Programarea aplicatiilor pe telefoane mobile

Student: Bogdan-Alexandru Dig

An universitar 2022 - 2023

Contents

1 Introducere

1.1 Context

Scopul acestei lucrări de laborator este de a explica într-un mod simplu, concis și ușor de înțeles etapele necesare proiectării și implementării unei aplicații mobile destinată telefoanelor cu sistem de operare Android.

1.2 Specificatii

Mediul de dezvoltare utilizat pentru implementarea si simularea aplicatie va fi Android Studio iar în cazul limbajului de programarea se va opta pentru Kotlin, recomandat pentru dezvoltarea aplicatiilor Android. Acest IDE dispune de numeroase instrumente care vor usura proiectarea aplicatiei, insa cele mai importante vor fi emulatoarele incorporate in IDE care vor putea simula rularea aplicatiei pe un telefon mobil. Bineinteles ca aplicatia propusa va putea fi rulata si pe dispozitivele fizice.

1.3 Obiective

Obiectivele acestui proiect vor fi de a implementa si proiecta o aplicatie mobila care sa printr-o succesiune de pasi, fiecare pe un ecran diferit, modul de dezvoltare al unei aplicatii Android. Se vor folosi toate resursele disponibile sistemului de operare cat si senzorii telefonului pentru a exemplifica cateva idei de programe simple ce pot fi realizate chiar si de utilizator, precum: utilizarea senzorilor de luminozitatea pentru a detecta daca e zi sau noaptea, lucrul cu camera foto, etc.

2 Limbajul Kotlin

Kotlin este un limbaj de programare creat de compania JetBrains, foarte des intalnit in dezvoltarea aplicatiilor mobile care faciliteaza implementarea datorita:

- sintaxei simple și concise
- Null safety (cea mai des intalnita exceptie in Java NullPointerException, in Kotlin este tratată folosind numeroase functii specifice let, also, apply etc)
- Funcții extensie, utile pentru a extinde functionalitatea unei clase deja implementate (String, Int, etc)
- Clasele data care incorporeaza metodele si functiile cele mai des implementate (equals, hash, toString, etc)
- Suprascrierea operatorilor

3 Layouts

Pentru implementarea interfetei grafice, clasele cele mai folosite sunt layout-urile, extrem de folositoare pentru poziționarea elementelor grafice cu care utilizatorul interacționează. Acestea pot fi implementate atât "programabil" (utilizând cod Kotlin), dar și în cod xml, adesea cel mai folosit deoarece asigură o separarea mai bună între modulele de logică a aplicației și cele de view (interfața utilizator). Cele mai folosite layout-uri sunt următoarele:

- FrameLayout
- LinearLayout
- RelativeLayout
- GridLayout

Fiecare din clasele de mai sus sunt proiectate pentru a potrivi elementele grafice pentru orice ecran de telefon. Acestea fac ca aplicația să fie fezabilă pentru oricare set de telefoane mobile Android, indiferent de dimensiune, model sau furnizor.

4 Activități și Fragmente

4.1 Activități

Pentru ca un utilizator să navigheze afară, înapoi și în interiorul unei aplicații mobile, activitățile au rolul de a asigura tranziția aplicației în diferite stări din ciclul său de viață. Activitățile oferă o gamă variată de "apeluri inverse" care au rolul de a ajuta ajuta activitatea știe starea actuală a aplicației, de exemplu dacă sistemul crează, oprește sau reia o anumită activitate sau dacă distruge procesul în care se află activitatea. Având în vedere aceste metode care asigură ciclul de viață al unei activități, acestea pot fi folosite pentru implementarea comportamentului aplicației atunci când, de exemplu, utilizatorul alege să părăsească sau să reintre în activitate. În alte cuvinte, fiecare funcție din ciclul de viață al unei activități conferă libertatea de a implementa o nouă funcționalitatea care să fie apropiată de anumită schimbare de stare. Astfel că, implementând funcționalitatea corectă la momentul potrivit va face ca aplicația să fie mai robustă și performantă. Revenind la ciclul de viață al unei aplicații, există un număr variat de evenimente care pot să îl influențeze. Poate că cel mai bun exemplu este cel în care utilizatorul trece orientarea aplicației din portret în landscape. Alte cazuri pot fi acelea în care limba device-ul este schimbată sau user-ul pur și simplu alege să lase aplicația deschisă și să pornească alta. Astfel că, în urma acestor evenimente, funcțiile ce se vor apela, tratând fiecare eveniment în parte vor fi următoarele:

- onPause()
- onStop()
- onDestroy()

4.2 Fragmente

Fragmentele, la fel ca și activitățile, sunt componente Android care conțin o parte din interfața de intrare a unei activități. După cum sugerează și numele, fragmentele nu sunt clase independente, ci depind de o anumită activitate, fiind înglobate în ciclul său de viață, având din multe puncte de vedere o funcționalitate asemănptoare cu activitățile. Principalele avantaje ale acestora sunt modularitatea, reutilizabilitate și adaptabilitatea.

5 Arhitectura MVVM (Model-View-ViewModel)

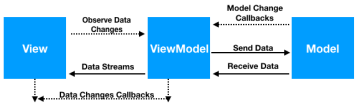
Model-View-ViewModel (MVVM) este o arhitectură folosită în dezvoltarea de interfeței utilizator (IU) pentru aplicații. MVVM se bazează pe separarea responsabilităților în trei componente distincte: model, vizualizare și model de vizualizare.

Modelul reprezintă datele și logica de afaceri a aplicației. Acesta conține informațiile cu care lucrează aplicația și modul în care aceasta le manipulează.

Vizualizarea este aspectul vizual al aplicației, cum arată și cum se comportă aceasta pentru utilizator. Aceasta poate include elementele de interfață ale utilizatorului, cum ar fi formularele, butoanele și listele, precum și animațiile și tranzițiile între diferitele elemente ale IU.

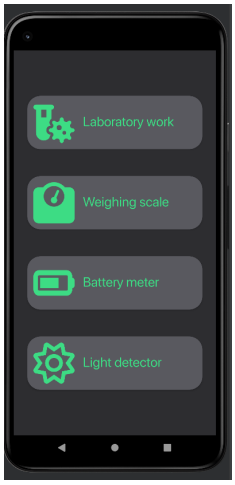
Modelul de vizualizare (MV) este componenta care leagă modelul și vizualizarea împreună. Acesta conține logica pentru a prezenta datele din model în vizualizare și a gestiona interacțiunile utilizatorului cu aceasta. De exemplu, dacă un utilizator face clic pe un buton într-o vizualizare, modelul de vizualizare poate gestiona această acțiune și poate actualiza modelul în consecință.

Această separare între model, vizualizare și model de vizualizare face ca aplicația să fie mai ușor de dezvoltat și întreținut, deoarece fiecare componentă are o responsabilitate specifică și poate fi testată independent. De exemplu, modelul poate fi testat în mod independent de vizualizare, iar vizualizarea poate fi testată în mod independent de modelul de afaceri.



6 Implementare

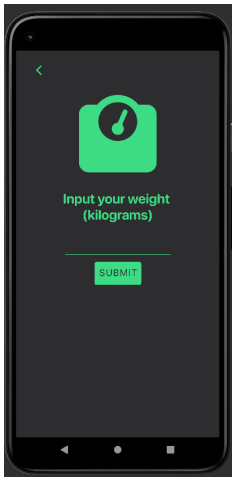
Pentru a exemplifica mai bine procesul de proiectare si implementare al unei aplicatii mobile, se vor prezenta in sectiunile care urmeaza cateva aplicatii mai mici, ordonate dupa gradul de implementare. Toate aceste trei proiecte respecta arhitectura MVVM și vor fi prezentate într-un meniu, după cum se vede îm figura de mai jos.:



6.1 Weighing scale

După cum sugerează și numele, acestă aplicație are rolul de a-i oferi utilizatorului informații despre greutatea sa. În esență, deoarece s-a dorit exemplificarea unei aplicații de tip "Hello World", utilizatorul își va introduce propria greutate iar aplicația va afișa un efect de încărcare care, după câteva secunde, va afișa într-un popup valoarea introdusă de către utilizator. Pentru implementare, interfata din figura de mai jos a fost implementată intr-un fragment din cadrul unei activitati care va cuprinde toate celelalte ecrane ce vor aparea in aplicatia principala.Pentru logica elementelor grafice (efectul de login) s-a creat un view model care are rolul de afișa animația de loading pentru un anumit interval de timp. Codul kotlin specific animației este următorul:

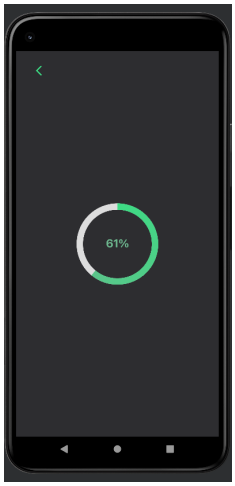
```
fun progressAnimation(prog : ProgressBar, showDialog: () -> Unit) {
    viewModelScope.launch {
        prog.visibility = View.VISIBLE
        delay(5000)
        prog.visibility = View.GONE
        showDialog()
    }
}
```



6.2 Battery Meter

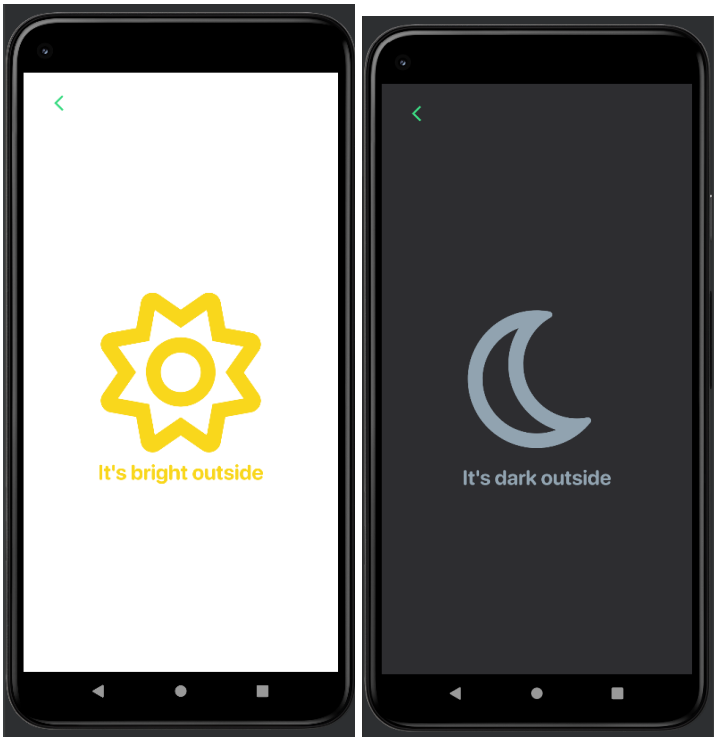
După cum sugerează și numele, acestă aplicație are rolul de a-i oferi utilizatorului informații legate de nivelul bateriei din telefon. La fel ca și la aplicația anterioară, s-a decis ca interfața să fie cuprinsă într-un fragment, fiind suprascrisă funcția de onCreateView. Cât despre clasa view model al fragmentului, cea mai importantă funcție este getBatteryLevel. Scopul aceste funcții este de a crea un Intent care îi va cere sistemului de operare să-i transmită o valoare legată de nivelul actual al bateriei. Întrucât se dorește afișarea unei valori procentuale, valoarea primită prin intermediul Intent-ului va fi ajustată.

```
fun getBatteryLevel(batteryIntent: Intent?) : Float{
    val level = batteryIntent?.getIntExtra(BatteryManager.EXTRA_LEVEL, -1)
    val scale = batteryIntent?.getIntExtra(BatteryManager.EXTRA_SCALE, -1)
    return scale?.let {
        if (level == -1 && scale == -1)
            50.0f
        else
            ((level?.toFloat()?.div(it.toFloat()))?.times(100))
    } ?: 0.0f
}
```



6.3 Light Detector

După cum sugerează și numele, acestă aplicație are rolul de a-i oferi utilizatorului informații legate de nivelul de lumină prezent în mediul exterior. Interfața utilizator a fost implementată în interiorul unui fragment, aceasta reprezentând partea cea mai ușoară din procesul de implementare. Cât despre logica din spate, aici lucrurile încep să se complice. În primul rând o limitare a acestei funcționalități este disponibilitatea unui senzor de lumină, acesta nefiind prezent în toate telefoanele mobile. În al doilea rând, pentru a realiza un răspuns dinamic din partea interfeței grafice, a fost nevoie ca la nivelul variabilei isLight din cadrul view modelului să folosească design pattern-ului "Observer". La fiecare schimbare a valorii isDark, observer-ul va invoca metodele specifice de schimbare al interfeței grafice. Pentru a prelua datele senzorului de lumină a fost nevoie de asemenea de crearea unui listener, utilizând conceput de LiveData, specific limbajului kotlin și mediului de programare Android. Astfel că, după implementare clasei MeaasurableSensor, clasa de view model va fi mult mai concis și flexibil de tratat pentru modificările ce pot apărea în viitor. Funcția care se ocupă cu observarea senzorului de lumină este următoarea:



(a) Interfața grafică atunci când senzorul detectează lumină

(b) Interfața grafică atunci când senzorul nu detectează lumină

```
var lightSensor: MeasurableSensor? = null
private val _isDark = MutableLiveData(false)
val isDark: LiveData<Boolean> = _isDark

fun observeLightSensor() {
    lightSensor?.startListening()
    lightSensor?.setOnSensorValuesChangedListener { values ->
        val lux = values[0]
        _isDark.value = lux < 60f
    }
}
```

Cât despre clasa LightSensor, MeasurableSensor, aceasta va avea rolul a abstractiza senzorii telefonului care pot să ofere o valoare numerică, măsurabilă. Clasa Android Sensor se va ocupa cu generalizarea metodelor de listening, fiind la curent cu orice schimbare care poate apărea pe valoarea oferită de senzori în timp real.

7 Testare

Testarea unei aplicații Android implică verificarea funcționalităților și a performanceului aplicației în diferite condiții și pe diferite dispozitive pentru a se asigura că aceasta rulează corect și îndeplinește cerințele definite. Există mai multe tipuri care se pot face pentru a verifica o aplicație Android:

- Teste unitare: aceste teste verifică funcționalitatea unui modul individual al aplicației și se folosesc pentru a găsi și remedia erorile cât mai devreme posibil în procesul de dezvoltare.
- Teste de integrare: aceste teste verifică modul în care diferite module ale aplicației funcționează împreună și se folosesc pentru a găsi problemele care pot apărea atunci când diferite componente sunt combinate.
- Teste de acceptare: aceste teste verifică dacă aplicația îndeplinește cerințele specifice și se folosesc pentru a asigura că aplicația este gata pentru lansare.
- Teste de performanță: aceste teste verifică viteza și stabilitatea aplicației și se folosesc pentru a găsi problemele care pot apărea în condiții de încărcare mare sau atunci când aplicația rulează pe dispozitive cu specificații hardware limitate.

Pentru a efectua testele, se pot folosi diverse instrumente și framework-uri de testare, cum ar fi JUnit, Espresso, Robotium, etc. Este important de acordat o atenție deosebită testării aplicației, deoarece o un sistem Android cu probleme poate duce la o experiență neplăcută pentru utilizatori și poate afecta încrederea acestora în aplicația ta.

În aplicațiile de mai sus, testarea a fost din punct de vedere al funcționalității, verificându-se ca acestea să ofere rezultatele așteptate pentru o gamă variată de date de intrare. În cazul primei aplicații, s-a optat pentru introducerea atât unor date numerice și corecte, cât și a unor șiruri de caractere care nu pot reprezenta o valoare pentru o greutate. În ambele cazuri, sitemul a răspuns corect, atenționând

user-ul atunci când acesta a introdus date greșite. Cât pentru celelalte două aplicații, aceasta a fost realizată pe o gamă variată de telefoane și pentru nivele diferite al luminozității și bateriei. Toate aceste teste au fost realizate cu succes.

8 Bibliography

- https://www.geeksforgeeks.org/kotlin-programming-language/
- https://developer.android.com/kotlin/first
- https://developer.android.com/codelabs/kotlin-bootcamp-welcome
- https://www.youtube.com/@PhilippLacknr
- https://www.digitalocean.com/community/tutorials/android-mvvm-design-pattern
- https://fontawesome.com/