Usporedba MVVM i MVP arhitektura pri izradi Android aplikacija na primjeru vođenja skladišta

Zagoršćak, Martin

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac Z1P - Obrazac za ocjenu završnog rada na preddiplomskom sveučilišnom studiju

Osijek, 04.09.2020.

Odboru za završne i diplomske ispite

Prijedlog ocjene završnog rada na preddiplomskom sveučilišnom studiju

Ime i prezime studenta:	Martin Zagoršćak
Studij, smjer:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	R4160, 20.09.2019.
OIB studenta:	63420626835
Mentor:	Doc.dr.sc. Mirko Köhler
Sumentor:	Luka Omrčen
Sumentor iz tvrtke:	
Naslov završnog rada:	Usporedba MVVM i MVP arhitektura pri izradi Android aplikacija na primjeru vođenja skladišta
Znanstvena grana rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Predložena ocjena završnog rada:	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 2 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	04.09.2020.
Datum potvrde ocjene Odbora:	09.09.2020.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

IZJAVA O ORIGINALNOSTI RADA

Osijek, 14.09.2020.

Ime i prezime studenta:	Martin Zagoršćak
Studij:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	R4160, 20.09.2019.
Turnitin podudaranje [%]:	7

Ovom izjavom izjavljujem da je rad pod nazivom**: Usporedba MVVM i MVP arhitektura pri izradi Android aplikacija na primjeru vođenja skladišta**

izrađen pod vodstvom mentora Doc.dr.sc. Mirko Köhler

i sumentora Luka Omrčen

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

USPOREDBA MVVM I MVP ARHITEKTURA PRI IZRADI ANDROID APLIKACIJA NA PRIMJERU VOĐENJA SKLADIŠTA

Završni rad

Martin Zagoršćak

Osijek, 2020

SADRŽAJ

1. UVOD

Kompleksnost izrade Android aplikacije ovisi o samoj funkcionalnosti planirane aplikacije te uzevši u obzir kvalitetno korisničko iskustvo, te kako bi se planirani ciljevi aplikacije izvršili potrebno je napisati velike količine koda te dodavanjem svake nove funkcionalnosti koje u početku možda nisu bile planirane uz to i čitanje koda može biti povelik problem. U sveukupnom procesu kreiranja nove aplikacije, neovisno koliko ljudi sudjeluje u procesu, više vremena se provodi čitajući kod nego pišući kod te se uvijek preporučuje dobro promišljanje prije početka pisanja te se koristi poznata fraza:“ Think first, code later“ te takvim načinom razmišljanja neki mogući budući problemi koju mogu spontano nastati eliminirani su u početku te nije potrebno trošiti vrijeme ispravljajući ih. Korištenjem praksi koje omogućavaju „čist kod“ koji će se kasnije moći lakše čitati te potom lakše dodavati nove funkcionalnosti, kao na primjer: korištenje pravilnih imena pri imenovanju klasa, atributa, funkcija i ostalo, pisanje jednostavnih funkcija, pravilno i razumljivo pisanje komentara u kodu, formatiranje koda te praćenje SOLID, KISS i drugih načela. Uporabom većine navedenih praksi u cilju „čistog koda“ i dalje se nailazi na probleme, aplikacija postaje stabilna i čista tek kada ona ima svoju arhitekturu. Cilj ovog završnog rada je prezentirati različite arhitekture prilikom izrade aplikacije. Korištenjem neke od dobro poznatih arhitektura omogućava: odvajanje funkcionalnosti od svog korisničkog sučelja u svrhu lakšeg čitanja koda, dodavanja novih funkcionalnosti, testabilnosti i ostalih procesa koje omogućuju jednostavniju, ali kvalitetniju izradu aplikacije. Najveća pogreška odnosno najgori stav programera je razmišljanje tako da aplikacija neće zahtijevati ažuriranja i da će zauvijek ostati na prvoj finalnoj verziji, te takvim načinom razmišljanja dolazi do najjednostavnijeg načina pisanja koda koji izaziva lošu praksu izrade aplikacije koji se protivi „čistom kodu“, ovaj pristup još se naziva „naivnom implementacijom“.

1.1. Zadatak završnog rada

Zadatak završnog rada je objasniti „Model-View-Presenter“ (MVP) i „Model-View-ViewModel“ (MVVM) arhitekturu, usporediti arhitekture, objasniti pojedine elemente arhitekture te izraditi aplikaciju vođenja skladišta na dva načina, koristeći MVP arhitekturu i MVVM arhitekturu. Na temelju izrađenih aplikacija usporediti i analizirati te odabrati koja arhitektura bolje odgovara aplikaciji vođenja skladišta.

2. PREGLED ARHITEKTURA ZA IZRADU APLIKACIJA

Problem loše organizacije koda rješavaju arhitekturni obrasci koji kod dijele na slojeve. Svaki sloj ima svoju ulogu i određen proces komunikacije među ostalim slojevima. Model-Pogled-Kontroler (eng. Model-View-Controller – MVVC) je tradicionalno rješenje razdvajanja koda u slojeve. Danas, Model-Pogled-Prezenter (eng. Model-View-Presenter – MVP) i Model-Pogled-Model pogleda (eng. Model-View-ViewModel – MVVM) obrasci su najpopularniji pri izradi Android aplikacija te nude jednostavniju implementaciju, testabilnost i lakše održavanje koda.

2.1. MVP

Model-Pogled-Prezenter (eng. Model-View-Presenter - MVP) je arhitektura koja je derivat od tradicionalne Model-View-Controller arhitekture. MVP arhitektura se sastoji od 3 sloja: sloj modela, sloj pogleda i prezentacijski sloj, svaki sloj je precizno definiran sa svojom ulogom u aplikaciji. Cilj arhitekture je uspješno razdvajanje poslovne logike od pogleda i vezanje pogleda i prezentera putem slabe poveznice (eng. loose coupling) koja se postiže preko sučelja (eng. interface). [1]

Uloga sloja modela je strukturno čuvanje podataka za prikaz podataka u aplikaciji preko sloja pogleda. Korisnik aplikacije nije upoznat sa strukturom sloja modela. Sloj ima mogućnost povezivanja s vanjskim entitetima i komunikaciju s bazama podataka i sl.

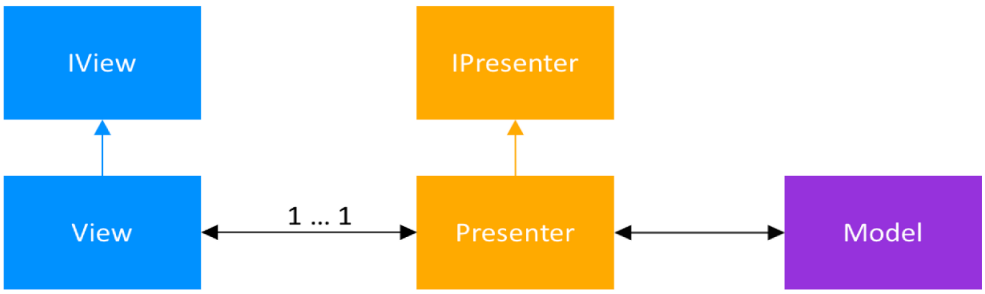
Uloga prezentacijskog sloja je izvršavanje definiranih akcija na temelju nastalih događaja u sloju pogleda te direktna interakcija sa slojem modela gdje dohvaća potrebne podatke i izvršava promjenu ovisnu o događaju. Kada prezenter primi nove podatke iz sloja modela on ih obrađuje i prosljeđuje gotove za prikaz sloju pogleda. Pogled i prezenter su povezani putem slabe reference koristeći sučelje, unaprijed definiranim ugovorom. U suštini on predstavlja posrednika između sloja pogleda i sloja modela.

```
interface ContractSignIn {
    interface ViewContract {
        fun onConnected()
        fun onNotConnected()
        fun setInputErrors(emailError:String, passwordError:String)
        fun onCorrectInput(email: String, password: String)
        fun onSuccessfulSignIn()
        fun onUnsuccessfulSignIn()
    }

    interface PresenterContract {
        fun getCurrentUser(): FirebaseUser?
        fun checkConnectivity(cm:ConnectivityManager)
        fun checkInput(email: String, password: String)
        fun signIn(email:String, password:String)
    }
}
```

Slika 2.1. Primjer ugovora između pogleda i prezentera

Sloja pogleda služi za prikaz korisničkog sučelja i podataka dobivenih iz sloja modela i obavještavanje prezentera da je nastao neki događaj na koji prezenter reagira i izvršava određeni zadatak odnosno funkciju. Sloj pogleda strogo treba što manje znati o poslovnoj logici te on samo komunicira sa svojim prezenterom ovisno što mu treba i potom prikazuje podatke putem sučelja.

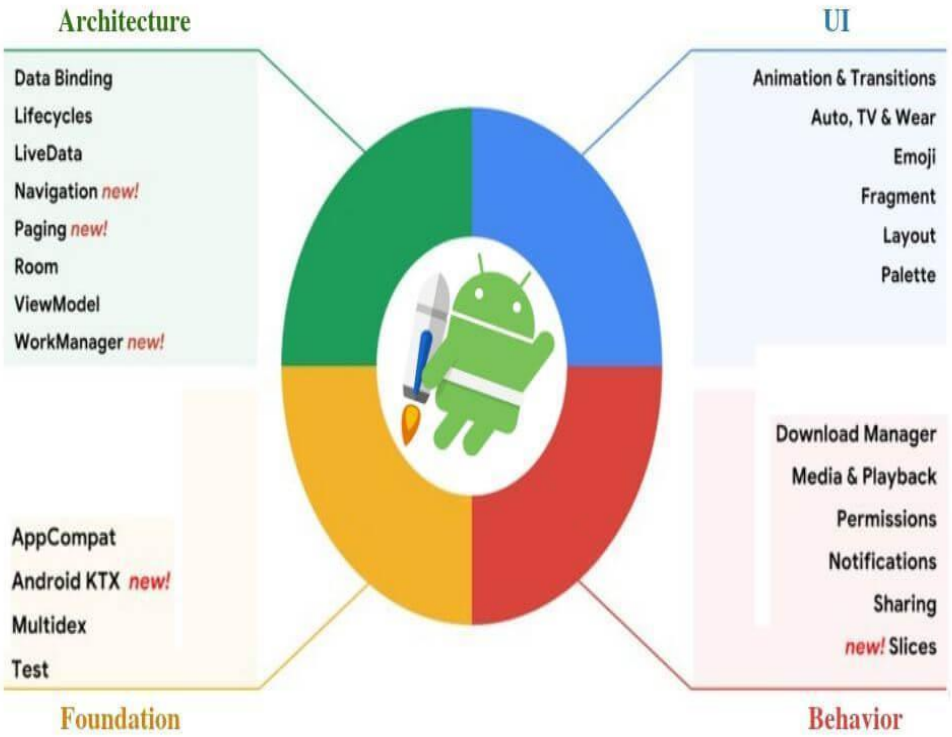
Razlikuju se od tradicionalnog MVC arhitekture po tome što se Kontroler sloj temelji na ponašanju pogled sloja i može biti vezan uz više pogleda, također u MVC-u pogled direktno komunicira s modelom. Unutar MVP arhitekture prezenter služi kao posrednik i obavlja sve akcije vezano za pogled sloj zato što je model sloj odvojen od pogled sloja, tako očito onda prezenter sloj ažurira pogled.



Slika 2.2. Struktura MVP arhitekture [4]

2.2. MVVM

Nova arhitektura Model-Pogled-Model pogleda (eng. Model-View-ViewModel - MVVM) je unaprijeđena MVP arhitektura koja je bogatija Google-ovim dodatnim klasama i komponentama koje olakšavaju lakšu implementaciju i čišći kod.

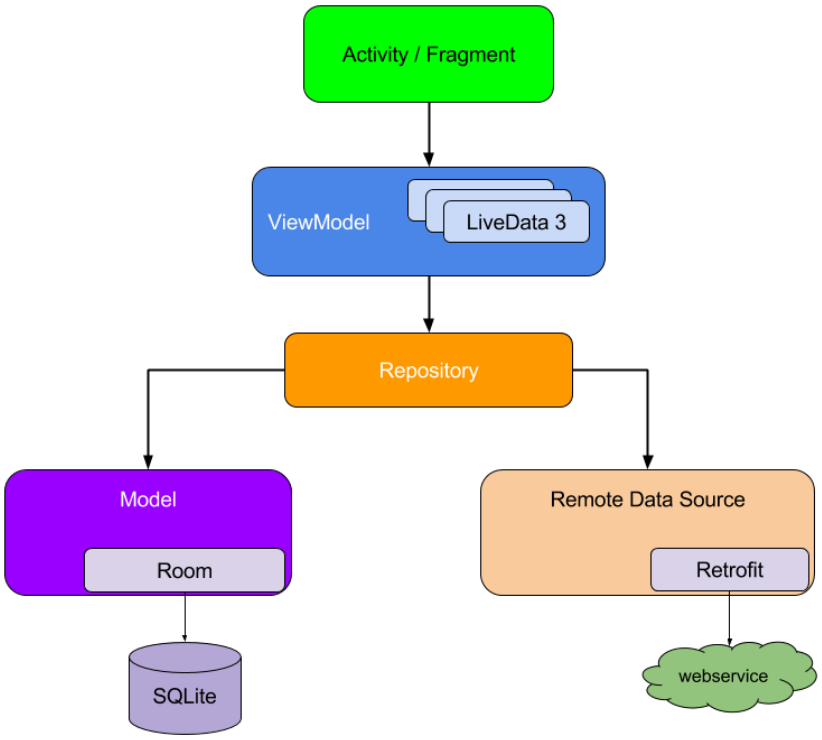


Slika 2.3. Android Jetpack - pomoćni library [7]

Kako je naglašeno u gornjem dijelu rada, MVVM je sličan MVP arhitekturi, razlika između prezentera i modela pogleda gdje model pogleda ne sadrži referencu na pogled i nema nikakva znanja o pogledu koji ga koristi, dok u MVP arhitekturi pogled i prezenter su dvosmjerno povezani. Također dodatak MVVM strukturi predstavlja dodatni pomoćni element: repozitorij s ulogom dohvaćanja podataka modelu pogleda.

Pogled je pretplaćen na podatke modela te osluškuje model ako dođe do promjene strukturiranih podataka. Prilikom promjene podataka unutar modela, aktivira se ažuriranje korisničkog sučelja. Ovo svojstvo definira se korištenjem „Live Data“ komponente. Uz podatke koji se ažuriraju u realnom vremenu (eng. Live Data) koji značajno smanjuje količinu koda k tome još postoji i podatkovna veza (eng. Data Binding). Podatkovna veza predstavlja tehniku u kojoj se dohvaća dio pogleda iz modela pogleda i sinkronizirano prikazuje podatke u korisničkom sučelju u skladu s promjenama podataka unutar svog izvora, uz takav pristup omogućava pogledu da može sadržavati

reference na više modela pogleda, dok pogled u MVP arhitekturi može imati samo jednog prezentera.

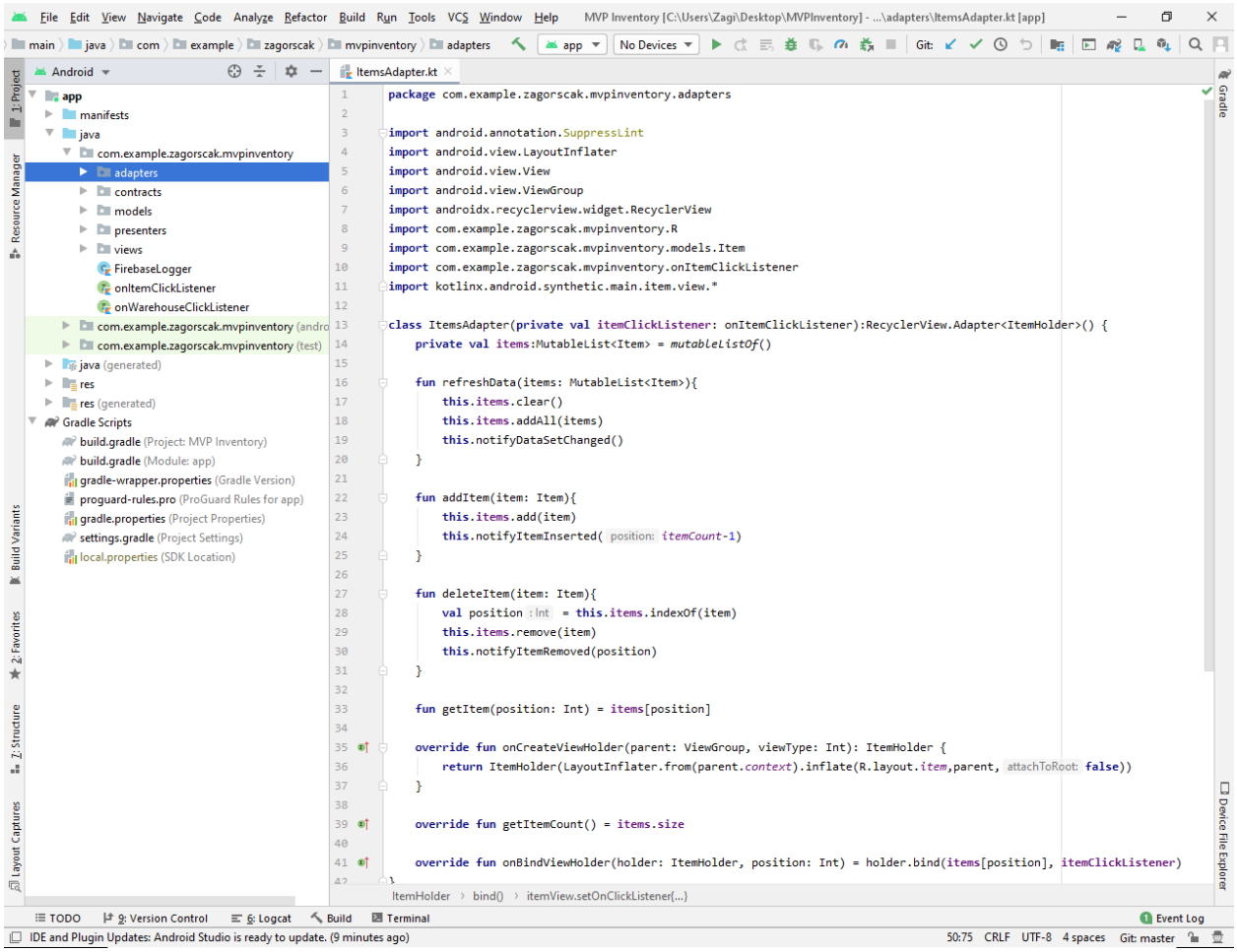


Slika 2.4. Struktura MVVM arhitekture [5]

3. RAZVOJNO OKRUŽENJE I PROGRAMSKI JEZIK

3.1. Android Studio

Android studio je integrirana okolina za razvoj Android aplikacija. Temeljen odnosno izgrađen na JetBrains-ovom IntelliJ-u, prva stabilna verzija predstavljena u prosincu 2014. godine. Danas je Android Studio najpopularnija okolina za razvoj Android aplikacija te ona nudi: brzo pokretanje koda na uređaju ili emulatoru, inteligentni code emulator, poboljšani emulator, podrška za sve uređaje, predloške i integraciju s GitHub-om, alate za testiranje i mnoge frameworke.

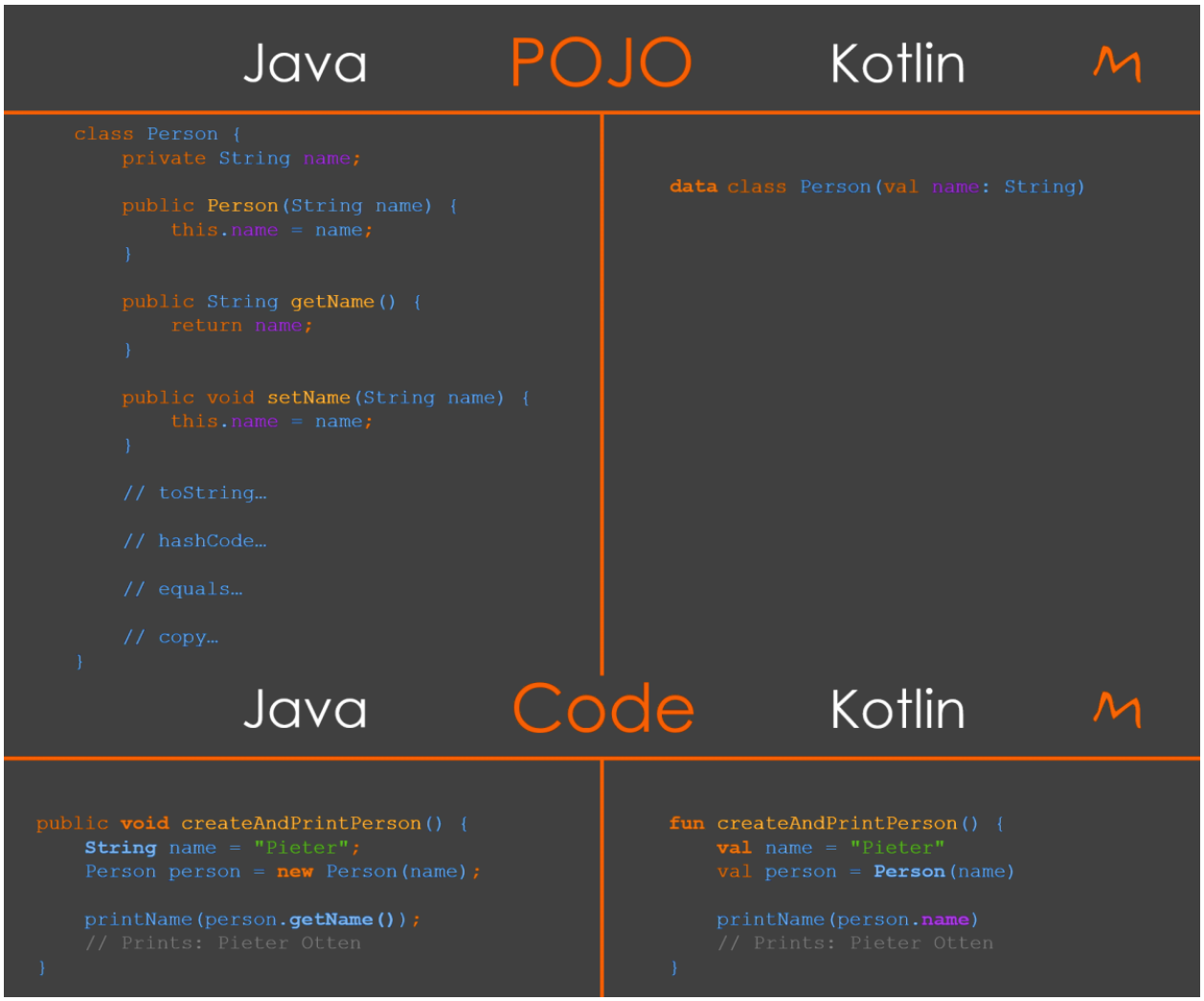


Slika 3.1. Korisničko sučelje Android Studio

3.2. Kotlin

Kotlin je programski jezik za razvoj multiplatformskih aplikacija. Razvio ga je JetBrains, početak projekta Kotlin je objavljen 2011. godine te prva verzija je objavljena u veljači 2016. godine. Vrlo

je sličan Javi, nadmašuje Javu i ističe svoje značajke kao što su: sažetost (skraćivanje koda koji se ponavlja), sigurnost (izbjegavanje raznih errora kao što je „NullPointerException“), interoperabilnost, korištenje u raznim Java razvojnim okolinama (IntelliJ IDEA, Android Studio, Eclipse ...) [3]



Slika 3.2 Usporedba Kotlina s Javom [2]

4. POHRANA PODATAKA

Firebase je platforma koja omogućuje kreiranje web i mobilnih aplikacija, pruža usluge kao što su: baza podataka u stvarnom vremenu, provjera autentičnosti korisnika i pohranu podataka na oblaku. Za potrebu pohrane podataka ovog završnog rada korištena je baza podataka u stvarnom vremenu.

Firebase-ova baza podataka u stvarnom vremenu je NoSQL baza, specifična po tome što se podaci prikazuju i mogu mijenjati u isto vrijeme putem različitih uređaja, te su podaci dostupni i kada aplikacija nema pristupa internetu. Podaci su pohranjeni u JSON formatu. Promjenom na promatranom polju podataka, firebase pomoću okidača događaja šalje obavijest aplikaciji da se dogodilo ažuriranje podataka, te aplikacija osvježava svoje podatke.

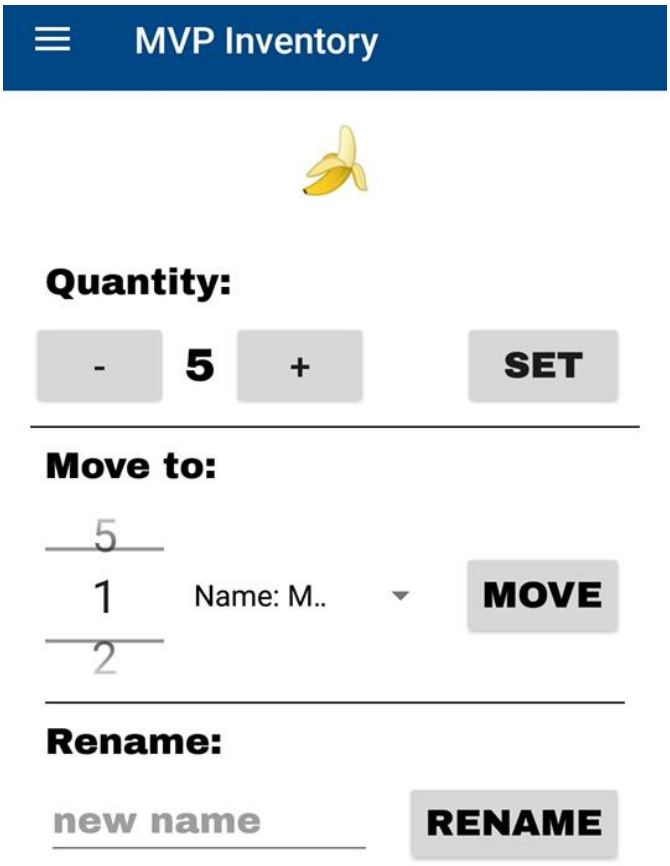
Slika 4.1 Podaci pohranjeni u Firebase bazi podataka stvarnog vremena

5. IMPLEMENTACIJA I USPOREDBA MVP I MVVM ARHITEKTURE

5.1. IMPLEMENTACIJA MVP ARHITEKTURNOG OBRASCA

Za razliku od pisanja aplikacije bez korištenja arhitekturnog obrasca, MVP arhitekturni obrazac aplikaciju dijeli u slojeve: modeli, pogledi i poslovna logika izdvojena u zasebnu klasu koja predstavlja prezenter. Korištenjem ovog obrasca aplikaciji se pridonosi bolja organiziranost, jasan smisao i čišći kod. Navedeni obrazac preporučuje se razvojnim prograrmerima koji nemaju iskustva u korištenju arhitekturnih obrasaca radi svoje jednostavne implementacije.

Na slici 5.1. se nalazi jedan od fragmenata koji spada u sloj pogleda te na njemu će se pojasniti implementacija MVP arhitekturnog obrasca. Funkcije na fragmentu su: postavljanje količine objekta u skladištu, premještanje objekta u neko drugo skladište i preimenovanje objekta. Promjenom svojstava objekta (imena ili količine) automatski se ažurira pogled. U slučaju brisanja objekta skladišta sa drugog uređaja, a da je pri tome prikazan fragment objekta skladišta, prikazuje se prikaz fragmenta s popisom skladišta koji je i ujedno prvi fragment koji je prikazan nakon korisničke prijave.



Slika 5.1. Pogled definiran u xml-u

Smisao obrasca je da svaki pogled ima svoj prezenter, sva poslovna logika sadržana je u prezenteru tako da pogled samo prikazuje potrebne podatke koje prezenter dostavi.

5.1.1. Model

Sloj modela MVP obrasca čine servisi ili repozitoriji koji dohvaćaju i spremaju podatke, može se reći da model predstavlja domenu objekta. Konkretno u ovom slučaju sloj modela ove aplikacije čine klase koje čuvaju podatke o objektu kao što su skladište, objekt u skladištu.

```
class Warehouse{
    var id:String = ""
    var address:String = ""
    var maxCapacity:Int = 0
    var name:String = ""
    var items = HashMap<String,Item>()
    override fun toString(): String {...}
    @Exclude
    fun getCurrentCapacity():Int{...}
}
```

Slika 5.2. Model skladišta

```
class Item {
    var id:String = ""
    var name:String = ""
    var quantity:Int = 0
}
```

Slika 5.3. Model objekta u skladištu

Postupak uključivanja Firebase-a u Android Studio projekta je jednostavan. Potrebno je :

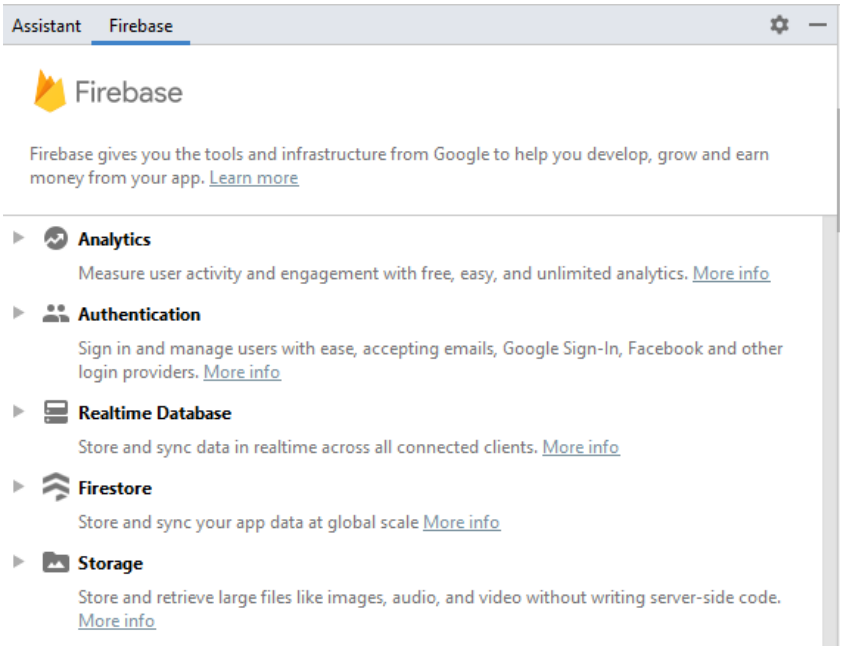
1. Povezivanje aplikacije s Firebase-om
2. Dodavanje servisa koje Firebase nudi, u ovom projektu korišten je sustav za autentifikaciju korisnika i baza podataka stvarnog vremena
3. Dohvaćanje instance servisa
4. Postavljanje reference na bazu podataka
5. Postavljanje promatrača na promjenu promatrane vrijednosti
6. Implementacija potrebnih funkcija za obradu događaja

Funkcija „onCancelled“ se okida kada dođe do pogreške na poslužitelju. Funkcija „onDataChange“ se koristi za dohvaćanje podataka iz baze te koju prezenter koristi kako bi uspio ažurirati pogled sa novim podacima.

```
private val mDatabase : FirebaseDatabase  = FirebaseDatabase.getInstance()
private val mDbRef : DatabaseReference  = mDatabase.reference.child( pathString: "warehouses")

init {
    mDbRef.addValueEventListener(object : ValueEventListener {
        override fun onCancelled(p0: DatabaseError) {

        }

        override fun onDataChange(p0: DataSnapshot) {

        }
    })
}
```

Slika 5.4. Predložak korištenja Firebase baze podataka stvarnog vremena

Detaljan postupak za korištenje pojedinog servisa prikazan je u Android Studio Assistant-u pod temom Firebase.



Slika 5.5. Android Studio Assistant za Firebase

5.1.2. Ugovor

Kako bi se stvorila veza između pogleda i prezentera potrebno je stvoriti ugovor među njima, a to se postiže kreiranjem sučelja (eng. interface) koji jasno definira metode pogleda i metode prezentera. Čitanjem ugovora u koliko god velikom projektu jednostavno se pronalazi uloga svakog od pogled-prezenter para. Svakom novom funkcionalnosti između pogleda i prezentera potrebno je prvo u ugovor dodati nove funkcije potom implementirati u pogled i prezenter.

```
interface ContractItemFragment {
    interface ViewContract {
        fun setSpinnerData(warehouses: List<Warehouse>)
        fun showToast(message: String)
        fun setUI()
        fun changeToWarehousesFragment()
    }

    interface PresenterContract {
        fun getItemName(): String
        fun getItemQuantity(): Int
        fun setNewQuantity(newQuantity: Int)
        fun renameItem(newName: String)
        fun moveItem(quantity: Int, destinationWarehouse: Warehouse)
        fun setView(viewItem: ContractItemFragment.ViewContract?)
        fun getMaxItemQuantity(): Int
        fun deleteItem()
    }
}
```

Slika 5.6. Ugovor između pogleda i prezentera za ItemFragment

5.1.3. Pogled

Nakon definiranog ugovora, potrebno je stvoriti fragment i njegov prezenter te implementirati sučelje za pogled i prezenter. Komunikacija između pogleda i prezentera vrši se isključivo putem propisanog ugovora odnosno sučelja, pogled ima referencu na prezenter i prezenter ima referencu na pogled koje su tipa kao i podsučelja ugovora. Svaka klasa koja prihvaća ugovor, obvezna je sve metode toga ugovora prepisati odnosno implementirati. Referenca prezentera na pogled ostvaruje se preko konstruktora ili metode koja postavlja referencu.

```
class ItemFragment : Fragment(), ContractItemFragment.ViewContract {
    private lateinit var presenter: ContractItemFragment.PresenterContract
    private var changeToWarehouseFragmentWhileOnPause = false

    companion object {
        fun getInstance(warehouseID: String, itemID: String): ItemFragment {
            val itemFragment = ItemFragment()
            itemFragment.presenter = ItemFragmentPresenter(itemFragment, warehouseID, itemID)
            return itemFragment
        }
    }
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {...}
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {...}

    override fun onDestroy() {
        presenter.setView(null)
        super.onDestroy()
    }
    private fun setListeners() {...}
    override fun setUI() {...}
    override fun changeToWarehousesFragment() {...}
    private fun clearBackStack() {...}
    override fun onResume() {...}
    override fun setSpinnerData(warehouses: List<Warehouse>) {...}
    override fun showToast(message: String) {...}
}
```

Slika 5.7. Implementiran MVP obrazac na strani pogleda

Značajan problem kod ovog arhitekturnog obrasca stvara to što je prezenter „zauvijek“ vezan za pogled, a problem stvaraju situacije kao što je pokušaj ažuriranja pogleda koji se više ne koriste. Primjer takve situacije može biti da prezenter šalje poslužitelju zahtjev koji traje 10 ili više sekundi

dok korisnik pritom zatvori taj pogled koji treba biti ažuriran. Navedeni problem rješava se prepisivanjem „onDestroy()“ metode u pogledu i u prezenteru se postavi referenca na pogled na null vrijednost.

5.1.4. Prezenter

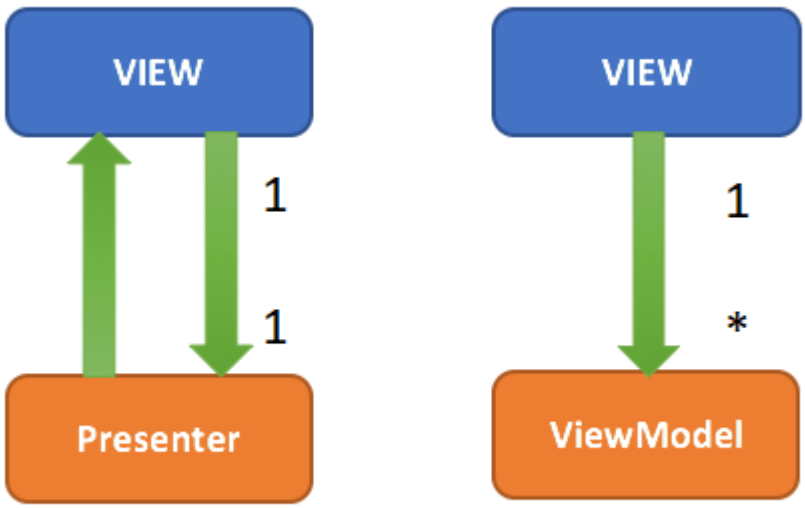
Funkcija prezentera je dostavljanje potrebnih podataka za prikaz na pogledu, podatke pribavlja koristeći Firebase bazu podataka stvarnog vremena te na svaku promjenu promatranih podataka prezenter je obaviješten i pohranjuje što mu treba i potom dostavlja podatke u pogled kako bi ga ažurirao. U ovom slučaju za fragment objekta skladišta u pogledu potrebno je ažurirati promjene koje se dogode na objektu skladišta (naziv i količina) i popis skladišta koji se prikazuje u opciji premještanja objekta skladišta u drugo skladište, također prezenter pohranjuje podatke u Firebase koje korisnik želi i zapisuje aktivnosti koje korisnik uradi tijekom korištenja aplikacije.

```
class ItemFragmentPresenter(
    private var viewItem: ContractItemFragment.ViewContract?,
    private var warehouseID: String,
    private var itemID: String
) : ContractItemFragment.PresenterContract {
    private lateinit var warehouse: Warehouse
    private lateinit var item: Item
    private val mDatabase : FirebaseDatabase  = FirebaseDatabase.getInstance()
    private val mDbRef : DatabaseReference  = mDatabase.reference.child( pathString: "warehouses")
        .child(warehouseID)

    init {...}

    override fun getItemName() = item.name
    override fun getItemQuantity() = item.quantity
    override fun getMaxItemQuantity(): Int {...}
    override fun setNewQuantity(newQuantity: Int) {...}
    override fun renameItem(newName: String) {...}
    override fun moveItem(quantity: Int, destinationWarehouse: Warehouse) {...}
    override fun deleteItem() {...}
    @SuppressLint( ...value: "SimpleDateFormat")
    private fun log(action: String) {...}

    override fun setView(view: ContractItemFragment.ViewContract?) {
        this.viewItem = view
    }
}
```

Slika 5.8. Implementiran MVP obrazac na strani prezentera

5.2. IMPLEMENTACIJA MVVM ARHITEKTURNOG OBRASCA

MVP obrazac vrlo je jednostavan za shvatiti i implementirati ali izložen je problemima koji su navedeni u ovome radu pod potpoglavljem 5.3 („zauvijek“ vezani prezenter za pogled) i dodavanje nove funkcionalnosti vremenski je zahtjevno. Primjereniji obrazac bio bi MVVM obrazac, značajno se razlikuje od MVP obrasca po tome što: ne postoji dvosmjerna veza između pogleda i u ovom slučaju modela pogleda (eng. viewmodel) odnosno pogled ima referencu na model pogleda dok model pogleda nema na pogled. Također važno je napomenuti kako pogled u MVVM obrascu može imati više modela pogleda (veza jedan naprema više) dok kod MVP obrasca pogled može imati samo jednog prezentera (veza jedan naprema jedan). Najveća razlika između MVP i MVVM obrasca je to da pri kreiranju modela pogleda u pogledu model pogleda ima životni tijek svog vlasnika (eng. lifecycle owner) odnosno svog pogleda, tako da ne treba brinuti više o problemu gdje u MVP obrascu prezenter ima referencu na pogled koji se više ne koristi.



Slika 5.9. Razlika u strukturi između MVP i MVVM obrasca [22]

5.2.1. Podaci stvarnog vremena (eng. Live Data)

MVVM obrazac temelji se na podacima koji se ažuriraju u realnom vremenu (eng. Live Data). Razlika u odnosu na uobičajeni obrazac promatrača je ta da je povezan sa životnim vijekom (eng. lifeclycle aware). To znači da njegova svijest o životnom vijeku pogleda osigurava da se podaci ažuriraju samo kada je pogled u aktivnom stanju. Podaci koji se ažuriraju u realnom vremenu

dodatno unaprjeđuju eleganciju koda MVVM obrasca. Kako bi u kodu mogli koristiti potrebnu biblioteku, potrebno je u ovisnostima (eng. dependencies) aplikacije dodati poveznicu na izvor biblioteke.

Konkretno u ovom obrascu model pogleda sadrži podatke u stvarnom vremenu odnosno sadrži privatan podatak u stvarnom vremenu kojeg je moguće mijenjati samo u modelu pogleda, te isto tako sadrži isti podatak u stvarnom vremenu koji je javan odnosno pogled ga može dohvatiti ali ne može promijeniti njegovu vrijednost.

```
private val _toast: MutableLiveData<String> = MutableLiveData<String>()
val toast: LiveData<String>
    get() = _toast
```

Slika 5.10. Podatak stvarnog vremena unutar modela pogleda

```
mDbRef.addValueEventListener(object : ValueEventListener {
    override fun onCancelled(p0: DatabaseError) {
        _toast.value = p0.message
    }
```

Slika 5.11. Postavljanje vrijednosti podatka stvarnog vremena u modelu pogleda

Učinkovitost podatka u stvarnom vremenu primjećuje se u pogledu, gdje pogled dohvaća podatak stvarnog vremena koji je javnog pristupa iz modela pogleda i promatra ga, te na promjenu vrijednosti pogled se ažurira.

```
viewModel.toast.observe(viewLifecycleOwner, Observer { it: String!
    showToast(it)
})
```

Slika 5.12. Promatranje podatka stvarnog vremena u pogledu

5.2.2. Podatkovna veza (eng. Data Binding)

Podatkovna veza (eng. Data Binding) je pristup gdje xml datoteka ima pristup podacima koje će prikazivati, u ovom slučaju pristup podacima modela pogleda. Korištenjem podatkovne veze s podacima stvarnog vremena aplikacija postiže visoku razinu čistog koda te jednostavnosti koda.

Prednosti podatkovne veze:

- Ne postoji mogućnost pada sustava aplikacije zbog zaustavljenih aktivnosti ili fragmenata. U slučaju da se aktivnost nalazi u stogu aktivnosti (eng. back stack), aktivnost ne prima obavijesti o promjeni vrijednosti podatka stvarnog vremena.
- Pravilne promjene konfiguracije. Kada se aktivnost ponovno kreira zbog promjene konfiguracije, na primjer: rotacija uređaja, aktivnost odmah dobiva najnovije dostupne podatke.
- Nema propusta unutar alociranja memorije. Nema potrebe za ručnim kontroliranjem pretplata na podatke stvarnog vremena, promatrači se čiste kada im se životni ciklus uništi.

Kako bi započeli korištenje podatkovne veze potrebno je u datoteci „build.gradle(:app)“ uključiti podatkovnu vezu.

```
android {
    compileSdkVersion 29
    buildToolsVersion "29.0.2"
    defaultConfig {...}
    buildTypes {...}
    dataBinding {
        enabled true
    }
}
```

Slika 5.13. Build.gradle(:app) uključivanje podatkovne veze u projekt

Prvi korak implementacije podatkovne veze u projekt je xml „omotavanje“ (eng. wrapping) korijenskog elementa (eng. root element) u sloj rasporeda (eng. layout layer) potom dodavanje elementa podatak u kojem se nalazi varijabla, u ovom slučaju je tipa model pogleda.

```xml
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools">
    <data>
        <variable
            name="viewModel"
            type="com.example.zagorscak.mvvminventory.viewmodels.ItemFragmentViewModel"/>
    </data>
    <FrameLayout...>
</layout>
```

Slika 5.14. Omotavanje xml korijenskog elementa u podatkovni sloj

U MVVM obrascu, pogled može sadržavati više modela pogleda, pa tako da ako pogled sadrži više modela pogleda, oni se u xml datoteku analogno dodaju u sloj podatka kao varijabla.

```xml
<TextView
    android:id="@+id/tv_item_name"
    android:text="@{viewModel.item.name}"
```

Slika 5.15. Primjer korištenja jednosmjerne podatkovne veze

Postoje jednosmjerna i dvosmjerna podatkovna veza, razlika između njih je to što se podaci kod jednosmjerne podatkovne veze prenose iz modela pogleda u xml datoteku pogleda, dok kod dvosmjerne podatkovne veze postoji i veza kojom se podaci iz polja unosa (eng. input field) prenose iz xml datoteke u podatak stvarnog vremena u modelu pogleda.

```xml
<EditText
    android:id="@+id/et_item_name"
    android:text="@={viewModel.item.name}"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
```

Slika 5.16. Primjer korištenja dvosmjerne podatkovne veze

Kako bi definirali podatkovnu vezu, potrebno je u pogledu postaviti varijablu model pogleda i postaviti vlasnika životnog ciklusa u podatkovnu vezu.

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    val binding: FragmentItemBinding =
        DataBindingUtil.inflate(inflater, R.layout.fragment_item, container, attachToParent: false)
    viewModel = ViewModelProviders.of( fragment: this, ItemFragmentViewModelFactory(warehouseID, itemID))
        .get(ItemFragmentViewModel::class.java)
    binding.viewModel = viewModel
    binding.lifecycleOwner = this
    return binding.root
}
```

Slika 5.17. Postavljanje podatkovne veze unutar pogleda

5.2.3. Model pogleda

Implementacija MVVM obrasca kao i implementacija MVP obrasca prikazat će se na istom primjeru odnosno fragmentu objekta skladišta. Sloj modela poprilično je identičan kao i u MVP obrascu pa tako prvi korak pri implementiranju MVVM obrasca predstavlja kreiranje modela i postavljanje xml datoteke pogleda i postupak implementacije podatkovne veze. Potrebno je kreirati model pogleda i naslijediti klasu „ViewModel“. Za razliku od MVP obrasca u MVVM obrascu ne postoji nikakav ugovor između pogleda i modela pogleda. Pogled je pretplaćen na podatke unutar modela pogleda koji se ažuriraju u stvarnom vremenu. Zadani konstruktor modela pogleda je bez parametara ukoliko je potrebno da model pogleda ima parametarski konstruktor, potrebno je kreirati tvornicu za taj model pogleda.

```
class ItemFragmentViewModelFactory(private val warehouseID: String, private val itemID: String) :
    ViewModelProvider.NewInstanceFactory() {
    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        return ItemFragmentViewModel(warehouseID, itemID) as T
    }
}
```

Slika 5.18. Tvornica modela pogleda

```
class ItemFragmentViewModel(private val warehouseID: String, private val itemID: String) : ViewModel() {
    private val mDatabase : FirebaseDatabase  = FirebaseDatabase.getInstance()
    private val mDbRef : DatabaseReference  = mDatabase.reference.child( pathString: "warehouses").child(warehouseID)
    private lateinit var warehouse: Warehouse

    private val _toast: MutableLiveData<String> = MutableLiveData<String>()
    val toast: LiveData<String>
        get() = _toast
    private var _warehouses : MutableLiveData<MutableList<Warehouse>!>  = MutableLiveData(mutableListOf<Warehouse>())
    var warehouses: LiveData<MutableList<Warehouse>>
        get() {...}
        private set(value) {}
    private val _changeToWarehousesFragment: MutableLiveData<Boolean> = MutableLiveData<Boolean>()
    val changeToWarehousesFragment: LiveData<Boolean>
        get() = _changeToWarehousesFragment
    private val _item: MutableLiveData<Item> = MutableLiveData<Item>()
    var item: LiveData<Item>
        get() = _item
        set(value) {}

    init {...}
    fun getMaxItemQuantity(): Int {...}
    fun setNewQuantity(newQuantity: Int) {...}
    fun renameItem(newName: String) {...}
    fun moveItem(quantity: Int, destinationWarehouse: Warehouse) {...}
    fun deleteItem() {...}

    @SuppressLint( ...value: "SimpleDateFormat")
    private fun log(action: String){...}
}
```

Slika 5.19. Implementiran MVVM obrazac na strani modela pogleda

Za razliku od prezentera, model pogleda je kompaktibilniji s pogledom i kod je puno čitljiviji čim se koriste dodatni obrasci kao u ovom slučaju obrazac promatrač. Smisao modela pogleda je priprema podataka i obavještavanje pogleda da je došlo do promjene promatrane vrijednosti. Značajnu prednost modela pogleda nad prezenterom predstavlja sigurnost sustava aplikacije, nije potrebno voditi računa o životnom ciklusu pogleda jer korištenjem modela pogleda potrebno je i postaviti vlasnika životnog ciklusa kojeg čini pogled i time se postiže sinkronizacija osvježavanja podataka kada je aplikacija u neaktivnom stanju.

5.2.4. Pogled

Zadatak pogleda je na svaku promjenu vrijednosti promatranih podataka stvarnog vremena iz modela pogleda ažurirati podatke na zaslonu, upravljati kontrolama korisničkog sučelja te koristiti funkcionalnosti modela pogleda na događaje koje je korisnik uzrokovao.

```
class ItemFragment(private val warehouseID: String, private val itemID: String) : Fragment() {
    private lateinit var viewModel: ItemFragmentViewModel

    companion object {
        fun getInstance(warehouseID: String, itemID: String): ItemFragment {...}
    }

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        val binding: FragmentItemBinding =
            DataBindingUtil.inflate(inflater, R.layout.fragment_item, container,  attachToParent: false)
        viewModel = ViewModelProviders.of( fragment: this, ItemFragmentViewModelFactory(warehouseID, itemID))
            .get(ItemFragmentViewModel::class.java)
        binding.viewModel = viewModel
        binding.lifecycleOwner = this
        return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        viewModel.changeToWarehousesFragment.observe(viewLifecycleOwner, Observer {...})
        viewModel.toast.observe(viewLifecycleOwner, Observer {...})
        viewModel.warehouses.observe(viewLifecycleOwner, Observer {...})
        viewModel.item.observe(viewLifecycleOwner, Observer {...})
        btn_item_quantity_minus.setOnClickListener {...}
        btn_item_quantity_plus.setOnClickListener {...}
        btn_set_quantity.setOnClickListener {...}
        btn_move.setOnClickListener {...}
        btn_rename.setOnClickListener {...}
    }
    private fun clearBackStack() {...}
    private fun showToast(text: String) {...}
}
```

Slika 5.20. Implementiran MVVM obrazac na strani pogleda

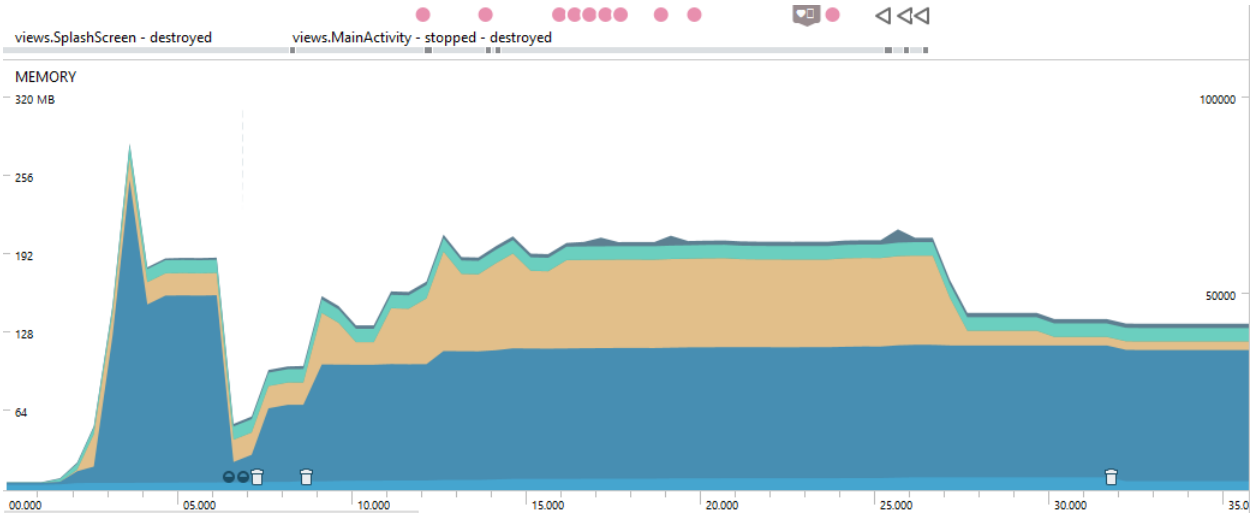
5.3. USPOREDBA

MVP obrazac je poprilično jednostavan za implementaciju te je preporučljiv za početnike, problem kod ovog obrasca jest to što pri pisanju aplikacije kreira se velik broj ugovora odnosno sučelja te za svaku novu funkcionalnost potrebno je prvo dodati u ugovor te potom implementirati novu funkcionalnost u ostale slojeve aplikacije. Kako svaki ugovor pokriva svaku korisničku interakciju u detalje, dolazi do velikog broja metoda u ugovoru. Vrlo značajan problem kod MVP obrasca predstavlja prezenterova neosvještenost o životnom ciklusu pogleda, potrebno je napisati dodatan kod samo kako nebi došlo do pada sustava aplikacije. Što se tiče prednosti, MVP ima vrlo jednostavan pogled, zadaća mu je nadzor nad korisničkim interakcijama i prosljeđivanje zadataka prezenteru koji vraća gotov rezultat koji je spreman za prikazivanje na pogledu. Obrazac je testabilan jer je cijela poslovna logika odvojena iz pogleda i nalazi se samo u prezenteru. Također prednost MVP obrasca jest to što za svaki par pogled-prezenter u ugovorima se jasno vidi funkcionalnost tog para.

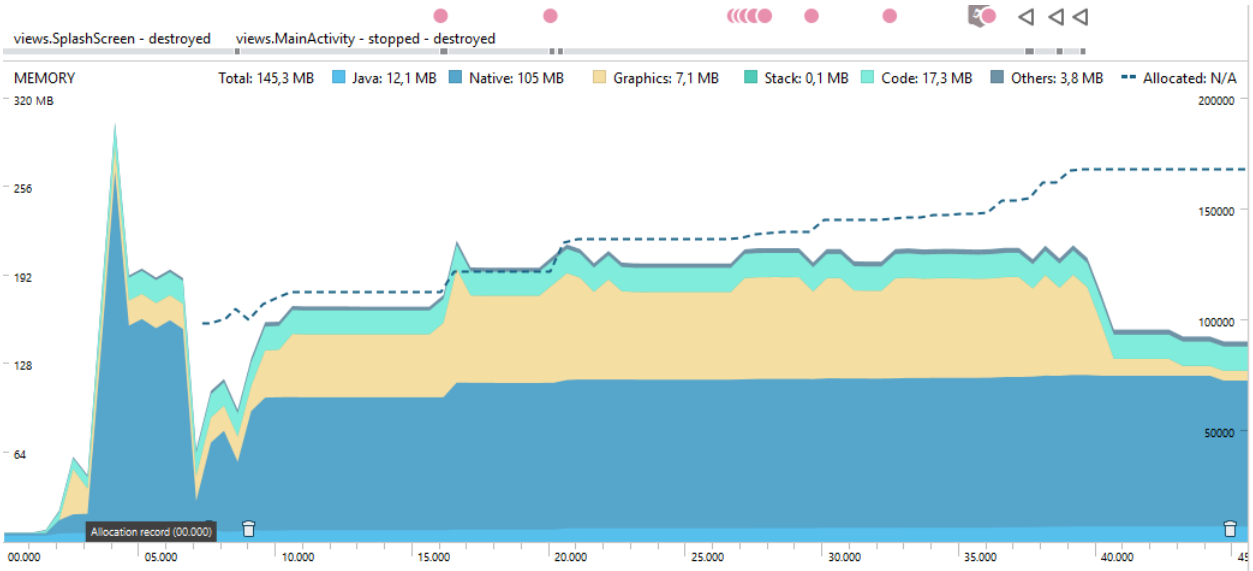
MVVM obrazac u kombinaciji s podacima stvarnog vremena i podatkovnom vezom predstavlja elegantno rješenje aplikacije, te u ovom obrascu nema problema što se tiče pada sustava aplikacije kada je pogled u neaktivnom stanju jer je model pogleda osvješten o životnom ciklusu pogleda. Zbog podataka stvarnog vremena i podatkovne veze izbjegava se kod koji se ponavlja odnosno onaj dio koda koji dohvaća i prosljeđuje podatke između modela i modela pogleda jer se temelji na obrascu promatrača koji je osviješten o životnom ciklusu. Mana MVVM obrasca se može pojaviti pri otklanjanju neispravnosti (eng. debugging) i testiranju, podatkovne veze su teže pri otklanjaju neispravnosti za razliku od koda pisanog u Javi ili Kotlinu gdje se jednostavno samo stavi točka prekida (eng. breakpoint) na liniju koda koju želimo promatrati i njenu okolinu. Unatoč težem otklanjanju neispravnosti u podatkovnoj vezi, jedinično testiranje (eng. unit testing) je izrazito jednostavnije zbog toga što model pogleda nema referencu na pogled dok MVP-ov prezenter ima. Dodavanjem nove funkcionalnosti u MVVM obrazac vremenski je manje zahtjevno i manja količina koda je potrebna za napisati, također dodavanjem novih pogleda i modela pogleda naglašava veću brzinu pisanja koda za razliku u MVP obrascu, jedino u slučaju da model pogleda zahtjeva parametarski konstruktor potrebno je napisati klasu koja predstavlja tvornicu modela pogleda.

Tablica 5.1. Numeričke razlike između MVP i MVVM obrasca

	MVP arhitekturni obrazac	MVVM arhitekturni obrazac
Broj klasa	24	27
Broj sučelja	10	2
Veličina APK datoteke	4,8 MB	5,5 MB
Veličina zauzete memorije na uređaju nakon instalacije	15,61 MB	18,32 MB



Slika 5.21. MVP korištenost memorije



Slika 5.22. MVVM korištenost memorije

Na slikama 5.21. i 5.22. nalazi se graf korištenosti radne memorije uređaja prilikom izvođenja aplikacije, x os predstavlja vrijeme [s], y os predstavlja količinu zauzete memorije [MB]. Znakovi

iznad grafa predstavljaju korisničku interkaciju s aplikacijom. Android komponente potrebne za rad aplikacije (eng. Native) najviše zauzima memorije (na grafu označeno plavom bojom), uz njega grafika (eng. Graphics) u pojedinim fragmentima zauzima veliku količinu memorije (na grafu označeno smeđom bojom), dok Java (koju čine klase) zauzima konstantu količinu memorije (na grafu označeno svijetlo plavom bojom) tijekom izvođenja aplikacije. Pri izvođenju MVVM aplikacije funkcionalan kod aplikacije zauzima više memorije nego kod MVP aplikacije (na grafovima označeno tirkiznom bojom).

6. ZAKLJUČAK

U ovom završnom radu na primjeru jednostavne aplikacije objašnjene su arhitekturni obrasci MVP i MVVM, prikazani njihovi postupci implementacija, objašnjene zadaće slojeva arhitekturnih obrazaca te u završnici se nalazi usporedba obrazaca na konkretnom projektu.

Na temelju analize koja je provedena, nije moguće konkretno definirati koji od navedenih arhitekturnih obrasaca ima dominatniju prednost u odnosu na drugi. U slučaju da je potrebno kreirati jednostavnu aplikaciju, MVVM obrazac bi bio previše za tu aplikaciju, dok bi MVP obrazac imao prednost u jednostavnim aplikacijama. Pri testiranju obrazaca MVVM je lakše testirati zbog jednosmjerne reference pogleda i modela pogleda, ali i teže zbog podatkovne veze. Dodavanje novih funkcionalnosti i novih pogleda vremenski je zahtjevnije u MVP obrascu i potrebnije više linija koda. Što se tiče preglednijeg programskog koda prednost bi imao MVVM obrazac zbog neovisnosti modela pogleda prema pogledu to jest model pogleda nema nikakvih informacija o pogledu koji će ga koristiti. Za definirani projekt i projekte gdje ima više pogleda prednost bi iskazivao MVVM obrazac, neovisno o broju ljudi u timu koji rade na projektu MVVM obrazac nudi vrlo čist kod i jednostavnost pri korištenju modela pogleda u pogledu putem podataka stvarnog vremena, također MVVM obrazac skraćuje kod koji u pogledu ažurira podatke koristeći podatkovnu vezu. Kod MVVM obrasca koja nadmašuje MVP obrazac u ovom projektu jest to što u sloju poslovne logike MVVM obrasca model pogleda je osvješten o životnom ciklusu pogleda za razliku od MVP obrasca gdje je potrebno rukovati s metodama pogleda kada dođe do ažuriranja dok je pogled u neaktivnom stanju.

Kod MVVM obrasca je veći broj klasa zbog potrebe klasa tvornica modela pogleda. Broj sučelja u MVVM obrascu značajno je manji za razliku od MVP obrasca, jer u MVVM obrascu nije potrebno definiranje ugovora između modela pogleda i pogleda. Količina zauzete memorije na uređaju nakon instalacije veća je kod MVVM obrasca zbog potrebnih klasa podatkovne veze, podataka stvarnog vremena i klasa modela pogleda. Na temelju grafova (slika 5.21. i slika 5.22.), ne primjećuje se razlika ukupno zauzete memorije uređaja tijekom izvođenja aplikacija, jedina razlika primjećuje se u zauzeću memorije funkcionalnog koda aplikacije.

LITERATURA

[1] F. Babić, „ARHITEKTURA ANDROID APLIKACIJA“, str. 43.

[2] Mediaan blog, „Kotlin vs Java“, *Mediaan*, lip. 28, 2018. https://www.mediaan.com/mediaan-blog/kotlin-vs-java (pristupljeno srp. 07, 2020).

[3] „Kotlin Programming Language“, *Kotlin*. https://kotlinlang.org/ (pristupljeno srp. 07, 2020).

[4] F. Muntenescu, „Android Architecture Patterns Part 2: Model-View-Presenter“, *Medium*, ožu. 17, 2017. https://medium.com/upday-devs/android-architecture-patterns-part-2-model-view-presenter-8a6faaae14a5 (pristupljeno srp. 07, 2020).

[5] „Guide to app architecture“, *Android Developers*. https://developer.android.com/jetpack/guide (pristupljeno srp. 07, 2020).

[6] „Android JetPack - collection of developer’s libraries“. https://www.nexmobility.com/articles/android-jetpack-library.html (pristupljeno srp. 07, 2020).

[7] A. Leiva, „MVP for Android: how to organize the presentation layer“, *Antonio Leiva*, srp. 04, 2018. https://antonioleiva.com/mvp-android/ (pristupljeno kol. 08, 2020).

[8] „Getting Started with MVP (Model View Presenter) on Android“, *raywenderlich.com*. https://www.raywenderlich.com/7026-getting-started-with-mvp-model-view-presenter-on-android (pristupljeno kol. 08, 2020).

[9] F. Cervone, „Model-View-Presenter: Android guidelines“, *Medium*, lip. 02, 2017. https://medium.com/@cervonefrancesco/model-view-presenter-android-guidelines-94970b430ddf (pristupljeno kol. 08, 2020).

[10] A. Finogenova, „Advanced Data Binding: Binding to LiveData (One- and Two-Way Binding)“, *Medium*, ruj. 13, 2019. https://proandroiddev.com/advanced-data-binding-binding-to-livedata-one-and-two-way-binding-dae1cd68530f (pristupljeno kol. 08, 2020).

[11] „What are Android Architecture Components?“ https://blog.mindorks.com/what-are-android-architecture-components (pristupljeno kol. 08, 2020).

[12] „Reference - Kotlin Programming Language“, *Kotlin*. https://kotlinlang.org/docs/reference/index.html (pristupljeno kol. 08, 2020).

[13] „Architecting Android...The clean way? | Fernando Cejas“. https://fernandocejas.com/2014/09/03/architecting-android-the-clean-way/ (pristupljeno kol. 08, 2020).

[14] S. S. Jung, „Adapter, What Responsibility is it In MVP Pattern? Data? View?“, *Medium*, ožu. 25, 2016. https://medium.com/@jsuch2362/adapter-what-role-is-it-data-view-13c713cdae0b (pristupljeno kol. 08, 2020).

[15] J. Michalik, „Refactoring RecyclerView adapter to data binding“. https://www.untitledkingdom.com/blog/refactoring-recyclerview-adapter-to-data-binding-5631f239095f-0 (pristupljeno kol. 08, 2020).

[16] „LiveData Overview“, *Android Developers*. https://developer.android.com/topic/libraries/architecture/livedata (pristupljeno kol. 08, 2020).

[17] „Understanding LiveData made simple | by Elye | Mobile App Development Publication | Medium“. https://medium.com/mobile-app-development-publication/understanding-live-data-made-simple-a820fcd7b4d0 (pristupljeno kol. 08, 2020).

[18] Elye, „Understanding Live Data made simple“, *Medium*, lip. 06, 2020. https://medium.com/mobile-app-development-publication/understanding-live-data-made-simple-a820fcd7b4d0 (pristupljeno kol. 08, 2020).

[19] „MVVM Compared To MVC and MVP“. http://geekswithblogs.net/dlussier/archive/2009/11/21/136454.aspx (pristupljeno kol. 09, 2020).

[20] A. Gao, „How to add additional parameters to ViewModel via Kotlin“, *Through the binary*, tra. 13, 2018. http://www.albertgao.xyz/2018/04/13/how-to-add-additional-parameters-to-viewmodel-via-kotlin/index.html (pristupljeno kol. 09, 2020).

[21] P. Kumar, „Understanding MVVM Architecture in Android“, *Medium*, tra. 25, 2020. https://medium.com/swlh/understanding-mvvm-architecture-in-android-aa66f7e1a70b (pristupljeno kol. 09, 2020).

[22] K. Seweryn, „MVP to MVVM transformation“, *Medium*, sij. 25, 2018. https://proandroiddev.com/mvp-to-mvvm-transformation-611959d5e0ca (pristupljeno kol. 09, 2020).

[23] „Model-View-ViewModel (MVVM) Explained“, *Wintellect*, tra. 24, 2014. https://www.wintellect.com/model-view-viewmodel-mvvm-explained/ (pristupljeno kol. 09, 2020).

[24] A. Verma, „Introduction to MVVM“, *Medium*, velj. 10, 2019. https://medium.com/mindorks/introduction-to-mvvm-836b1f3b7f61 (pristupljeno kol. 09, 2020).

[25] hongbeom, „Create Android app with MVVM pattern simply using Android Architecture Component“, *Medium*, tra. 15, 2020. https://medium.com/hongbeomi-dev/create-android-app-

with-mvvm-pattern-simply-using-android-architecture-component-529d983eaabe (pristupljeno kol. 09, 2020).

SAŽETAK

Cilj ovog završnog rada je objasniti postupak izrade aplikacije korištenjem MVP i MVVM arhitekture te usporediti neke od bitnih parametara pri razvoju Android aplikacije, također primijetiti vrline i mane MVP i MVVM arhitekture. U radu se jasno vidi postupak implementacije obje arhitekture te njihove prednosti korištenja za razliku od naivne implementacije što se tiče testabilnosti, dodavanjem novih funkcionalnosti, čitanja koda i sl. Proces implementacije arhitekture je započeo s definiranjem pogleda i njegovog izgleda u xml-u, potom kreiranje potrebnih modela i implementacijom prezentera ili pogled-modela (ovisno o kojoj arhitekturi se radi) te u konačnici završavanje pogleda (postavljanje korisničkog sučelja, postavljanje slušatelja na događaje (eng. event listeners), postavljanje promatrača na varijable model-pogleda i pozivanje potrebnih funkcija kako bi se događaj uspješno izvršio). Izrada aplikacija izvršila se u razvojnom okruženju Android Studio, a kod je pisan u programskom jeziku Kotlin. Za pohranu podataka korištena je platforma Firebase, baza podataka u stvarnom vremenu (eng. Realtime database)

Ključne riječi: Android aplikacija, baza podataka u stvarnom vremenu , Kotlin, MVP, MVVM

ABSTRACT

Comparison of MVVM and MVP architectures based on Android application for warehouse management

The goal of this thesis is to explain the process of creating an application using MVP and MVVM architecture and compare some of the important parameters in the development of Android application, also to notice strengths and weaknesses of MVP and MVVM architecture. This thesis clearly shows the process of implementing both architectures and their advantages of use, unlike naive implementation in terms of testability, adding new functionalities, reading code, etc. The process of implementing the architecture began with defining the view and its appearance in xml, then creating the necessary models and implementing the presenter or view-model (depending on which architecture we are using) and finally finishing the view (setting up the user interface, setting the event listeners, placing observers on model-view variables and calling the necessary functions to handle the event). The application was created in Android Studio, and the code was written in Kotlin programming language. The Firebase real-time database was used to store the data.

Keywords: Android application, Kotlin, MVP, MVVM, real-time database

ŽIVOTOPIS

Martin Zagoršćak rođen je 4.3.1999. u Osijeku. Pohađao osnovnu školu „Josipovac“ u Josipovcu, prigradsko naselje uz Osijek. Nakon završetka osnovnoškolskog obrazovanja, svoju znatiželju usmjerava u pravcu računarstva te upisuje Elektrotehničku i prometnu školu Osijek, smjer tehničar za računalstvo. Te potom, upisuje Fakultet elektrotehnike, računarstva i informacijskih tehnologija u Osijeku pod preddiplomskim studijem računarstva.

_____

Martin Zagoršćak

PRILOZI

- Link na GitLab repozitorij, projekt „MVP Inventory“: https://gitlab.com/zagi031/mvp-inventory
- Link na GitLab repozitorij, projekt „MVVM Inventory“: https://gitlab.com/zagi031/mvvm-inventory
- Na optičkom disku u prilogu nalaze se .docx i .pdf verzija završnog rada kao i izvorni kodovi u direktoriju „izvorni kodovi“