLine = true // Nastavit na 'true' pro jednořádkový
TextField.
    ) }
```



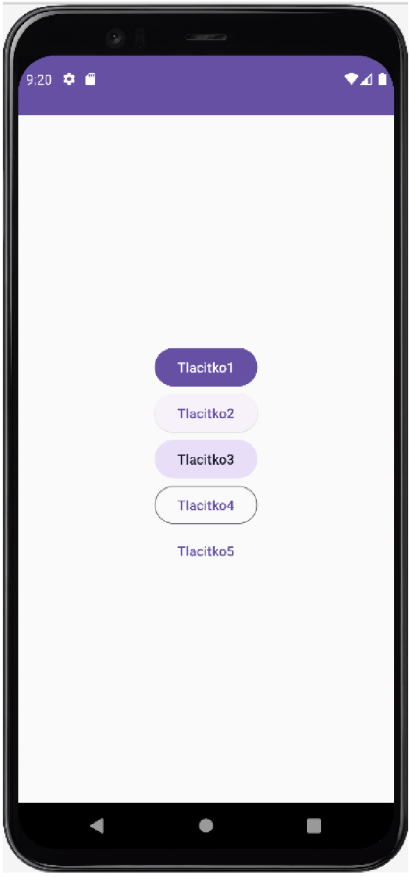
Obrázek 3 Výsledek příkladu „TextFieldPriklad“
Zdroj: Autor

Aby komponenta „TextField“ fungovala správně, je potřeba zadat hodnotu, která se během rekompozice nemění (proces který je vysvětlen v kapitole 3.5.4). Pomoci „mutableStateOf()“ funkce, se proměnná zabalí (v tomto příkladu je zabalena proměnná „text“). Tento postup umožní uložit a zobrazit text ve vstupním poli a zároveň nastaví výchozí hodnotu proměnné text na prázdný řetězec. Navíc je hodnota zabalena do funkce „remember“, což je metoda Compose, která definuje že hodnota by měla být zachována při rekompozici, protože pokud se tahle funkce nepoužije tak hodnota bude ztracena a znovu nastavena na výchozí hodnotu (“”) pokaždé když ta funkce bude volána. Vnitřní stav zařízení se nyní mění pokaždé, když uživatel stiskne klávesu na klávesnici. Výsledkem bude rekompozice a překreslení komponenty „TextField“ novým textem. To vše se stane opravdu rychle a uživatel nebude schopen poznat rozdíl. [4, s. 67] Toto se provádí pomoci důležité lambda funkce „onValueChange“ s parametrem typu „String“, která je volána, pokaždé když uživatel provede změnu v textovém poli. Jejím účelem je aktualizovat hodnotu textu v proměnné „text“ na základě uživatelovy interakce s komponentou „TextField“.

4.5.3 Button

Základní součástí prakticky každého programu, je tlačítko. Současná sada nástrojů učitelského rozhraní Android, Jetpack Compose, nabízí široký výběr předem definovaných tlačítek. Pomoci těchto tlačítek je snadné vytvořit dynamické a uživatelsky přívětivé rozhraní pro aplikaci. Zde je ukázka některých typů tlačítek, který Jetpack Compose nabízí:

```
@Composable
fun Tlacitka() {
        Button(onClick = { /*TODO*/ }) {Text("Tlacitko1")}
        ElevatedButton(onClick = { /*TODO*/ }) {Text("Tlacitko2")}
        FilledTonalButton(onClick = { /*TODO*/ }){Text("Tlacitko3")}
        OutlinedButton(onClick = { /*TODO*/ }) {Text("Tlacitko4")}
        TextButton(onClick = { /*TODO*/ }) {Text("Tlacitko5")}
}
```



Obrázek 4 Výsledek tlačítek
Zdroj: Autor

- **Button**: Pro základní akce, které musí být viditelné a snadno dostupné v rozhraní se používá typické tlačítko. Je vhodné omezit jejich nadměrné používaní a využívat pouze na jednu hlavní věc, která obrazovka dělá.

- **ElevatedButton**: Toto tlačítko se používá k upoutání pozornosti na hlavní akci v dané aplikaci. Má výraznější pozadí a vytváří pocit důležitosti.
- **FilledTonalButton**: Používá se pro doplňkové akce, která vyžadují odlišní vizuální styl než standartní tlačítko. Tento typ tlačítek může být nadměrně používán a mít tak více tlačítek tohoto typu na obrazovce, jako na rozdíl od normálního typu tlačítka, které je doporučeno používat jenom pro jednu hlavní věc na obrazovce.
- **OutlinedButton**: Pokud se má upozornit na akci bez plného pozadí, je vhodné použít tento typ tlačítka. Je vhodný pro méně důležité úkoly a vytváří tenký okraj kolem tlačítka.
- **TextButton**: Jednoduchý text, který slouží jako tlačítko pro úkoly, které nejsou příliš důležité. Používá se pro odkazy nebo méně významné interakce.

Jak lze konstatovat, existuje důvod pro více předdefinovaných tlačítek, protože některá z nich jsou důležitější než jiná. Při výběru typu tlačítka je třeba se zamyslet nad významem akce a nad tím, jak by se tlačítko mělo odlišovat od ostatních prvků obrazovky, a vybrat tak typ, který nejlépe vyhovuje požadavkům a preferencím aplikace. Je důležité poznamenat že pro každé tlačítko je třeba implementovat funkci „onClick“ která určuje, co se stane, když uživatel na to tlačítko klikne.

I přestože může původně připadat, že důležitost výběru tlačítka správného typu není relevantní, opak je pravdou. Protože dobrý vývojář UI aplikací, věnuje pozornost malým detailům, aby dosáhl vizuálně přitažlivou aplikaci.

4.6 Umístění prvků uživatelského rozhraní

Je zásadní pochopit, jak jsou tyto prvky uživatelského rozhraní uspořádány při jejich implementaci. Při prvním setkání s Jetpack Compose a jeho prvky se vývojář může domnívat, že vyvolání dvou funkcí za sebou povede k jejich vertikálnímu skládání. Ale není to realita. Je nutné uzavřít funkce do „Row“ a „Column“, aby bylo možné spravovat jejich umístění, s uvedením, zda mají být prvky uživatelského rozhraní organizovány svisle nebo vodorovně. Pokud tak není učiněno, může dojít k překrytí, jak ukazuje následující příklad.

```
@Composable
fun SkladaniTextu() {
    Text("Daniel")
    Text("Pitropovski")
}
```

```
//výsledek
DanielPitropovski
```

V Jetpack Compose jsou řádky a sloupce dva základní kontejnery rozložení. Kontejnery, které zarovnávají prvky vodorovně, se nazývají „Row“, zatímco kontejnery, které zarovnávají prvky svisle, se nazývají „Column“. Je možné navrhnout složitá uživatelská rozhraní s několika vrstvami hierarchie vnořením řádků a sloupců do sebe. [21] Zde jsou dva stejné příklady které toto využívají:

```
@Composable
fun RozlozeniTextuPomociColumn () {
    Column() {
        Text("Daniel")
        Text("Pitropovski")
    }
}
```

```
//výsledek
Daniel
Pitropovski
```

```
@Composable
fun RozlozeniTextuPomociRow() {
    Row() {
        Text("Daniel")
        Text("Pitropovski")
    }
}
```

```
//výsledek
DanielPitropovski
```

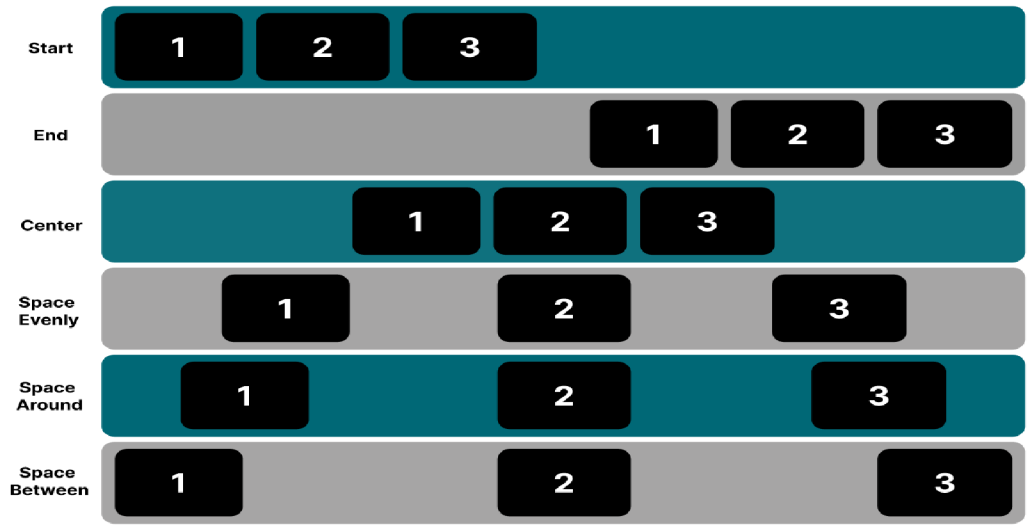
Funkce „Column“ a „Row“ slouží primárně k zarovnání prvků uživatelského rozhraní shora dolů pro sloupec nebo zleva doprava pro řádek. Nabízejí však flexibilní řadu možností pro úpravu vnitřní organizace a zarovnaní jejich podřízených prvků. Tyto funkce zahrnují řadu modifikátorů, které umožňují vývojářům pečlivě přizpůsobit, jak jsou objekty uspořádány ve sloupci nebo v řádku.

4.6.1 Řádek (Row)

Jak již bylo zmíněno, řádky v Jetpack Compose fungují jako kontejnery pro horizontální uspořádaní komponent. Tyto skládací prvky jsou uspořádány vodorovně podle hlavní osy v řadách. Umístění položek podél řádku je určeno touto osou. Při práci s hlavní osou v řadách jsou dva nejdůležitější faktory, které je třeba vzít v úvahu, a to je uspořádání a vyrovnání (arrangement and alignment).
Mezi běžné možnosti uspořádaní patří:

- **Start:** Komponenty jsou uspořádány od začátku řádku.
- **End:** Komponenty jsou uspořádány na konce řady.

- **Center**: Komponenty jsou v řadě vodorovně vycentrovány.
- **Space Evenly**: Rozděluje prostor mezi komponenty rovnoměrně.
- **Space Around:** Kolem každé komponenty je přidán stejný prostor. Mezera před první a za poslední komponentou však může být poloviční než mezera mezi položky.
- **Space Between**: Komponenty jsou rovnoměrně rozmístěny podél řady se stejnou mezerou mezi nimi.

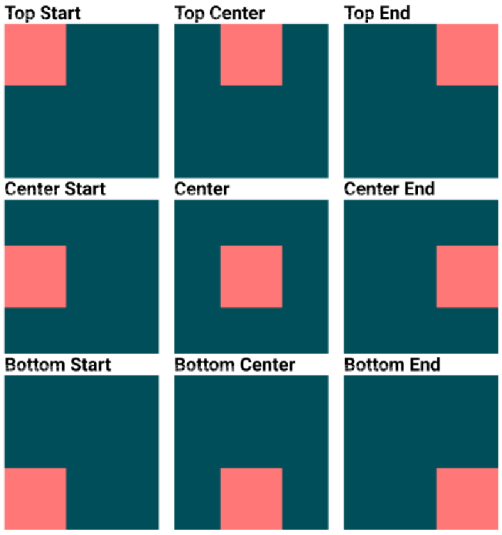


Obrázek 5 Možnosti uspořádání řádku v Jetpack Compose
Zdroj: Vlastní zpracování dle (4)

Mezi běžné možnosti vyrovnání patří:

- **Top Start**: Zarovná komponenty k horní části a začátku.
- **Top Center**: Zarovná komponenty k horní části a vodorovně na střed v kontejneru.
- **Top End**: Zarovná komponenty k horní časti a ke konci kontejneru.
- **Center Start**: Svisle vycentruje komponenty a vodorovně je zarovná k začátku kontejneru.
- **Center**: Umístí komponenty do svislých a vodorovných středů kontejneru.
- **Center End**: Svisle vycentruje komponenty a vodorovně je zarovná ke konci kontejneru.
- **Bottom Start**: Zarovná komponenty ke spodní části a začátku kontejneru.
- **Bottom Center**: Zarovná komponenty vodorovně dolů a na střed kontejneru.

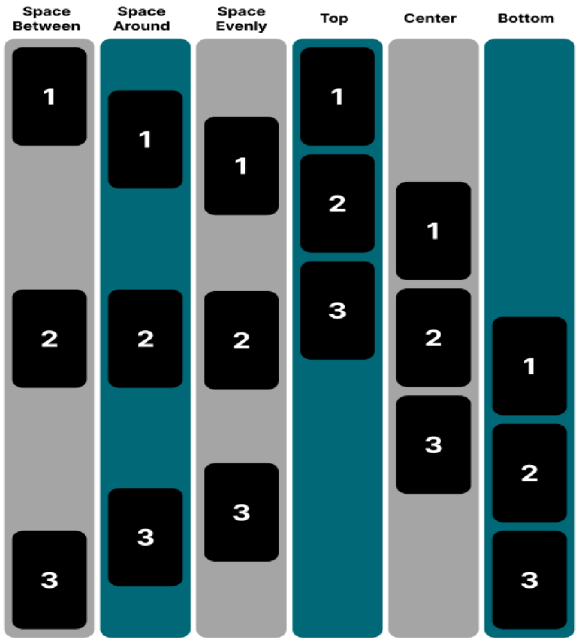
- **Bottom End**: Zarovná komponenty ke spodní části a ke konci kontejneru.



Obrázek 6 Možnosti vyrovnání řádku v Jetpack Compose
Zdroj: Autor

4.6.2 Sloupec (Column)

Schopnost sloupu uspořádat věci vertikálně je jednou z jeho primárních charakteristik. Stejně jak jeho řádkový ekvivalent „arrangement" určuje zarovnání a vzdálenost mezi jednotlivými komponenty. Vlastnosti uspořádaní, které lze použít pro sloupec, jsou podobné vlastnostem řádku, ale jsou aplikovány vertikálně.

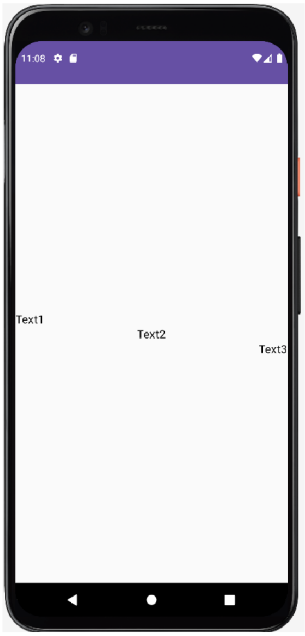


Obrázek 7 Možnosti uspořádání řádku v Jetpack Compose

Zdroj: Vlastní zpracování dle (4)

Zde je praktický příklad použitý řádků a sloupců v Jetpack Compose:

```
//  Komponovatelná funkce jménem "PrikladZarovnaniKomponent", která
demonstruje použití komponovatelných prvků Column a Row v Jetpack
Compose.
//  Vytváří vertikální uspořádání pomocí Column a v něm tři řádky s
různým uspořádáním.
//
// *Celkový layout je nastaven tak, aby byl vertikálně i
horizontálně zarovnán uprostřed dostupného prostoru.
//
// - První řádek ("Text1") je zarovnán vlevo (Arrangement.Start) v
celé šířce sloupce.
// - Druhý řádek ("Text2") je zarovnán do středu vertikálně i
horizontálně v celé šířce sloupce.
// - Třetí řádek ("Text3") je zarovnán vpravo (Arrangement.End) a
dole v celé šířce sloupce.
@Composable
fun PrikladZarovnaniKomponent() {
    Column(
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally,
        modifier = Modifier.fillMaxSize()
    ) {
        // První řádek
        Row(
            modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.Start
        ) {
            Text(text = "Text1")
        }
        // Druhý řádek
        Row(
            modifier = Modifier.fillMaxWidth(),
            verticalAlignment = Alignment.CenterVertically,
            horizontalArrangement = Arrangement.Center
        ) {
            Text(text = "Text2")
        }
        // Třetí řádek
        Row(
            modifier = Modifier.fillMaxWidth(),
            verticalAlignment = Alignment.Bottom,
            horizontalArrangement = Arrangement.End
        ) {
            Text(text = "Text3")
        }
    }
}
```



Obrázek 8 Výsledek příkladu „PrikladZarovnaniKomponent“
Zdroj: Autor

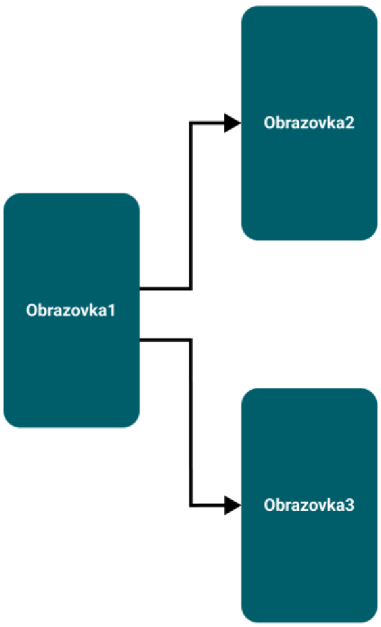
4.7 Navigace v Jetpack Compose

Paradigma pro vývoj komplexních rozhraní se v dynamickém světě aplikací a technologií změnilo. V dnešní době je neobvyklé setkat se s aplikací pouze s jednou obrazovkou; místo toho je běžnější vidět několik složitě propojených obrazovek spojených dohromady. Kromě vzájemné komunikace vyžadují tyto obrazovky chytrý a uživatelsky přívětivý navigační mechanismus.

Používaní navigačních knihoven je běžným přístupem v oblasti vývoje aplikací pro Android. Tyto účinné nástroje jsou nezbytné pro zjednodušení procesu navigace, protože poskytují vývojářům silný rámec pro koordinaci hladkého přechodu mezi různými obrazovkami.

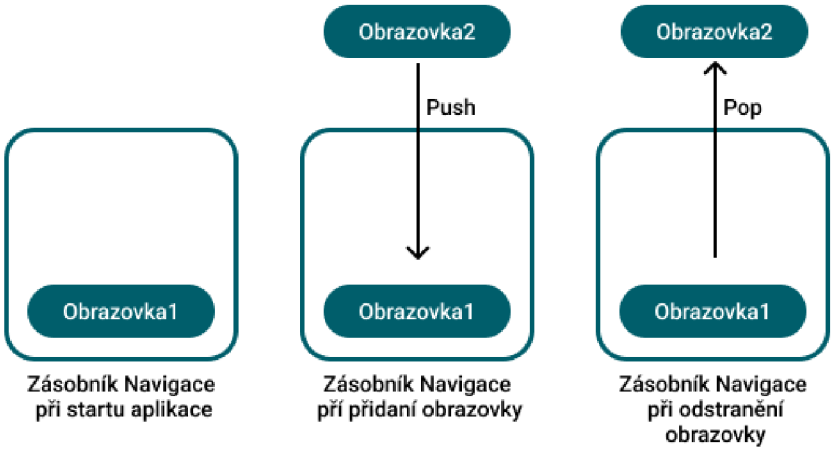
V současném prostředí aplikací nelze dostatečně zdůraznit důležitost efektivní navigace. Slouží jako lepidlo, které spojuje mnoho obrazovek do bezproblémového uživatelského zážitku. Kromě usnadnění vývoje zlepšují navigační knihovny také celkovou produktivitu a spokojenost uživatelů. Jak se technologie dále rozvijí, stává se začlenění těchto navigačních knihoven nejen osvědčeným postupem, ale také klíčovým nástrojem, který mohou vývojáři Androidu mít ve své sadě nástrojů. To pomáhá zajistit, že aplikace nejen splňují, ale také překonávají vysoké standardy dnešních náročných uživatelů.

V následujícím příkladu budou definováni tři obrazovky, které budou propojeny takto:



Obrázek 9 Navigace mezi obrazovky
Zdroj: Autor

Navigační architektura sleduje pochyb uživatele pomocí navigačního zásobníku. V tomto příkladu jako první obrazovka, která bude na začátku zásobníku je „Obrazovka1“, to znamená že při prvním spuštění aplikace se zobrazí právě tato. Jak se uživatel naviguje mezi obrazovkami, tyto obrazovky se přidávají do zásobníku anebo se odebírají.



Obrázek 10 Zásobník navigace
Zdroj: Vlastní zpracování dle [1, s. 396]

Použití této architektury k přidaní navigace do aplikace pro Android je jednoduchý proces, který vyžaduje: „NavigationGraph“, „NavigationHost“, navigační akce a velmi málo psaní kódu pro interakci s instancí navigačního ovladače. [1, s. 395] [1, s. 396]

Předtím než se vůbec uživatel může někam navigovat je potřeba nejdřív definovat „Navigation Graph“, neboli obrazovky, které bude aplikace obsahovat. Každá obrazovka, která tvoří aplikaci se označuje jako trasa (route) podle které se potom uživatel naviguje.

Pro definování jednotlivých obrazovek jako jedna z metod se používá „enum class“. Třída „enum“ v Kotlinu je jako speciální třída používaná k definování specifické, neměnné sady souvisejících hodnot. Tyto hodnoty jsou obvykle konstanty s názvy, což usnadňuje práci s nimi v kódu. Třídy „enum“ poskytují úhledný a bezpečný způsob, jak reprezentovat a zpracovávat předdefinovanou skupinu možností. [18]

Pokud jde o použití této třídy pro navigační trasy, tak nabízí několik výhod. Jedna z nich je že pomoci předem definovaných navigačních tras se sníží riziko překlepu nebo chyb za běhu. Díky tomu je kód robustnější a pomáhá zachytit potenciální problémy v době kompilace, což poskytuje typově bezpeční a jasný způsob, jak zvládnout navigaci v celé aplikaci. Jak je ale zmíněno je třeba nejdřív definovat trasy pro každou obrazovku:

```
enum class NavigationGraph {
    Obrazovka1,
    Obrazovka2,
    Obrazovka3,
}
```

Pokud vývojář bude chtít do projektu aplikace zahrnout navigaci, bude potřeba vytvořit instanci „NavHostController“. To má na starost udržovaní přehledu o tom, která obrazovka je aktuálním cílem a spravuje také zásobník. Pro vytvoření této instance je potřeba zavolat funkci „rememberNavController()“. [1, s. 397]

```
val navController = rememberNavController()
```

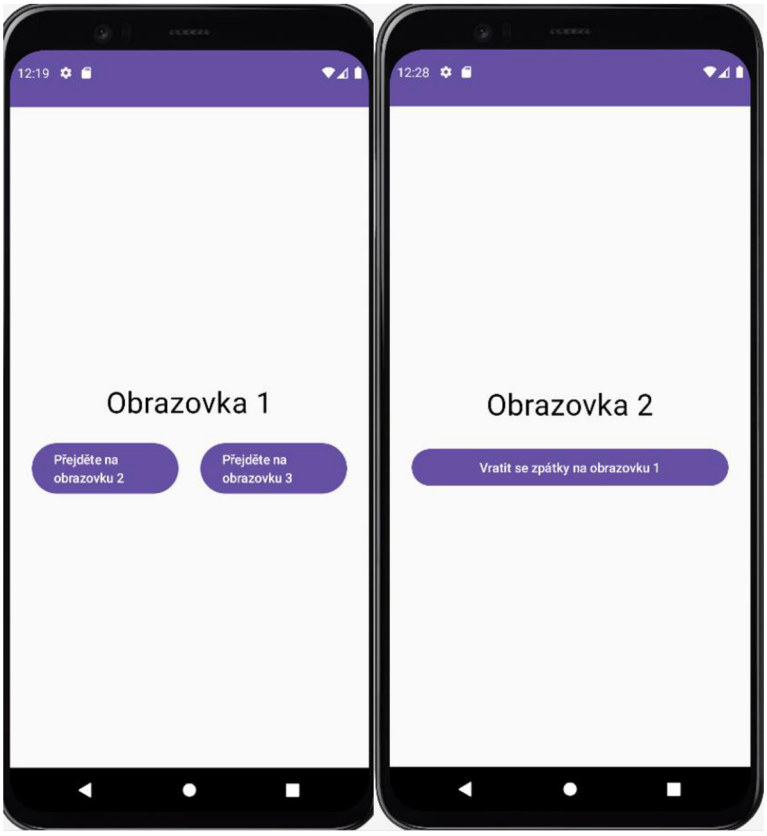
„NavHost“ je speciální komponenta kterou lze zahrnout do svého rozvržení. Zobrazuje různé cíle z navigačního grafu. „NavHost“ spojuje „NavController“ s navigačním grafem, což umožňuje samotnou navigaci.

```
@Composable
fun Navigation() {
    val navController = rememberNavController()
    NavHost(
        navController = navController,
        startDestination = NavigationGraph.Obrazovka1.name
    ){
        composable(route=NavigationGraph.Obrazovka1.name){
            Obrazovka1(navController=navController)
        }
        composable(route=NavigationGraph.Obrazovka2.name){
            Obrazovka2(navController=navController)
        }
        composable(route=NavigationGraph.Obrazovka3.name){
            Obrazovka3(navController=navController)
        }
    }
}
```

- **NavHost:** Komponenta aplikace, která spravuje navigaci. Tento příklad obsahuje objekt zvaným „navController“, který umožňuje spravovat navigaci v programu. Proto také jako parametr se tato instance „navController“ přiděluje k jednotlivým funkcí pro každou obrazovku. To nám umožní navigovat se z jedné obrazovky do druhé.
- **startDestination:** Označuje cíl, který se zobrazí ve výchozím nastavení. Protože je v tomto příkladě nastavena na „NavigationGraph.Obrazovka1.name“, se při spuštění aplikace objeví první obrazovka (Obrazovka1).
- **composable:** Identifikuje různé obrazovky (cíle) a přiděluje odpovídající funkce zápisu, které vykreslují obsah každé obrazovky. V tomto případě by to znamenalo že k destinaci „NavigationGraph.Obrazovka1.name“ je přidělena funkce „Obrazovka1()” která obsahuje už prvky uživatelského rozhraní. Ostatní obrazovky jsou definovány podobným způsobem.

Zde je obsah první obrazovky a její komponenty uživatelského rozhraní:

```
@Composable
fun Obrazovka1(navController: NavHostController) {
    Column(modifier= Modifier.fillMaxSize().padding(24.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center)
    {
        Text(fontSize = 32.sp, text = "Obrazovka 1")
        Spacer(modifier=Modifier.size(24.dp))
        Row(){
            //Tlačítko které naviguje na obrazovku 2
            Button(
                onClick = {
navController.navigate(NavigationGraph.Obrazovka2.name) },
                modifier=Modifier.weight(1f),) {
                Text(text = "Přejděte na obrazovku 2")
            }
            Spacer(modifier=Modifier.size(24.dp))
            //Tlačítko které naviguje na obrazovku 3
            Button(
                modifier=Modifier.weight(1f),
                onClick = {
navController.navigate(NavigationGraph.Obrazovka3.name) }) {
                Text(text = "Přejděte na obrazovku 3")}
        }
    }
}
```



Obrázek 11 Výsledek příkladu „Obrazovka1“
Zdroj: Autor

Důležitým prvkem, který je potřeba zaznamenat, je implementace funkce „onClick“ pro všechna tlačítka. Při stisku prvního tlačítka s nápisem „Přejděte na obrazovku 2“ je využita instance „navController“ pro navigaci. Tato instance obsahuje funkci „.navigate()”, která vyžaduje cíl, na který se má přejít. Cíle jsou už vlastně definovány v navigačním grafu (NavigationGraph). „Obrazovka2“ obsahuje jediné tlačítko, které v „onClick“ používá metodu „navigation.popBackStack()”. Tato metoda odstraní obrazovku, ve které se nachází, ze zásobníku a tím se vrátí na předchozí obrazovku (v tomto případě se vrátí na obrazovku 1). Pomocí těchto komponent se uživatel dokáže navigovat mezi jednotlivé obrazovky.

4.7.1 Předávání argumentů mezi obrazovky

Přechod z jedné obrazovky na druhou často vyžaduje odeslání konkrétních argumentů do cíle. Compose poskytuje kompletní podporu pro předávaní argumentů jakéhokoli druhu mezi obrazovkami pomocí sady definovaných kroků. Compose zjednodušuje proces a zaručuje bezproblémový tok informací mezi různými oblastmi použití, ať už jde o základní data nebo složitější položky.

První krok při navigaci s argumenty zahrnuje přidání názvu argumentu do cílové trasy. V příkladu bude předán argument s názvem „jmeno“ z „Obrazovka1“ do „Obrazovka2“.

```
composable(route=NavigationGraph.Obrazovka2.name + "/{jmeno}"){
// Pro získaní hodnoty toho argumentu je potřeba vykonat tento řádek
backStackEntry ->
val jmenoZObrazovky1 =backStackEntry.arguments?.getString("jmeno")
Obrazovka2(
        navController=navController,
        jmenoZObrazovky1=jmenoZObrazovky1
)}
```

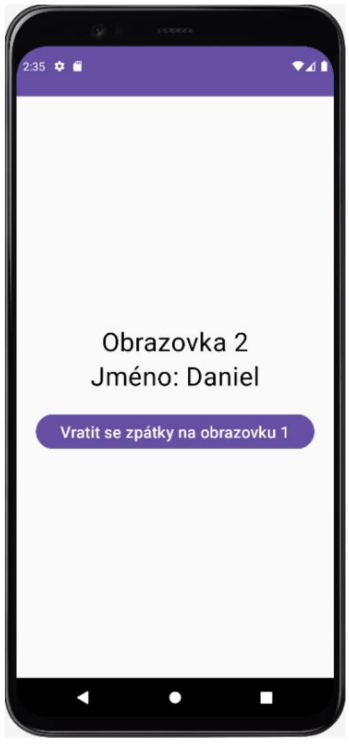
Na rozdíl od předchozích příkladu je zde specifikován argument, který má název „jmeno“. To znamená, že při přechodu na „Obrazovka2“ bude potřeba také poslat argument. Hodnota argumentu je uložena do proměnné „jmenoZObrazovky1“, která se potom posílá do Composable funkce „Obrazovka2“ aby se mohla zobrazit.

Ukázka navigace na „Obrazovka2“ a předání argumentu:

```
Button(
    onClick = {
navController.navigate(NavigationGraph.Obrazovka2.name+"/Daniel") },
    modifier=Modifier.weight(1f),) {
    Text(text = "Přejděte na obrazovku 2")
}
```

Zde je uvedeno zobrazení hodnoty argumentu v „Obrazovka2“ pomoci proměnné „jmenoZObrazovky1“:

```
@Composable
fun Obrazovka2(navController:NavHostController,
               jmenoZObrazovky1:String?){
    Column(modifier= Modifier.fillMaxSize().padding(24.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center) {
        Text(fontSize = 32.sp, text = "Obrazovka 2")
        if (jmenoZObrazovky1 != null) {
        Text(fontSize = 32.sp, text = "Jméno: $jmenoZObrazovky1")
        }
        Spacer(modifier= Modifier.size(24.dp))
        Row(modifier = Modifier.fillMaxWidth()) {
            Button(
                onClick = { navController.popBackStack() },
                modifier= Modifier.weight(1f),) {
                Text(text = "Vratit se zpátky na obrazovku 1")
            }
        }
    }
}
```



Obrázek 12 Výsledek příkladu „Obrazovka2“
Zdroj: Autor

Praktická část

5 Vývoj aplikace Yamb

Tato praktická část bakalářské práce se bude zajímat o znovuvytvoření deskové hry Jamb (Yamb) pomoci programovacího jazyka Kotlin a frameworku Jetpack Compose. Cílem je vytvořit funkční a uživatelsky přívětivou digitální verzi Yamb s využitím objektové orientovaných programovacích schopností (Kotlin) a efektivních nástrojů pro vývoj uživatelského rozhraní (Jetpack Compose). Porozumění pravidlům hry, vytvoření snadno použitelného uživatelského rozhraní a obratné použití herních mechanismů (na příklad házení kostkami) budou prioritami celého projektu. Je důležité uvést, že tato aplikace nebude obsahovat úložiště dat. Místo toho se zaměří výhradně na využití frameworku Jetpack Compose k sestavení všech komponent nastíněných v teoretické části pro vývoj aplikací. Pozoruhodné je, že zatímco data budou prezentovaná během používaní aplikace, nebudou uložena. Je také klíčové zmínit, že příklady kódu nebudou obsahovat celý kód kvůli jeho velikosti, avšak kompletní projekt je k dispozici v příloze číslo 1.

5.1 Yamb

Yamb, někdy označována jako Jamb, je oblíbená kostková hra, která se poprvé hrála v Jugoslávii ve 20. století a od té doby se rozšířila po celém světě. S použitím pěti kostek a výsledkové listiny je to strategická a náhodná hra. Cílem hry Yamb, kterou obvykle hrají dva nebo více hráčů, je získávat body strategickým házením kombinací kostek. Pokud hráč chce získat co nejvíce bodů, musí házet kostkami a vytvářet různé kombinace, z nichž každá má hodnoty určitého počtu bodů. Yamb je hra v kostky, která začala jako rozšíření hry Yahtzee od Miltona Bradleyho, která poprvé vyšla v 50. letech minulého století. [20]

5.1.1 Pravidla hry Yamb

Ve hře je několik kol. Hráč může dokončit svůj tah po jednom nebo dvou hodech, ale vždy dostane tři hody kostkou v každém kole. Hráč si může po prvním hodu uložit libovolnou kostku, kterou si zvolí, a znovu hodit zbývající kostky. Po druhém házení se tento proces opakuje. Hráč si může svobodně vybrat, kterou kostkou vůbec hodí.

Další možností je přehození oběma dříve hozenými kostkami. Hráč si po třetím hodu každého kola musí vybrat kategorii na výsledkové kartě Yamb. V závislosti na tom, zda pět kostek splňuje pravidla hodnocení kategorie, se skóre zaznamenává do pole. Na druhou stranu, pokud hráč po třech hodech nedokáže získat vhodné skóre v dané kategorii má možnost zadat nulu do pole kategorie. To znamená, že v tomto kole nebudou moci získat žádné další body, protože jejich současný hod nesplňuje požadavky dané kategorie. Hru dokončí každý hráč, pokud každé políčko je vyplněno. Součet všech políček plus případné bonusy určuje konečné skóre. [19]

	▼	▲	▼▲	N	O
1 (Ones)					
2 (Twos)					
3 (Threes)					
4 (Fours)					
5 (Fives)					
6 (Sixes)					
Min					
Max					
Trilling +10					
Leader+35/40					
Full +30					
Kare +40					
Yamb +50					

Obrázek 13 Výsledková tabulka ve hře Yamb
Zdroj: Autor

Existují různé varianty Yamb, každá s odlišným počtem sloupců a možnými kombinacemi. Pro nejzákladnější verzi je k dispozici tabulka s pěti sloupci („Dolů“, „Nahoru“, „Volný“, „Hlášení“ a „Odhlášení“). [20]

- Sloupec Dolů (▼): Hráč musí postupně vyplnit tento sloupec od 1 po Yamb, přičemž musí vyplnit všechna pole.
- Sloupec Nahoru (▲): Hráč musí vyplnit tento sloupec tak, že začne od Yamb k 1 a také nesmí vynechat žádná pole.

- Sloupec Volný (▼▲): Pořadí, ve kterém hráči vyplní tento sloupec, je zcela na nich.
- Sloupec Hlášení (N): Tento sloupec je aktivován po prvním hodu. Po výběru kostek, které jsou často ty s nejvyšší hodnotou, hráč řekne „Oznamuji“ nebo „Hlásím“ a může hodí ještě dvakrát, aby dosáhl nejvyšší počet bodů pro tu vybranou kombinaci, kterou hráč oznámil. Případně mohou dokončit svůj tah zaznamenáním výsledku prvního hodu.
- Sloupec Odhlášení (O): Po ohlášení předchozího hráče se tento sloupec stane aktivním. Pokud například první hráč oznámí šestky, další hráč musí upřednostnit zápis šestek do příslušného pole a vyplnit tento sloupec šestkami.

Výsledková tabulka ve hře Yamb je rozdělena na horní a dolní část:

Kategorie	Skoré	Příklad
Esa (Ones)	Součet kostek s číslem 1	 Skoré: 3
Dvojky (Twos)	Součet kostek s číslem 2	 Skoré: 6
Trojky (Threes)	Součet kostek s číslem 3	 Skoré: 9
Čtyřky (Fours)	Součet kostek s číslem 4	 Skoré: 8
Pětky (Fives)	Součet kostek s číslem 5	 Skoré: 15
Šestky (Sixes)	Součet kostek s číslem 6	 Skoré: 12

Obrázek 14 Horní část výsledkové tabule ve hře Yamb
Zdroj: Vlastní zpracování dle [20]

V Yamb je v horní polovině výsledkové karty šest políček očíslovaných od 1 do 6. Hráč může obdržet bonus, pokud může celkově získat 60 bodů nebo více vyplněním všech šesti těchto políček. Bonus se liší podle celkového počtu bodů:

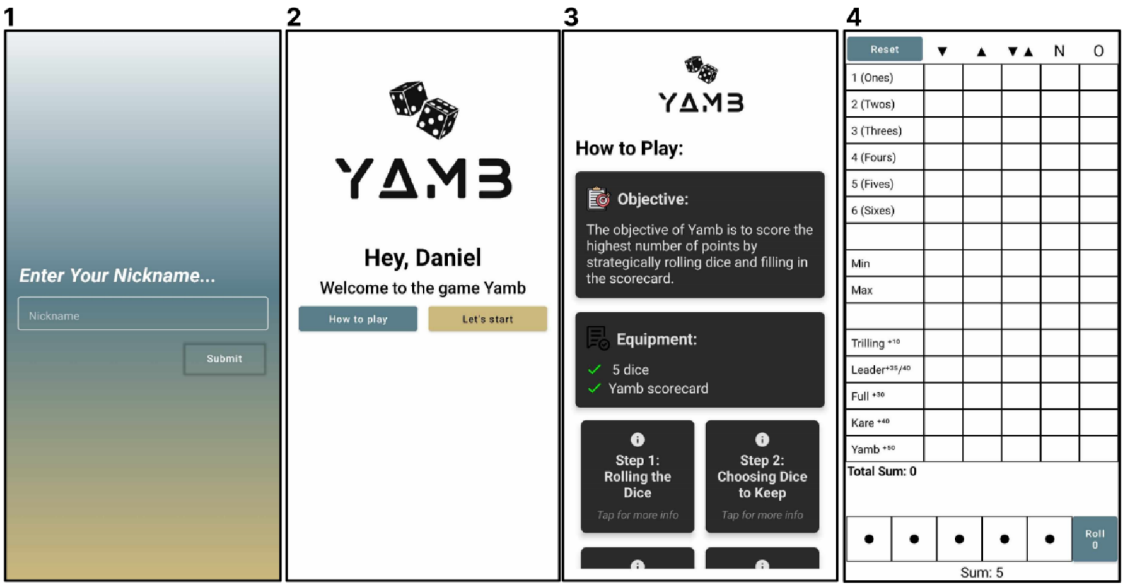
- Pokud hráč dosáhne 60 až 69 bodů, obdrží bonus 30 bodů.
- Pokud hráč dosáhne 70 až 79 bodů obdrží bonus 40 bodů.
- Pokud hráč dosáhne 80 nebo více bodů obdrží bonus 50 bodů.

Kategorie	Popis	Skoré	Příklad
Min	Hráč se snaží získat co nejnižší součet.	Součet všech kostek	Skoré: 8
Max	Hráč se snaží získat co nejvyšší součet.	Součet všech kostek	Skoré: 26
Trilling	Alespoň tři kostky stejné	Součet těch tří kostek +10	Skoré: 9+10=19
Leader	Pět po sobě jdoucích kostek (1-2-3-4-5 nebo 2-3-4-5-6)	Pokud (1-2-3-4-5) tak 35 Pokud (2-3-4-5-6) tak 40	Skoré: 35
Full	Tři z jednoho čísla a dva z druhého	Součet všech kostek +30	Skoré: 23+30=53
Kare	Alespoň čtyři kostky stejné	Součet těch čtiř kostek +40	Skoré: 20+40=60
Yamb	Všech pět kostek stejně	Součet všech kostek +50	Skoré: 25+50=75

Obrázek 15 Dolní část výsledkové tabule ve hře Yamb
Zdroj: Vlastní zpracování dle [20]

5.2 Implementace deskové hry Yamb pomoci programovacího jazyka Kotlin a frameworku Jetpack Compose

Následující aplikace bude obsahovat tyto čtyři obrazovky:



Obrázek 16 Obrazovky aplikace Yamb
Zdroj: Autor

Předtím než proběhne ukázka, jakým způsobem vytvořit tyto obrazovky, je potřeba nejdřív definovat obrazovky podobným způsobem jako v kapitole 3.8. Tato část bude potřeba nejen pro definici ale také umožní navigaci mezi obrazovky a případné posílání argumentů mezi nimi.

Obrazovka s číslem 1(viz. Obrázek 16) je ta první kterou uživatel uvidí, když otevře aplikaci. Jak je vidět, tak obsahuje gradientní pozadí, které lze dosáhnout takto:

```
Box(
    modifier = Modifier
        .fillMaxSize()
        .background(
            brush = Brush.verticalGradient(
                colors = listOf(
                    Color(0xFFFFFFFF), //bíla
                    Color(0xFF5A7D8A), //modrá
                    Color(0xFFCDB87D),)))//žlutá
    )
```

- **Box**: V Jetpack Compose je „Box“ základním prvkem rozvržení, který organizuje a prezentuje prvky uživatelského rozhraní v obdélníkové oblasti. Poskytuje vývojářům přizpůsobitelné plátno, na kterém lze uspořádat podřízené komponenty, což jim umožnuje vytvářet esteticky přitažlivá uživatelská rozhraní.
- **Modifier** (Modifikátor): Prvky uživatelského rozhraní Compose mohou mít různé style a transformace pomocí modifikátoru. V tomto případě se používá ke změně vlastností boxu.
- **fillMaxSize()**: Tento modifikátor zajišťuje, že „Box“ zabírá místa, které může poskytnout jeho nadřazené rozvržení. Výsledkem je, že „Box“ zabírá celý nadřazený kontejner nebo obrazovku.
- **background()**: Tato modifikace definuje změnu pozadí daného prvku (v tomto případě změna pozadí boxu).
- **brush=Brush.verticalGradient()**: Tento parametr uvnitř funkce „background()“ definuje vertikální přechod barev.
- **colors**: Do tohoto parametru se zadává seznam barev, které definují ten přechod. Přechodové pozadí bude vytvořeno smícháním barev dohromady.

Z obrázku vyplývá že komponenty jsou umístěny uprostřed obrazovky a uspořádány vpravo. Zde je kompletní kód obrazovky 1, která obsahuje předdefinované komponenty jako text (kapitola 3.6.1), textové pole (kapitola 3.6.2) a tlačítko (kapitola 3.6.3):

```
@Composable
fun NicknameEntryScreen(onNicknameEntered: (String) -> Unit) {
    var nickname by remember { mutableStateOf("") }
    var snackbarVisible by remember { mutableStateOf(false) }
    Box(
        modifier = Modifier
            .fillMaxSize()
            .background(
                brush = {...}))
    {
        Column(
            modifier = Modifier.fillMaxSize().padding(16.dp),
            verticalArrangement = Arrangement.Center,
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            Text(
                text = "Enter Your Nickname...",
                color = Color.White,
                fontSize = 28.sp,
                modifier = Modifier.fillMaxWidth()
            )
            OutlinedTextField(
                value = nickname,
                onValueChange = { nickname = it },
                label = { Text("Nickname") },
                modifier = Modifier.fillMaxWidth()
                    .padding(bottom = 16.dp).height(56.dp),
                colors = {...}
            )
            Row(
                modifier = Modifier.fillMaxWidth(),
                horizontalArrangement = Arrangement.End
            ) {
                Button(
                    onClick = {
                        if (nickname.isNotBlank()) {
                            onNicknameEntered(nickname)
                        }else {
                            snackbarVisible = true
                        }
                    },
                    modifier = Modifier
                        .height(48.dp).width(120.dp),
                    colors = {...}
                )
                { Text(text = "Submit", color=Color.White) }
            }
        }
    }
    MySnackbar(
        snackbarVisible = snackbarVisible,
        onDismissSnackBar = {snackbarVisible=false}
    )
}
```

Hlavní cílem této obrazovky je umožnit uživateli zadat přezdívku do textového pole a po kliknutí tlačítka tak přesměrovat na druhou uvítací obrazovku

(viz. Obrázek 16). Jak lze vidět z ukázky kódu, tak při stisknutí tlačítka se hodnota proměnné „nickname“ předává lambda funkci „onNicknameEntered“, která je nastavená jako parametr této Composable funkce. Hlavním cílem je odeslat hodnotu této proměnné na další obrazovku (kapitola 3.8.1).

```
composable(NavigationRoutes.LogInScreen.name) {
    NicknameEntryScreen { nickname ->
        val route =NavigationRoutes.GreetingScreen.name
        navController.navigate("$route/$nickname",)
    }
}
```

Další důležitou funkcí je „MySnackbar()“, která informuje uživatele, když se pokusí zadat prázdnou přezdívku. Cílem této funkce je informovat uživatele o problému zadávání prázdné přezdívky a tím tak zlepšuje uživatelskou zkušenost tím, že poskytuje jednoznačnou zpětnou vazbu.

```
@Composable
fun MySnackbar(snackbarVisible:Boolean,onDismissSnackBar:()->Unit) {
    if (snackbarVisible) {
        Snackbar(
            modifier = Modifier.padding(top = 65.dp),
            content = {
                Text(text = "Please enter a nickname",)
            },
            action = {
                Button(onClick = { onDismissSnackBar()}) {
                    Text("Dismiss")
                }
            }
        )
    }
}
```

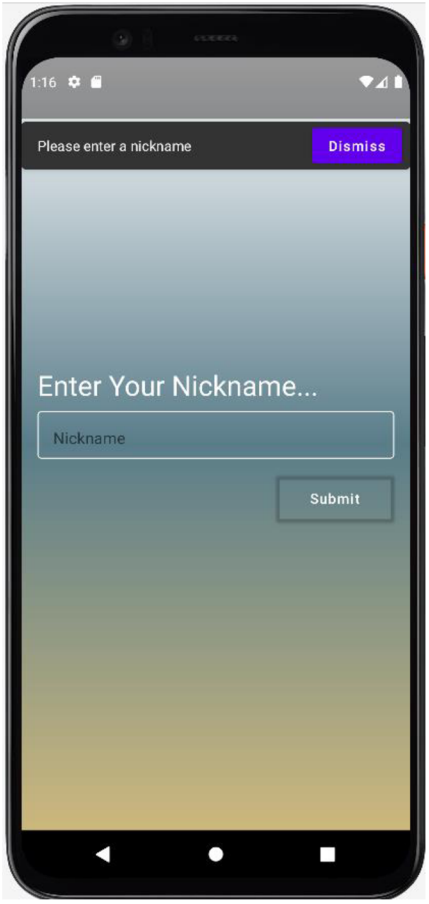
Parametry funkce:

- **snackbarVisible**: Viditelnost panelu „Snackbar“ je určena tímto parametrem. Protože jeho stav lze externě změnit, lze jej dynamicky zobrazovat v reakci na určité události, jako je pokus o odeslání prázdného uživatelského jména.
- **onDissmissSnackbar()**: Když uživatelé interagují s panelem „Snackbar“ (například kliknutím na tlačítko pro zrušení) tak tato funkce zpětného volání zruší „Snackbar“, když je vyvolán.

Pouze když je proměnná „snackbarVisible“ nastavena na „true“, objeví se „Snackbar“, která je jedna z předdefinovaných komponent Jetpack Compose. Toto

podmíněné vykreslování umožnuje kontextové zobrazení, které zaručuje, že se tato komponenta zobrazí v případě potřeby. Na panelu „Snackbar“ se zobrazí výrazné oznámení: „Please enter a nickname.“ Tato část upozorní uživatelé, aby zadali svou přezdívku.

Uživatelé mohou zavřít tento panel kliknutím tlačítka „Dismiss“, který nastaví hodnotu proměnné „snackbarVisible“ na „false“. Důležité je poznamenat že dokud uživatel nezadá hodnotu do textového pole tak se nemůže dále přesměrovat na jiné obrazovky.



Obrázek 17 Výsledek příkladu, kdy uživatel stiskne tlačítko, aniž by vyplnil textové pole
Zdroj: Autor

Pokud uživatel vyplnil textové pole a stiskl tlačítko, tak se nasměruje na druhou uvítací obrazovku (viz. Obrázek 16), ve které bude uvítán s jeho přezdívkou, která byla předána jako argument z první obrazovky. Hodnotu toho argumentu je potřeba ale dostat pomoci metody uvedené v kapitole 3.8.1. Poté bude tato hodnota poslána jako parameter při přesměrování na uvítací obrazovku.

```
val route=NavigationRoutes.GreetingScreen.name
composable("$route/{nickname}") { backStackEntry ->
    val nickname = backStackEntry.arguments?.getString("nickname")

    GreetingScreen(
       nickname = nickname ?: "",
       onHowToPlayClick ={navigateToHowToPlayScreen(navController)},
       onStartGameClick = { navigateToGameScreen(navController) }
    )
}
```

```
@Composable
fun GreetingScreen(nickname: String,onHowToPlayClick: () ->
Unit,onStartGameClick: () -> Unit) {
    Box(modifier = {...})
    {
        Column(
            horizontalAlignment = Alignment.CenterHorizontally,
            modifier = Modifier.fillMaxWidth()
        ) {
            Image(
                painterResource(id = R.drawable.yamb),
                contentDescription = "Background",
                modifier = {...},
                contentScale = ContentScale.FillWidth
            )
            AnimatedNicknameText(nickname = nickname)
            Text(
                text = "Welcome to the game Yamb",
                fontWeight = FontWeight.Medium,
                fontSize=24.sp,
            )
            Row({...}) {
                Button(
                    onClick = { onHowToPlayClick() },
                    modifier = Modifier.weight(1f),
                    colors = {...})
                { Text(text = "How to play", color = Color.White) }
                Spacer(modifier = Modifier.width(16.dp))
                Button(
                    onClick = { onStartGameClick() },
                    modifier = Modifier.weight(1f),
                    colors = {...})
                { Text(text = "Let's start", color = Color.Black) }
            }
        }
    }
}
```

Parametry:

- **nickname:** Parametr představující přezdívku uživatele.
- **onHowToPlayClick()**: Funkce lambda, která se spustí po kliknutí na tlačítko „How to play“.
- **onStartGameClick()**: Funkce lambada která se spustí po kliknutí na tlačítko „Let's start“.

Konstrukce této funkce začíná použití „Box“, který slouží jako kontejner pro ostatní komponenty. Tento kontejner vyplňuje maximálně dostupný prostor a má bíle pozadí. V rámci tohoto kontejneru je „Column“ který uspořádá veškeré potomky svisle. Jedna z komponent je předdefinovaná funkce Jetpack Compose s názvem „Image“ která zobrazuje obrázek, který je načítán z lokálního souboru v aplikaci (R.drawable.yamb). Po obrázku následují dvě komponenty „Text“. Ta první oslovuje osobu přezdívkou použití animace, zatímco druhá zobrazuje obecný pozdrav. Jako poslední je zde komponenta „Row“ která obsahuje dvě tlačítka a každé tlačítko má vlastní barvu pozadí a spouští příslušnou lambdu po kliknutí. Důležitou funkcí je „AnimatedNicknameText“ která jako parametr obsahuje právě přezdívku uživatele a zobrazí tak text v rámci animace tímto způsobem:

```
@Composable
fun AnimatedNicknameText(nickname: String) {
    //Tato proměnná udržuje stav animace.
    val animatedProgress = remember { Animatable(initialValue = 0f)}
    //Doba, která trvá na napsání jednoho znaku v milisekundách.
    val typingDurationPerChar = 130
    LaunchedEffect(nickname) {
        animatedProgress.animateTo(
         targetValue = 1f,
         animationSpec = tween(
         durationMillis = (nickname.length * typingDurationPerChar),
         easing = FastOutSlowInEasing))
    }
//Zobrazení část textu, který je momentálně viditelný podle průběhu
animace.
        val visibleText = nickname.take((animatedProgress.value *
nickname.length).toInt())
        Text(text = "Hey, $visibleText",)
}
```

Protože se jedná o animaci tak je klíčové použít „LaunchedEffect“ (kapitola 3.4), který zaručí aby se tahle část pustila asynchronně na jiném vlákně a tím zajistila

hladkou zkušenost při použití aplikace. Jednoduše řečeno pomáhá předcházet zpomalení nebo zamrzání uživatelského rozhraní.

Poté co uživatel stiskne na tlačítko „How to play“, se přesměruje na obrazovku s návodem, která podrobně vysvětluje pravidla a postup hry (viz. Obrázek 16). Tady je ukázka kódu třetí obrazovky:

```
@Composable
fun HowToPlayScreen() {
    Box(modifier = {...}) {
        Column(modifier = {...}) {
            HowToTitle()
            ObjectiveCard()
            EquipementCard()
            LazyVerticalGrid(
                columns = GridCells.Fixed(2),
                modifier = Modifier.padding(vertical = 8.dp)
            ) {
                items(stepsList.size) { index ->
                    val step = stepsList[index]
                    StepCard(
                        stepTitle = step.title,
                        expandedDescription = step.description,
                        modifier = {...}
                    ) }
            }
        }
    }
}
```

První tři komponenty, které jsou umístěny v sloupci nebudou vysvětleny, protože jejich definice je podobná jako u ostatních předchozích příkladů. Hlavním zaměřením bude funkce, kterou framework Jetpack Compose nabízí, a to je „LazyVerticalGrid()“.

Tato předdefinovaná funkce je určena k efektivnímu a línému zobrazení vertikální mřížky prvků. Cílem je maximalizovat efektivitu a využití paměti při práci s velkými datovými sadami nebo dynamickými daty, které nevyžadují okamžité načítaní a vykreslování. „LazyVerticalGrid“ načte pouze položky, které jsou aktuálně viditelné na obrazovce, spolu s několika dalšími položkami, které zaručí plynulé skrolování. Pomoci skrolování se dynamicky načítají nové položky bez zpoždění, a tak se zajišťuje že to uživatelské rozhraní zůstane responzivní a nedochází k zaseknutí. Komponenty jsou uspořádány svisle ve vzoru mřížky. To znamená že programátor může určit počet sloupců a položky tak budou uspořádány podle tohoto počtu. V tomto příkladu je počet sloupců určen na dva.

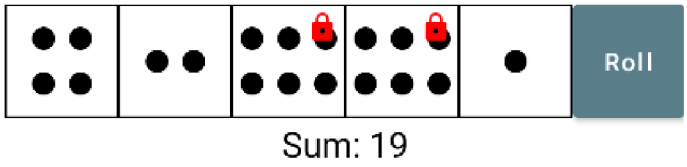
- **columns=GridCells.Fixed(2)**: Označuje počet sloupců v mřížce (v tomto příkladu dva).
- **items(stepsList.size){index->...}**: Tato funkce prochází seznam a generuje jednotlivé mřížky. „StepList.size“ definuje, kolik položek se zobrazí v mřížce.
- **val step= stepsList[index]**: Načte aktuální krok ze seznamu na základě aktuálního indexu.
- **StepCard()**: Vlastní Composable funkce, která zobrazuje podrobnosti o určitém kroku a představuje položku v mřížce.

Poslední důležitou komponentu tohoto příkladu je právě funkce „StepCard()“. Tato funkce se od ostatních liší tím, že obsahuje předdefinovanou komponentu nazvanou „Card“, na kterou lze kliknout. Po kliknutí uživatele na jednu položek z mřížky se tato otevře a poskytne další informace (viz. Obrázek 18). Otevíraní a zavíraní karty je regulováno pomoci logické „boolean“ hodnoty, jak je vidět z této ukázky kódu:

```
@Composable
fun StepCard(
    stepTitle: String,
    expandedDescription: String,
    modifier: Modifier = Modifier
) {
    var expanded by remember { mutableStateOf(false) }
    Card(
        modifier = modifier.clickable { expanded = !expanded },
        elevation = 6.dp, shape = RoundedCornerShape(8.dp),
    ) {
        Box(modifier = {...}) {
            Column {
                Icon(
                    Icons.Default.Info,
                    contentDescription = "Step Icon",
                    modifier = {...}
                )
                Spacer(modifier = {...})
                Text(
                    text = stepTitle,
                    {...}
                )
                if (expanded) {
                    Text(text = expandedDescription)
                } else {
                    Box(
                        modifier = {...},
                        contentAlignment = Alignment.Center
                    ) { Text(text = "Tap for more info", {...})}
                }}}}
```

Obrázek 18 Ukázka výsledku funkce „LazyVerticalGrid“ a „StepCard“
Zdroj: Autor

Poslední obrazovka, označena číslem čtyři (viz. Obrázek 16), obsahuje výsledkovou tabulku, do které uživatel zaznamenává body. Body jsou získané „házení“ kostek které jsou specifikovány pod tabulkou a jejich součet je zobrazen v dolní části. Po kliknutí na tlačítko umístěné napravo od řady obsahující kostky je každá kostka náhodně vržena, přičemž se zobrazí čísla v rozmezí od jedné do šesti. Uživatelé mají navíc možnost zamknout kostku kliknutím na ni, což jim umožnuje sledovat konkrétní kombinace podle jejich představ stejně jak je zde znázorněno:



Obrázek 19 Kostky v aplikaci Yamb
Zdroj: Autor

Zde je ukázka, jak pomoci Jetpack Compose lze vytvořit takovou kostku:

```
@Composable
fun Circle(color: Color, size: Dp) {
    Box(modifier = Modifier.size(size)
        .background(color, CircleShape))
}
@Composable
fun Dice(number: Int, locked: Boolean, onClick: (Boolean) -> Unit) {
    Box(
        modifier = {...}
    ) {
        val dotColor = Color.Black
        val dotSize = 13.dp
        val dotSpacing = 8.dp
        when (number) {
            1 -> {
                Box(
                    modifier = Modifier.fillMaxSize(),
                    contentAlignment = Alignment.Center
                ) { Circle(dotColor, dotSize) }
            }
            2 -> {...}
            3 -> {...}
            4 -> {...}
            5 -> {...}
            6 ->{...}
        }
        if (locked) {
            Icon(
                Icons.Default.Lock,
                contentDescription = "Locked",
                tint = Color.Red,
                modifier = {...}
            )
        } }
}
```

Poslední a nejvíc komplexní komponenta této aplikace je výsledková tabulka (viz. Obrázek 13). Zde je vysvětlení, toho co má následující kód dělat:

- Struktura tabulky: Tabulka má pevný počet sloupců(šest) a řádků (patnáct). Každá buňka tabulky může obsahovat textová data.
- Správa dat: Data tabulky jsou spravována pomocí „MutableStateList“.
- Zpracování výběru: Umožnuje výběr buněk. Po kliknutí na buňku (která není v prvním sloupci) se vybere a zobrazí se textové pole, do kterého může uživatel zadávat data. Vybraná pozice buňky je uložena do proměnné „selectedCellPosition“.
- Rozvržení: Rozvržení tabulky je reprezentováno pomocí sloupců a řádků. Každá buňka je bud „Text“ nebo „TextField“ podle toho, zda je vybrána nebo ne.

Je důležité poznamenat že následující kód je zkrácení a není kompletní z důvodu jeho velikosti a komplexnosti.

```
@Composable
fun TableLayout() {
    val numColumns = 6
    val numRows = 15
    val firstColumnCells = remember {
        mutableStateListOf<String>().apply {...}}

    val cells = remember {
        mutableStateListOf<MutableList<String>>().apply {
            repeat(numRows) { rowIndex ->
                add(mutableStateListOf<String>().apply {
                    repeat(numColumns) { columnIndex ->
                        if (columnIndex == 0) {
                            add(firstColumnCells[rowIndex])
                        } else { add("") }
                    }
                    )
            }
        }
    }

    var selectedCellPosition by remember
    { mutableStateOf(CellPosition(-1, -1)) }
    Column(modifier = {...}) {
        cells.forEachIndexed { rowIndex, row ->
            Row(modifier = {...}) {
                row.forEachIndexed { columnIndex, cellText ->
                    val isSelected = selectedCellPosition ==
CellPosition(rowIndex, columnIndex)
                    if (isSelected && columnIndex != 0) {
                        TextField(
                            value = cellText,
                            onValueChange = { newValue: String ->
                        cells[rowIndex][columnIndex] = newValue
                            },
                            modifier = {...}
                    )} else {
                        Text(
                            cellText,
                            Modifier
                           .weight(if (columnIndex == 0) 2f else 1f)
                           .border(width = 1.dp, Color.Black)
                           .clickable(
                                    enabled = !isSelected,
                                    onClick = {
 selectedCellPosition = CellPosition(rowIndex, columnIndex)
                                    }))}
                }
            }
        }
    }
}
```

Nyní je potřeba jenom tyto komponenty zkombinovat a vytvořit tak kompletní obrazovku:

```
@Composable
fun GameScreen() {
    val diceNumbers = remember { mutableStateListOf(1, 1, 1, 1, 1) }
    val lockedDiceIndices = remember { mutableStateListOf<Int>() }
    val sum = diceNumbers.sum()
    Column(modifier = {...}) {
        TableLayout()
        Box(modifier = {...}) {
            Row(modifier = {...}) {
                diceNumbers.forEachIndexed { index, number ->
                    Dice(
                        number = number,
                        locked = index in lockedDiceIndices,
                        onClick = { locked ->
                       if (locked) {lockedDiceIndices.remove(index)}
                       else { lockedDiceIndices.add(index) }
                        })
                }
                Button(
                    onClick = {
diceNumbers.indices.filterNot {index -> index in lockedDiceIndices }
         .forEach { index -> diceNumbers[index] = (1..6).random() }
                    },
                    modifier = {...}
                ) { Text(text = "Roll") }
            }
        } Text(text = "Sum: $sum")}}
```

Vysvětlení časti kódu:

- **val diceNumbers**: Seznam, který obsahuje čísla na kostkách a je počátečně naplněn pěti jedničkami. Na jeho základě budou tak vykresleny pět kostek pomoci funkce „forEachIndexed“.
- **val lockedDiceIndicies**: Seznam, který uchovává indexy kostek, které jsou zamčené.
- **val sum**: Vypočítá součet všech čísel na kostkách.
- **filterNot{index -> in lockedDiceIndicies}**: Tento krok filtruje indexy kostek, které nejsou obsaženy v seznamu „lockedDiceIndicies“. Jinými slovy, vybere indexy těch kostek, které nejsou zamčené.
- **forEach{index -> diceNumbers[index]=(1..6).random()}**: Při zmáčknutí tlačítka se pro všechny kostky které nejsou zamčené vybere náhodné číslo v rozsahu od jedné do šesti. Na základě tohoto čísla se potom vykreslí odpovídající kostka.

6 Shrnutí výsledků

S cílem vytvořit mobilní aplikaci pro Android s Kotlin a Jetpack Compose tento projekt přinesl důležité poznatky a pozorovatelné výsledky. V průběhu vývoje aplikace bylo dosaženo několika významných výsledků.

Z hlediska vývoje uživatelského rozhraní se Jetpack Compose ukázal jako revoluční. Vytváření dynamických a interaktivních uživatelských rozhraní je mnohem jednodušší a intuitivnější díky využití jeho deklarativní povahy. Předdefinované komponenty Jetpack Compose usnadňují modulární design a podporují udržovatelnost a znuvupoužitelnost kódu.

V průběhu vývojového procesu pomohlo použití Kotlinu. Jeho null-safety funkce, krátká syntaxe a silné rozšiřující funkce usnadnily tvorbu kódu a snížily riziko problémů za běhu aplikace. Navíc možnost používat trailing lambdy byla hodně užitečná. Tyto lambdy zlepšují čitelnost a udržovatelnost kódu uživatelského rozhraní tím, že umožnují stručnější a výraznější syntaxi při definování Composable položek. Tento vztah mezi programovacím jazykem Kotlin a deklarativním frameworkem Jetpack Compose je příkladem toho, jak mohou moderní vývojová paradigmata zvýšit produktivitu vývojářů a kvalitu kódu.

Aby byla zaručena bezproblémová uživatelská zkušenost v celé aplikaci, byli zavedeny techniky optimalizace výkonu. Využitím funkcí animace Jetpack Compose a funkce coroutines Kotlin pro asynchronní programování funguje aplikace optimálně za různých okolností použití. Konkrétně aplikace používá „LaunchedEffect“ k hladké koordinaci animací, například těch, které se objevují, když se zobrazí přezdívka uživatele. Díky tomu uživatelská zkušenost zůstane hladká, což zvyšuje celkovou spokojenost a použitelnost.

Dalším zásadním prvkem, který má za následek plynulý průběh a je snadno implementovatelný, jsou různé obrazovky, kterými aplikace disponuje a mezi kterýma se se plynule naviguje. Také je umožněná funkcionalita posílaní argumentů mezi obrazovky jako u příkladu, když z jedné obrazovky se přenáší přezdívka uživatele na druhou obrazovku, na které se potom vykreslí.

Jednou z nejpozoruhodnějších funkcí aplikace je zahrnutí komponenty interaktivní tabulky a házení kostek. Tabulka umožnuje uživatelům pracovat

s jednotlivými buňkami a poskytuje intuitivní rozhraní pro zadávání dat. Na druhou stranu funkcionalita házení kostek přináší nový prvek interaktivity, který uživatelům umožnuje hrát si s aplikací a zkoumat její možnosti. Tyto interaktivní komponenty činí aplikaci uživatelsky přívětivější.

Současně je kompletní projekt k dispozici v online repositáři (1) a je k dispozici pro stažení na Google Play Store. (2)

7 Závěry a doporučení

Tato práce ukázala důležitost propojení teoretického porozumění s praktickými dovednostmi při vývoje aplikací. S pouhým malým množstvím kódu mohou vývojáři vytvářet esteticky příjemné a užitečné aplikace pomoci Jetpack Compose a programovacího jazyka Kotlin.

Teoretická část zahrnuje základní komponenty a funkcionality které jsou potřebné pro vytvoření takové aplikace pro Android platformu. Tato část pokrývá širokou škálu témat, a tak nabízí důkladné pochopení deklarativní metodologie vývoje uživatelského rozhraní a zdůrazňuje její výhody, pokud jde o čitelnost kódu, opětovnou použitelnost a udržovatelnost. Zabývá se také coroutines v Kotlinu a tím, jak umožnují asynchronní programování což usnadňuje provádění úkolu efektivně a bez zastavení aplikace. Navíc tato část klade důraz na předdefinované funkce a komponenty které Jetpack Compose nabízí. Tyto integrované funkce osvobozují vývojáře od nutnosti zabývat se opakujícím se kódem a umožnují jim soustředit se na vytváření působivých uživatelských rozhraní.

Hlavním cílem praktické části bylo vyvinout plně funkční Android aplikaci za účelem převedení teoretických konceptů do reálných výsledků. Z tohoto důvodu byla vybrána implementace hry Yamb, a tak ukázat že i programátoři s omezenými zkušenostmi mohou takové aplikace úspěšně vyvíjet.

Jeden z problémů, která tato práce obsahuje je neschopnost aplikace ukládat data lokálně nebo na vzdálené úložiště. Vzhledem k tomu že uživatelé očekávají že jejich data budou přístupná i potom co zavřou aplikaci, tak najednou zjistí že jejich data jsou ztracena. Tato zkušenost představuje podstatnou překážku poskytování bezproblémové uživatelské zkušenosti. Implementace uložení dat by zlepšila užitečnost a pohodlí aplikace a umožnila by uživatelům bez námahy začít tam, kde skončili bez toho, aby se ty data ztratila.

Je důležité mít na paměti že společnost Google vyvíjí Jetpack Compose a lze tak očekávat pravidelné aktualizace a přidávání nových funkcí. Doporučuje se, aby vývojáři používali stránky „Android Developers“ poskytované právě společností Google (3). Je důležité zůstat informován a držet krok s nejnovějšími technologiemi a nástroji. Tato strategie zaručuje úspěch v oblasti mobilního vývoje.

8 Seznam použité literatury

8.1 Tištěné zdroje

[1] SMYTH, Neil. Jetpack Compose 1.2 Essentials. Payload Media, Inc., 2022. 545 s. ISBN: 978-1-951442-49-1.

[2] KÜNNETH, Thomas. Android UI Development with Jetpack Compose. Packt Publishing, 2022. 227 s. ISBN: 978-1-80181-216-0.

[3] WAMBUA, Madona S. Modern Android 13 Development Cookbook. Packt Publishing, 2023. 296 s. ISBN: 978-1-80323-557-8.

[4] BUKETA, Denis, PRASAD Prateek. Jetpack Compose by Tutorials. Copyright © 2023 Kodeco Inc. 441s. ISBN: 978-1950325832

[5] MURPHY, Mark L. Elements of Android Jetpack. CommonsWare, LLC, 2021. 911s. ISBN: 9781005883348.

8.2 Internetové *zdroje*

[6] The history of Kotlin, 2023 [online]. [cit 01.10.2023]. Dostupné z: https://www.oreilly.com/library/view/kotlin-for-enterprise/9781788997270/ea4ec584-db64-4026-89a8-2086301eb9c5.xhtml

[7] Millie Macdonald, 26.01.2021, Tiny Programming Principles: Immutability [online]. [cit 15.10.2023]. Dostupné z: https://www.tiny.cloud/blog/mutable-vs-immutable-javascript/#:~:text=Better%20caching%20%20You%20can%20cache,code%2C%20and%20track%20down%20bugs

[8] Sujatha Mudadla, 22.07.2023, Trailing lambda in Kotlin [online]. [cit 18.10.2023]. Dostupné z: https://medium.com/@sujathamudadla1213/trailing-lambda-in-kotlin-1155403e4028#:~:text=In%20Kotlin%2C%20“trailing%20lambda”%20refers%20to%20a%20syntactic%20feature,last%20argument%20of%20that%20function.

[9] Ossian Muscad, 22.11.2022, Synchronous Vs. Asynchronous: A Guide To Choosing The Ideal Programming Model [online]. [cit 19.10.2023]. Dostupné z: https://datamyte.com/synchronous-vs-asynchronous/

[10] Concurrency and coroutines, 01.09.2023 [online]. [cit 20.10.2023]. Dostupné z: https://kotlinlang.org/docs/multiplatform-mobile-concurrency-and-coroutines.html

[11] kmDev, 06.08.2023, Thread vs. Coroutines: Choosing the Right Concurrency Approach in Kotlin Android [online]. [cit 20.10.2023]. Dostupné z: https://medium.com/@mkcode0323/thread-vs-coroutines-choosing-the-right-concurrency-approach-in-kotlin-android-3a56368d9768#:~:text=Threads%20are%20a%20more%20traditional,execution%2C%20and%20better%20exception%20handling.

[12] Anton Spaans, 07.10.2019, How can we use CoroutineScopes in Kotlin? [online]. [cit 21.10.2023]. Dostupné z: https://medium.com/swlh/how-can-we-use-coroutinescopes-in-kotlin-2210695f0e89

[13] Scopes in Kotlin Coroutines, 15.10.2021 [online]. [cit 22.10.2023]. Dostupné z: https://www.geeksforgeeks.org/scopes-in-kotlin-coroutines/

[14] Google, Build better apps faster with Jetpack Compose [online]. [cit 30.10.2023]. Dostupné z: https://developer.android.com/jetpack/compose#:~:text=Jetpack%20Compose%20is%20Android%27s%20recommended,tools%2C%20and%20intuitive%20Kotlin%20APIs.

[15] tomerpacific, 09.08.2022, Is Jetpack Compose Ready for You? [online]. [cit 30.10.2023]. Dostupné z: https://betterprogramming.pub/is-jetpack-compose-ready-for-you-eae6c93ad3f8

[16] Imperative vs. Declarative Programming [online]. [cit 31.10.2023]. Dostupné z: https://programiz.pro/resources/imperative-vs-declarative-programming/

[17] Alex Styl, 17.12.2022, Everything you need to know about State in Jetpack Compose with examples [online]. [cit 05.11.2023]. Dostupné z: https://www.composables.com/tutorials/state

[18] Antonello Zanini, 20.08.2021, A complete guide to enum classes in Kotlin [online]. [cit 22.01.2024]. Dostupné z: https://blog.logrocket.com/kotlin-enum-classes-complete-guide/

[19] Milton Bradley, 1996, Yahtzee [online]. [cit 28.02.2024]. Dostupné z: https://www.hasbro.com/common/instruct/yahtzee.pdf

[20] Yahtzee, 15.02.2024, [online]. [cit 05.03.2024]. Dostupné z: https://en.wikipedia.org/wiki/Yahtzee

[21] Google for Developers, Compose layout basics, [online]. [cit 13.04.2024]. Dostupné z: https://developer.android.com/develop/ui/compose/layouts/basics

9 Přílohy

1) Pitropovski Daniel, Praktický projekt implementace deskové hry Yamb. In: github.com [online]. [cit 04.04.20224] Dostupné z: https://github.com/BargoEmbargo/Yamb
2) Pitropovski Daniel, Aplikace Yamb. In: play.google.com[online]. [cit 05.04.2024]. Dostupné z: https://play.google.com/store/apps/details?id=com.yamb.chatgpt
3) Google for Developers, Android for Developers [online]. [cit 04.04.2024]. Dostupné z: https://developer.android.com
4) Abhishek Singhal, 07.08.2023, Column and Row in Jetpack Compose [obrázek] [online]. [cit 05.03.2024]. Dostupné z: https://www.c-sharpcorner.com/article/column-and-row-in-jetpack-compose/

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání bakalářské práce

Autor: **Daniel Pitropovski**

Studium: I2100718

Studijní program: B1802 Aplikovaná informatika

Studijní obor: Aplikovaná informatika

Název bakalářské práce: **Vývoj Android aplikací s Jetpack Compose**

Název bakalářské práce AJ: Android Application Development using Jetpack Compose

Cíl, metody, literatura, předpoklady:

Cíl: Cílem této bakalářské práce je představit nově vytvoření framework Jetpack Compose. Na základě znalostí, které čtenář získá, bude moci vytvořit funkční aplikaci pro Android, která bude vizualně přitažlivá a moderní.

Osnova:

1. Úvod

2. Teoretická část

2.1 Popis programovacího jazyka Kotlin

2.2 Popis frameworku Jetpack Compose

2.3 Popis UI element pomoci Jetpack Compose

3. Praktická část

3.1 Popis aplikace

3.2 Vývoj aplikace

4. Výsledky a závěr

1. SMYTH, Neil. Jetpack Compose 1.2 Essentials. Payload Media, Inc., 2022. ISBN: 978-1-951442-49-1.
2. KÜNNETH, Thomas. Android UI Development with Jetpack Compose. Packt Publishing, 2022. ISBN: 978-1-80181-216-0.
3. WAMBUA, Madona S. Modern Android 13 Development Cookbook. Packt Publishing, 2023. ISBN: 978-1-80323-557-8.
4. BUKETA, Denis, PRASAD Prateek. Jetpack Compose by Tutorials. Copyright © 2023 Kodeco Inc. ISBN: 978-1950325832
5. MURPHY, Mark L. Elements of Android Jetpack. CommonsWare, LLC, 2021. ISBN: 9781005883348.

Zadávající pracoviště: Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu

Vedoucí práce: doc. Mgr. Tomáš Kozel, Ph.D.

Datum zadání závěrečné práce: 26.1.2021