Aplikacija za pomoć pri plaćanju parkiranja

Kurtović, Luka

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU

FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Sveučilišni studij

APLIKACIJA ZA POMOĆ PRI PLAĆANJU PARKIRANJA

Luka Kurtović

Osijek, 2022.

SADRŽAJ

1. UVOD

U modernoj užurbanoj svakodnevici, gotovo svaki čovjek na svijetu uz sebe nosi barem jedan mobilni uređaj koji mu u većini vremena služi za automatizaciju i olakšavanje uobičajenih poslova. Jedan od tih uobičajenih poslova je i plaćanje parking karte, nešto što prosječan čovjek na dnevnoj bazi učini i više puta. Uzimajući to u obzir, kao i činjenicu da je mobilno plaćanje u velikom segmentu zamijenilo gotovi novac, može se zaključiti da su tradicionalni uređaji s kovanicama za naplatu parkinga stvar prošlosti te se gotovo uopće više ne koriste. Iz ovog razloga nastaje potreba za razvojem što jednostavnije i što korisnije aplikacije za mobilno plaćanje parkinga te je ova premisa glavni poticaj za kreiranje aplikacije *Simple parking* koja je tema ovog diplomskog rada.

Ovaj rad je podijeljen u četiri glavna poglavlja. U prvom poglavlju će se napraviti uvid u slična softverska rješenja i napravit će se usporedba s predloženim rješenjem koji je predmet ovog rada. Nakon toga će se objasniti tehnologije i programski alati koji su korišteni za razvoj aplikacije. Zatim, u sljedećem poglavlju će se opisati struktura aplikacije, odnosno raspodjela kreiranih klasa i modula po paketima. U istom poglavlju će biti detaljno objašnjena MVVM arhitektura koja je ujedno korištena i za dizajn razmatrane aplikacije. Nakon toga će biti prezentirani svi slučajevi korištenja aplikacije te će, kroz snimke zaslona, biti prikazane funkcionalnosti aplikacije s izgledima pojedinih ekrana, a na kraju rada će se dati uvid u rezultate obavljenih testiranja s pripadnim tumačenjima. U svrhu boljeg razumijevanja navedenih tumačenja, pobliže će se objasniti pojam *reverse geocoding* i način na koji ovaj algoritam funkcionira.

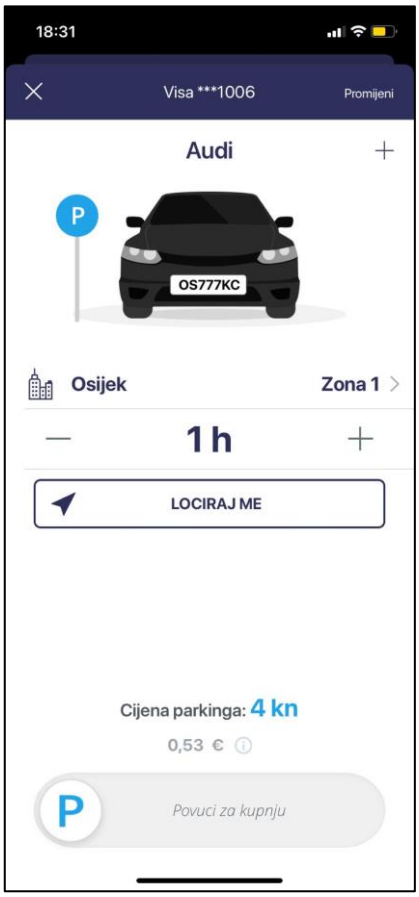
1.1. Zadatak diplomskog rada

Zadatak ovog diplomskog rada je izraditi mobilnu aplikaciju za *Android* uređaje koja će ubrzati i olakšati proces plaćanja parkirne karte. S obzirom na to da je uobičajeni način plaćanja karte slanjem SMS poruke na određeni broj relativno dugotrajan i sklon pogreškama (krivi broj, pogrešna parking zona, pogreška pri upisivanju registracije), glavni cilj ove aplikacije je automatizirati navedene procese koji su skloni pogreškama kako bi se umanjila njihova mogućnost. Glavni problem za ostvarenje ovog cilja je osmisliti način pridruživanja dobivene korisničke lokacije točnoj parkirnoj zoni.

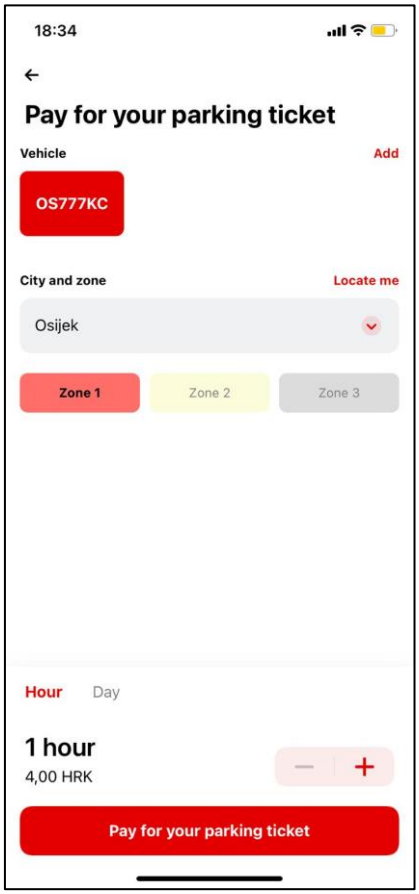
2. PREGLED PODRUČJA TEME

Aplikacija *Simple parking* povlači inspiraciju iz već postojećih programskih rješenja slične vrste poput *KEKS Pay*[1] i *Aircash*[2], iako glavna namjena ovih aplikacija nije kupovina parkirne karte nego brzi transfer novčanih sredstava s ostalim korisnicima putem mobilnih telefona.

Aplikacije *Keks Pay* i *Aircash* sadrže određene funkcionalnosti koje su srodne predloženom rješenju. Neke od tih funkcionalnosti su: mogućnost pohrane registracijske oznake, korištenje sustava geolociranja za određivanje zone, podsjetnik za istek valjanosti karte i mogućnost samostalnog odabira zone. Funkcionalnosti koje navedene aplikacije imaju, a predloženo rješenje nema su: mogućnost dodavanja više registracijskih oznaka, plaćanje parkirne karte za više sati ili čak za cijeli dan te mogućnost kartičnog plaćanja, no isto tako predložena aplikacija ima neke dodatne funkcionalnosti poput pregleda prethodnih transakcija i *timera,* koje ove aplikacije nemaju. Na slici 2.1. je prikazano korisničko sučelje aplikacije *Keks Pay,* a na slici 2.2. sučelje aplikacije *Aircash*

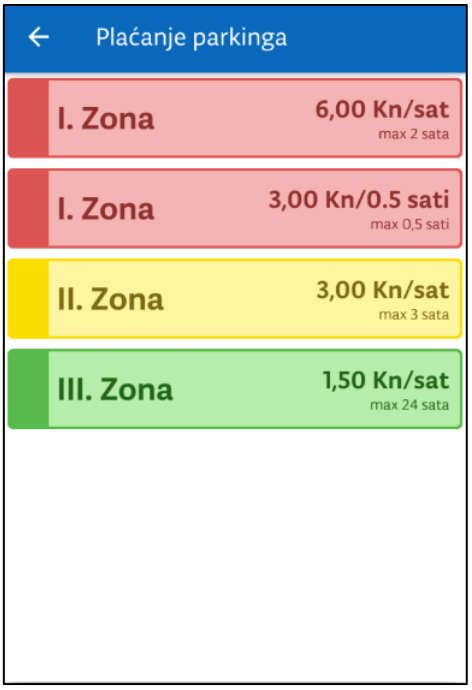


Slika 2.1. *Sučelje aplikacije Keks Pay*



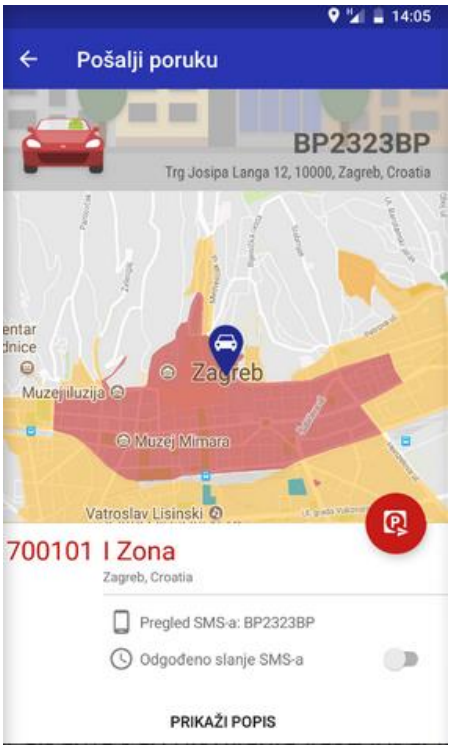
Slika 2.2. *Sučelje aplikacije Aircash*

Osim ove dvije navedene, postoje još mnoge slične aplikacije koje nude mogućnost plaćanja parkinga. Neke od njih su *ZgPark*[3], *SMS Parking*[4] i *Bmove*[5]. Aplikacija *ZgPark* je namijenjena plaćanju parkinga u gradu Zagrebu te nudi mogućnosti dodavanja više vozila, plaćanje karticom, kupovina karte za više sati i pregled prethodno kupljenih karti, a ne sadrži mogućnost slanja obavijesti o isteku valjanosti karte i automatskog lociranja korisnika. Prije korištenja aplikacije *ZgPark*, nužna je registracija korisnika. Na slici 2.3. je prikazano korisničko sučelje aplikacije *ZgPark*.



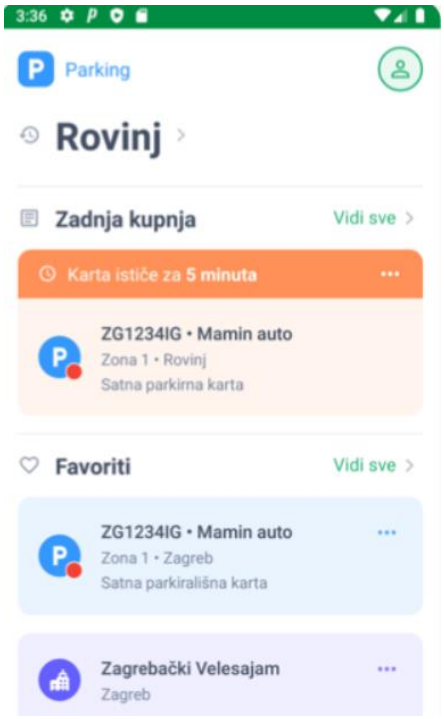
Slika 2.3. *Sučelje aplikacije ZgPark*

Aplikacija *SMS Parking* je izuzetno opsežna aplikacija s mnoštvom dodatnih funkcija i mogućnosti koje prethodno navedene aplikacije ne sadrže. Sadrži impresivno široku bazu podržanih gradova (više od 80 gradova u RH) i različitih država u Europi (više od 15). Podržava korištenje dvije SIM kartice, slanje obavijesti na pametni sat, automatski odabir zone, slanje poruke s odgodom i kartu gradova s ucrtanim zonama što dodatno poboljšava *user experience*. Slika 2.4. prikazuje korisničko sučelje navedene aplikacije.



Slika 2.4. *Korisničko sučelje aplikacije SMS Parking*

Aplikacija *Bmove* broji preko 500.000 aktivnih korisnika, a aktivna je u Hrvatskoj, Slovačkoj, Italiji i Austriji. *Bmove* je kompletan servis koji, pored standardnog plaćanja parkirne karte, nudi mogućnost implementacije sustava za nadzor parkinga unutar privatnog parkinga ili garaže. Na ovaj način vlasnik garaže može dobiti uvid u količinu prometa koja cirkulira u određenim danima i trenutnu okupaciju garažnog prostora. Sučelje je prikazano na slici 2.5.



Slika 2.5. *Sučelje aplikacije Bmove*

3. APLIKACIJA ZA POMOĆ PRI PLAĆANJU PARKINGA

U ovom poglavlju će biti definirani slučajevi korištenja aplikacije, objasnit će se arhitektura koja je korištena za izradu aplikacije, bit će navedene glavne tehnologije i alati korišteni za implementaciju te će na kraju, kroz primjere iz koda, biti opisana implementacija aplikacije.

3.1 Slučajevi korištenja aplikacije

U ovom potpoglavlju su definirani svi slučajevi korištenja aplikacije i hodogram za korisnika kroz svaki definirani ishod. Navedeno je šest različitih *use caseova.*

Tablica 3.1. prikazuje 1. slučaj korištenja. Ovim slučajem korištenja se dohvaća lokacija korisnika pritiskom na tipku „Locate me“ na početnom zaslonu. Preduvjet za izvršenje ovog *use casea* je prihvaćanje dozvole za praćenje lokacije.

Tablica 3.1. *Slučaj korištenja 1*

ID slučaja	UC1
Ime	Sustav locira korisnika
Preduvjet	Korisnik je prihvatio zahtjev aplikacije za praćenje lokacije
Glavni scenarij	1. Korisnik se nalazi na početnom zaslonu 2. Korisnik odabire tipku „Locate me“ 3. Lokacija korisnika se ispisuje na početnom zaslonu zajedno s trenutnom zonom
Alternativni scenarij	1. Korisnik nije prihvatio zahtjev za praćenje lokacije 1. Prikazuje se dijaloški okvir 2. Korisnik prihvaća zahtjev 3. Povratak na korak 3. glavnog scenarija 2. Korisnik nije prihvatio zahtjev za praćenje lokacije 1. Prikazuje se dijaloški okvir

	2. Korisnik odbija zahtjev 3. Prikazuje se „toast“ poruka

Tablica 3.2. prikazuje 2. slučaj korištenja. Ovaj *use case* je zadužen za kupovinu parkirne karte pritiskom na tipku „Buy ticket“ i potvrdom transakcije na *pop-up* dijaloškom okviru. Prije toga potrebno je prihvatiti dozvolu za slanje SMS-a.

Tablica 3.2. *Slučaj korištenja 2*

ID slučaja	UC2
Ime	Korisnik kupuje kartu
Preduvjet	Korisnik je prihvatio zahtjev aplikacije za slanje SMS-a
Glavni scenarij	1. Korisnik se nalazi na početnom zaslonu 2. Korisnik odabire tipku „Buy ticket“ 3. Prikazuje se dijaloški okvir za potvrdu transakcije 4. Nakon prihvaćanja u pozadini se šalje SMS na odgovarajući broj, pokreće se timer i prikazuje se „toast“ poruka
Alternativni scenarij	1. Korisnik nije prihvatio zahtjev za slanje SMS-a 1. Prikazuje se dijaloški okvir 2. Korisnik prihvaća zahtjev 3. Povratak na korak 3. glavnog scenarija 2. Korisnik nije prihvatio zahtjev za slanje SMS-a 1. Prikazuje se dijaloški okvir 2. Korisnik odbija zahtjev 3. Prikazuje se „toast“ poruka

Tablica 3.3. prikazuje 3. slučaj korištenja. Korisnik pritiskom na donju navigacijsku tipku „Transactions“ može pregledati sve svoje prethodno kupljene karte.

Tablica 3.3. *Slučaj korištenja 3*

ID slučaja	UC3
Ime	Pregled prethodnih transakcija
Preduvjet	Nema
Glavni scenarij	1. Korisnik se nalazi na početnom zaslonu 2. Korisnik odabire tipku „Transactions“ u donjoj navigaciji 3. Prikazuje se lista s detaljima svih prethodnih transakcija
Alternativni scenarij	Nema

Tablica 3.4. prikazuje 4. slučaj korištenja. Ovaj *use case* služi za promjenu registracijske oznake na način da korisnik pritiskom na donju navigacijsku tipku „Profile“ navigira na zaslon koji mu to omogućava.

Tablica 3.4. *Slučaj korištenja 4*

ID slučaja	UC4
Ime	Promjena registracije
Preduvjet	Nema
Glavni scenarij	1. Korisnik se nalazi na početnom zaslonu 2. Korisnik odabire tipku „Profile“ u donjoj navigaciji 3. Korisnik unosi novu registraciju 4. Korisnik odabire tipku „Save“ 5. Prikazuje se „toast“ poruka i povratak na početni zaslon

Alternativni scenarij	1. Korisnik unosi neispravnu registracijsku oznaku 1. Prikazuje se „toast“ poruka o nužnom unosu ispravne registracije

Tablica 3.5. prikazuje 5. slučaj korištenja. Ovaj *use case* je zadužen za podsjećanje korisnika o isteku karte na način da šalje obavijest u obliku notifikacije 5 minuta prije isteka.

Tablica 3.5. *Slučaj korištenja 5*

ID slučaja	UC5
Ime	Obavijest o isteku karte
Preduvjet	Korisnik je kupio kartu
Glavni scenarij	1. 55 minuta nakon kupovine karte, sustav šalje obavijest s porukom u obliku notifikacije
Alternativni scenarij	Nema

Tablica 3.6. prikazuje 6. slučaj korištenja. Pritiskom na tipku „Extend for 1 hour“ unutar notifikacije, korisnik može produljiti parkirnu kartu za dodatnih sat vremena.

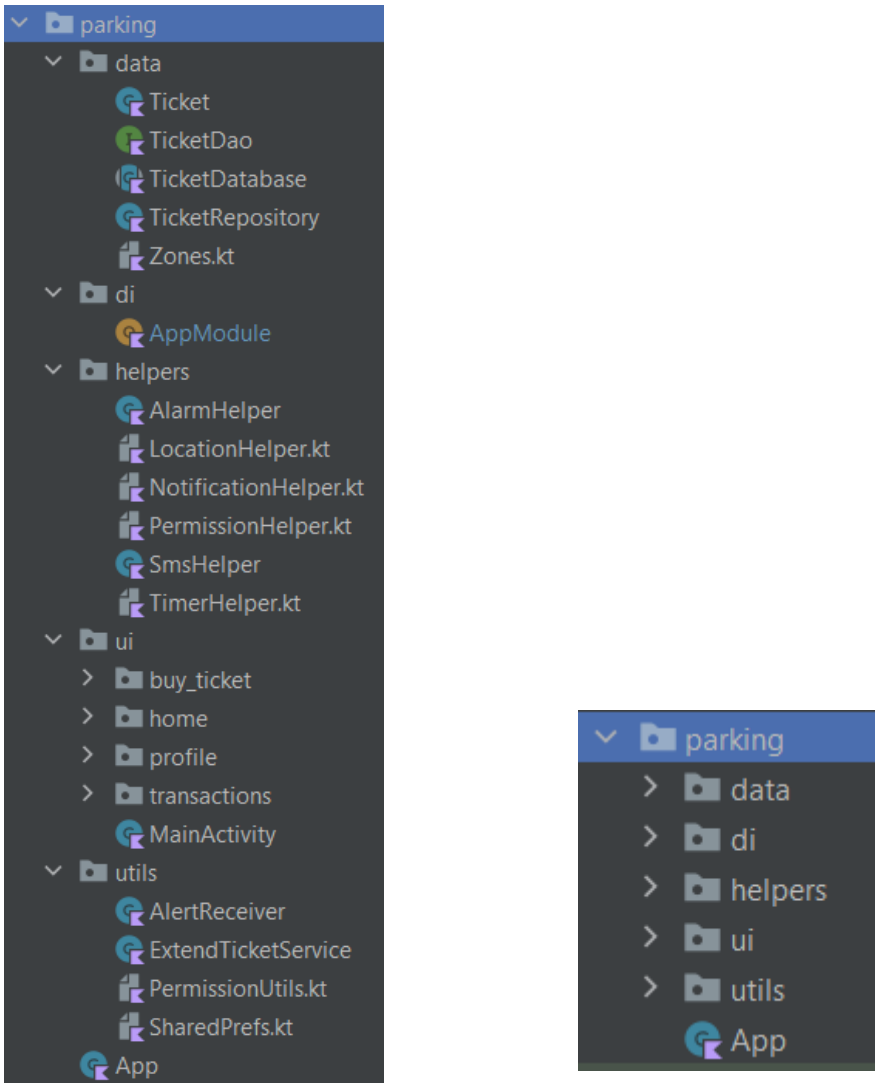
Tablica 3.6. *Slučaj korištenja 6*

ID slučaja	UC6
Ime	Produljenje karte
Preduvjet	Korisnik je primio obavijest o isteku karte
Glavni	1. Korisnik odabire tipku „Extend for 1 hour“ na notifikaciji 2. Prikazuje se „toast“ poruka

scenarij	
Alternativni scenarij	Nema

3.2. Arhitektura aplikacije

Dobra organizacija projekta je vrlo važna stavka u razvoju projekta bilo kakvih razmjera jer omogućava dobru preglednost i skalabilnost kōda te je lakše uvoditi promjene i testirati aplikaciju. Osim toga, rad u timu ne bi bio moguć bez unaprijed uvriježenih pravila pisanja kōda za projekte na kojima sudjeluje više članova. Zbog ovih razloga je potrebno strukturirati kōd unutar paketa prema logičkim cjelinama i potrebno je primijeniti odgovarajuću arhitekturu, na primjer *MVVM, MVP, MVC, MVI...*



Slika 3.1. *Struktura projekta po paketima*

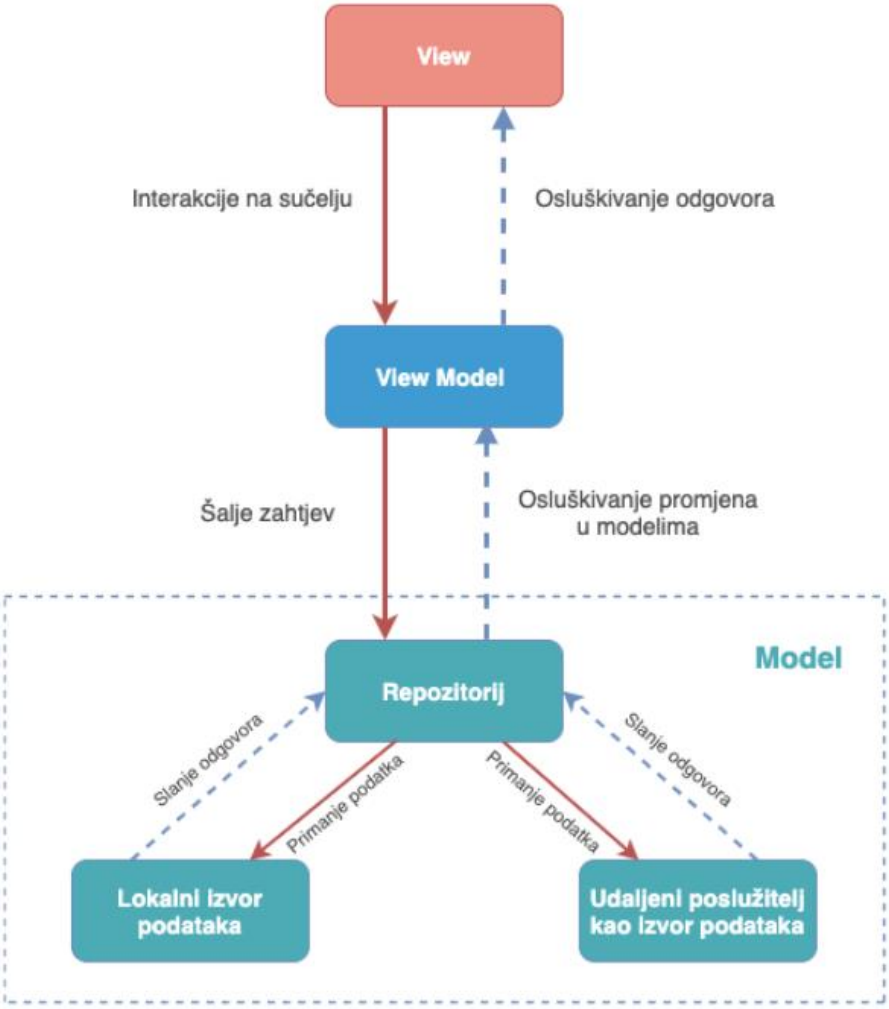
Na slici 3.1. se nalazi struktura projekta *Simple parking* koja se sastoji od 5 paketa, a „ui“ (*user interface*) paket se dodatno dijeli prema funkcionalnostima, odnosno prema pojedinom ekranu aplikacije. Za realizaciju projekta je korištena *MVVM* arhitektura.

MVVM (Model-View-ViewModel) je jedan od preporučenih strukturnih obrazaca za odvajanje sloja korisničkog sučelja (*UI layer*) od sloja koji je zadužen za obavljanje logike (*Data layer*)[6]. Ovaj obrazac se sastoji od tri ključna segmenta:

1. Model – donji sloj koji se sastoji od jednog ili više repozitorija i izvora podataka (lokalna baza i/ili vanjski poslužitelj). Ovaj sloj sadrži *business* logiku aplikacije koja rukuje podacima (stvara, sprema i mijenja podatke) te zatim te podatke nudi ostatku aplikacije, odnosno *ViewModelu.*

2. ViewModel – predstavlja srednji sloj koji služi kao komunikacija između *View* sloja i *Modela*. Uloga *ViewModela* je da očuva stanje ako dođe do uništavanja fragmenta prilikom rotacije ekrana te da na prikladan način pripremi podatke iz *Modela* za prikaz na ekranu. Podaci u *ViewModelu* se najčešće prebacuju u *LiveData* koji predstavlja *lifecycle-aware* i *observable* tip varijable, odnosno unutar *View-a* se moguće „pretplatiti“ na promjene *LiveData* varijabli koje se nalaze u *ViewModelu* te pravovremeno ažurirati prikaz tih varijabli na ekranu.
3. View – prezentacijski sloj čija je zadaća prikazati podatke na ekranu i osluškivati korisnikove interakcije (npr. *scrollanje*, pritisak tipke) te na temelju navedene interakcije proslijediti događaj *ViewModelu*. Unutar *View-a* se osluškuju promjene iz *ViewModela* te se automatski ažuriraju podaci na ekranu zahvaljujući *observable* varijablama.

Na slici 3.2. se nalazi grafički prikaz MVVM arhitekture te međusobna komunikacija i tok podataka između pojedinih slojeva. Valja napomenuti da unutar MVVM arhitekture gornji sloj drži referencu na sloj ispod sebe (npr. fragment(*View*) drži referencu na svoj *ViewModel*), dok obrnuta tvrdnja ne vrijedi. Razlog tome je što, u pravilu, *ViewModeli* „žive“ duže od fragmenata te ako bi unutar *ViewModela* držali referencu na fragment došlo bi do potencijalne pojave *memory leak* problema[7].



Slika 3.2. *MVVM arhitektura*

Razmatrana aplikacija za izvor podataka koristi lokalni bazu podataka (*Room*) te sadrži jedan repozitorij koji komunicira s bazom i ponašao se kao *single source of truth* za ostatak aplikacije. Ovaj pristup se još naziva i *repository pattern.* Dakle, sloj modela unutar aplikacije se sastoji od navedene dvije cjeline: baza podataka i repozitorij, a repozitorij je poveznica između izvora podataka i *ViewModela.*

```
class TicketRepository @Inject constructor(
    private val ticketDao: TicketDao
) {
    val allTickets = ticketDao.getAllTickets()

    suspend fun insert(ticket: Ticket) {
        ticketDao.insert(ticket)
    }
}
```

Slika 3.3. *TicketRepository klasa*

Na slici 3.3. je prikazna klasa koja predstavlja spomenuti repozitorij te se ona sastoji od jedne metode i jedne *member* varijable. U varijablu se spremaju sve prethodno kupljene karte koje repozitorij dohvaća iz baze podataka, a metoda *insert* služi za pohranu kupljene karte u istu bazu podataka.

Sloj *ViewModela* se sastoji od svih *ViewModel* klasa, a unutar ove aplikacije ih ima 3 jer se aplikacija sastoji od 3 različita fragmenta (ekrana), stoga svaki fragment ima svoj *ViewModel*. Uloga *ViewModela* je detaljnije opisana na početku poglavlja, a u navedenoj aplikaciji se svakom *ViewModelu* koji želi komunicirati s bazom mora „ubrizgati“ repozitorij kako bi imao pristup metodama za dohvaćanje i spremanje podataka u bazu. Slika 3.4. prikazuje *ViewModel* i jednu od pripadnih metoda, *onConfirmClick* koja pomoću metode iz repozitorija umeće podatke o kupljenoj karti u bazu podataka. Za ovu operaciju se koristi korutina i *CoroutineScope* jer se radi o *suspend* funkciji.

```
class HomeViewModel @Inject constructor(
    private val ticketRepository: TicketRepository,
    private val locationHelper: LocationHelper,
    private val timerHelper: TimerHelper,
    private val smsHelper: SmsHelper,
    private val prefs: SharedPrefs,
    private val alarmHelper: AlarmHelper
) : ViewModel() {
    val locationInfo = locationHelper.locationInfo.asLiveData()
    val timeLeft = timerHelper.timeLeft.asLiveData()

    fun onConfirmClick(ticket: Ticket) {
        viewModelScope.launch {  this: CoroutineScope
            ticketRepository.insert(ticket)
        }
```

Slika 3.4. *HomeViewModel klasa*

Slično kao i u prethodnom primjeru, klasa *TransactionsViewModel* također koristi repozitorij kako bi dohvatila podatke o svim prethodno kupljenim kartama koje zatim javno izlaže preko varijable *tickets* kako bi fragment mogao konzumirati te podatke (Slika 3.5.).

```
class TransactionsViewModel @Inject constructor(
    ticketRepository: TicketRepository
) : ViewModel() {
    val tickets = ticketRepository.allTickets
}
```

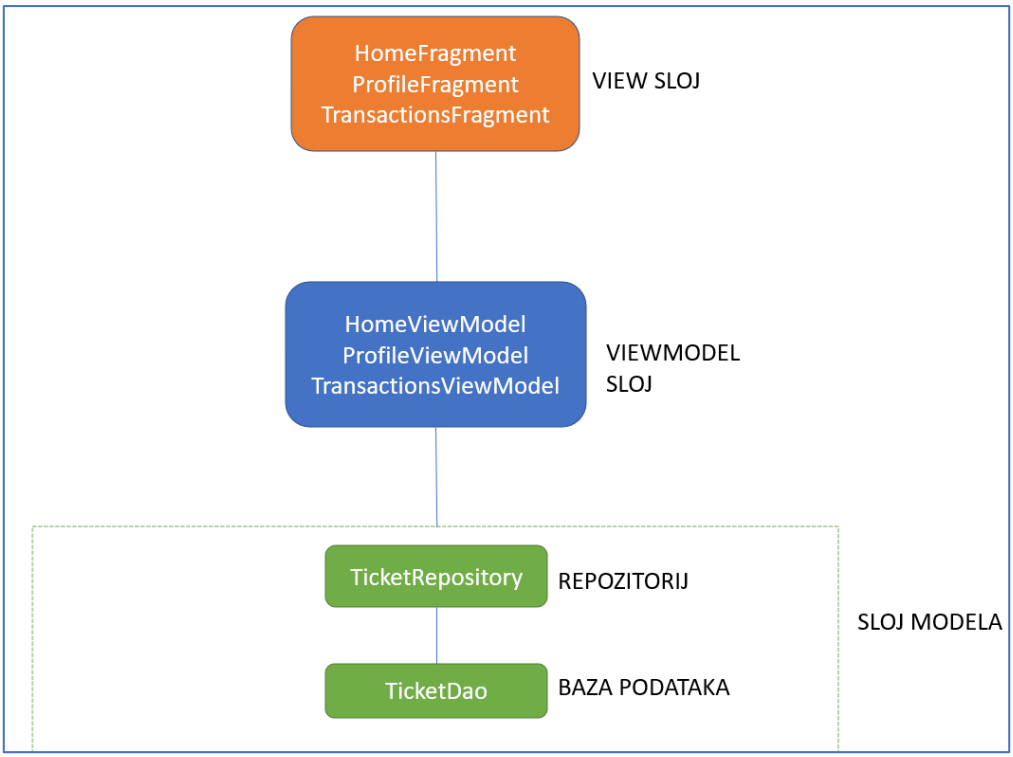
Slika 3.5. *TransactionsViewModel klasa*

View sloj aplikacije je zadužen za prikazivanje podataka, a kako bi to bilo moguće, potrebno je da se fragment „pretplati“ na promjene iz *ViewModela.* Na slici 3.4 se može uočiti varijabla *locationInfo* koja predstavlja informacije o trenutnoj lokaciji kao što su trenutna adresa i zona u kojoj se ta adresa nalazi. Ovo su podaci koje je potrebno prikazati na ekranu te je ova varijabla omotana u *live data* objekt, a time je omogućeno da se ona *observa* i fragment će reagirati na njene promjene. Kada god se dogodi promjena navedene varijable, fragment će dobiti najnovije podatke za prikaz na UI. Slika 3.6. prikazuje *observing* princip iz fragmenta na *live data* varijablu iz *ViewModela.*

```
viewModel.locationInfo.observe(viewLifecycleOwner) {  it: AddressZoneInfo!
    binding.apply {  this: FragmentHomeBinding
        "CURRENT ADDRESS: ${it.address}".also { tvAddress.text = it }
        "CURRENT ZONE: ${it.zone}".also { tvZone.text = it }
        binding.loadingIcon.isVisible = false
        btnPay.isEnabled = it.isInZone
        zone = it.zone
        when (zone) {
            "1" -> rb1.toggle()
            "2" -> rb2.toggle()
            "3" -> rb3.toggle()
            else -> radioGroup.clearCheck()
        }
    }
}
```

Slika 3.6. *HomeFragment klasa*

Slika 3.7. prikazuje MVVM arhitekturu razmatrane aplikacije s definiranim klasama za svaki pojedini sloj u koji pripadaju.



Slika 3.7. *Arhitektura aplikacije*

3.3. Obrazloženje korištenih tehnologija

U ovom potpoglavlju će biti navedene tehnologije i alati koji su korišteni za izradu razmatrane aplikacije.

3.3.1. Android Studio

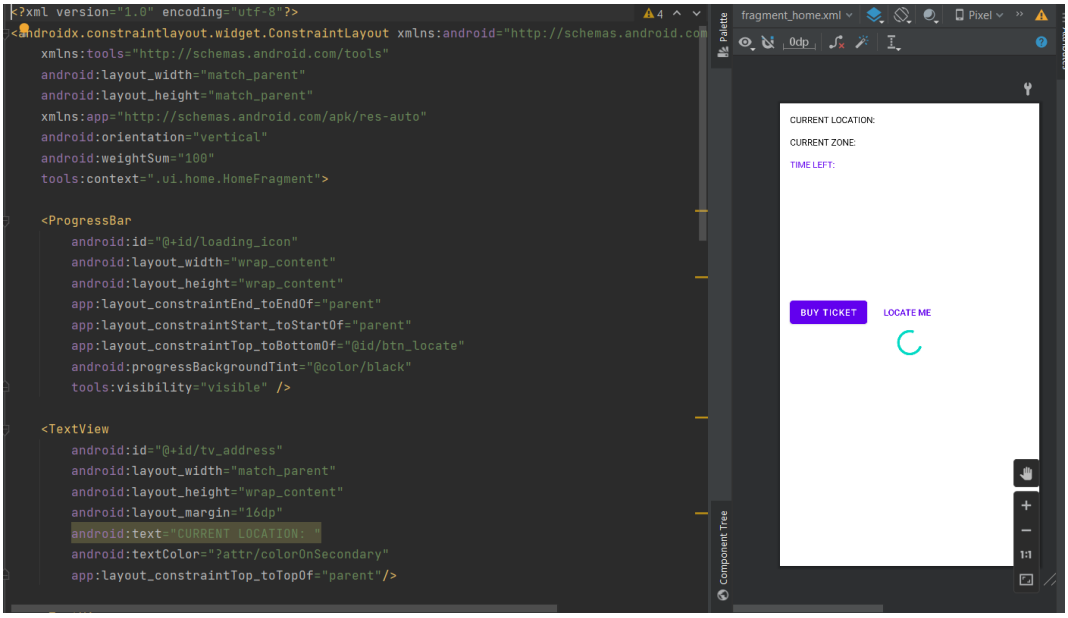
Android Studio je integrirano razvojno okruženje za kreiranje *Android* aplikacija koji je napravila češka tvrtka *JetBrains 2014. godine*, a bazira se na programu *IntelliJ IDEA* koji je dizajnirala ista tvrtka. Okruženje je izuzetno modernog sučelja s mnogobrojnim funkcionalnostima za olakšano programiranje i *debuggiranje* te je zbog toga korišteno za kreiranje razmatrane aplikacije. *Android Studio* pruža podršku za razvoj *Android* aplikacija u programskim jezicima *Kotlin* i *Java*, a za pokretanje aplikacija nije potrebno posjedovati stvarni *Android* uređaj jer okruženje nudi mogućnost testiranja na širokom spektru virtualnih mobilnih uređaja[7].

3.3.2. Kotlin

Programski jezik *Kotlin* nastao je 2011. godine te ga je također kreirala tvrtka *JetBrains*, a 2019. godine ga je *Google* proglasio kao službenim jezikom za razvoj *Android* aplikacija te je zamijenio do tada korištenu *Javu* i od tada prima punu podršku od *Google* tima[8]. *Kotlin* pruža mnoštvo poboljšanja i prednosti u odnosu na *Javu*, a jedno od najvećih prednosti je *null safety*, odnosno *Kotlin* kao programski jezik pokušava eliminirati opasnost od pristupanja *null* (nedefiniranim) varijablama i objektima što inače dovodi do urušavanja programa generiranjem *null pointer exception* iznimke[9]. Primitivni tipovi podataka (String, Int, Float, Double, Boolean...) su isključivo *non-nul,* no mogu se proglasiti kao *nullable* te je tada na programeru sva odgovornost da, ako pristupa tim varijablama, one budu definirane. Još jedna od prednosti *Kotlina* nad *Javom* je ta da *Kotlin* ima vrlo intuitivnu sintaksu za lansiranje pozadinskih niti pomoću korutina (*engl. coroutines*) koje na jednostavan način rješavaju vrlo bitan problem asinkronog (neblokirajućeg) pokretanja kōda. Korištenje pozadinskih niti je izuzetno važan koncept u programiranju jer se tako izbjegava blokiranje glavne niti koja je zadužena za prikaz korisničkog sučelja, a samim time se izbjegavaju situacije gdje aplikacija postaje neresponzivna.

3.3.3. XML

XML (*Extensible Markup Language*) je opisni jezik baziran na tekstualnom zapisu koji služi za kreiranje izgleda (*layouta*) *Android* aplikacija pomoću tagova. Svaki tag predstavlja određenu komponentu na zaslonu, a pravila za slaganje komponenti su definirana u *root layoutu* koji obuhvaća kompletni XML *layout* dokument. Neki od najčešće korištenih i najpopularnijih *root layouta* su *constraint layout* koji se temelji na fiksiranju početka, kraja, vrha i dna elemenata na druge dijelove *layouta*, *relative layout* koji postavlja elemente u odnosu na ostale elemente unutar *layouta* i *linear layout* koji je vrlo jednostavan jer slaže elemente jedan ispod drugog redom kako su definirani. Izuzetno korisna funkcionalnost unutar *Android Studia* je pregled izgleda ekrana u stvarnom vremenu, što znači da programer za vrijeme kreiranja *layouta* dodavanjem XML elemenata može gotovo trenutno imati uvid kako će se to odraziti na stvarnom uređaju na njegovom ekranu[10]. Na slici 3.8. je prikazan proces kreiranja *layouta* u *Android Studiu* pomoću XML-a.



Slika 3.8. *Kreiranje layouta u XML*

3.3.4. Android Jetpack

Android Jetpack je kolekcija biblioteka, komponenti, alata i naputaka koje pomažu programerima pratiti najbolje prakse, smanjiti količinu ponavljajućeg kōda (*engl. boilerplate*) i pisati programski kōd koji konzistentno radi na različitim *Android* uređajima i verzijama[11]. Neke od *Android Jetpack* komponenti korištene za razvoj aplikacije *Simple Parking* su: Room, ViewModel, LiveData, Navigation Component.

3.3.5. Navigation Component

Navigation Component biblioteka je jedna od *Android Jetpack* komponenti koja služi za olakšano kreiranje i dizajniranje korisničkog sučelja (*User interface*) te se najčešće koristi u kombinaciji sa *single-activity* arhitekturom, odnosno cijela aplikacija se sastoji od isključivo jednog *activityja* koji je „domaćin“ svim ostalim fragmentima unutar aplikacije[12]. Fragmenti su, za razliku od *activityja*, sistemski mnogo lakše komponente i upravljanje s više fragmenata je u pravilu mnogo jednostavnije od upravljanja s više *activityja.* Zbog toga se programere potiče da svoje aplikacije baziraju na *single-activity* te je iz tog razloga *Navigation Component* dizajniran kako bi se ostvarili benefiti korištenjem fragmenata. Osim što je navigiranje između fragmenata pomoću ove komponente značajno olakšano zbog korištenja automatski generirane *Directions* klase, *Navigation Component* nudi mogućnost *type-safe* navigacije između destinacija i prebacivanja argumenata između destinacija pomoću *Safe Args* komponente[13]. Ovo je izuzetno korisna funkcionalnost jer se različiti podaci (čak i *custom* klase) mogu prebaciti s početne na odredišnu destinaciju gdje se zatim ti podaci bez poteškoća mogu dohvatiti.

Kako bi se omogućilo korištenje *Navigation Component*, potrebno je prvo uključiti odgovarajući *dependency* unutar *Gradle* datoteke(Slika 3.9.),

```
// Kotlin
implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
```

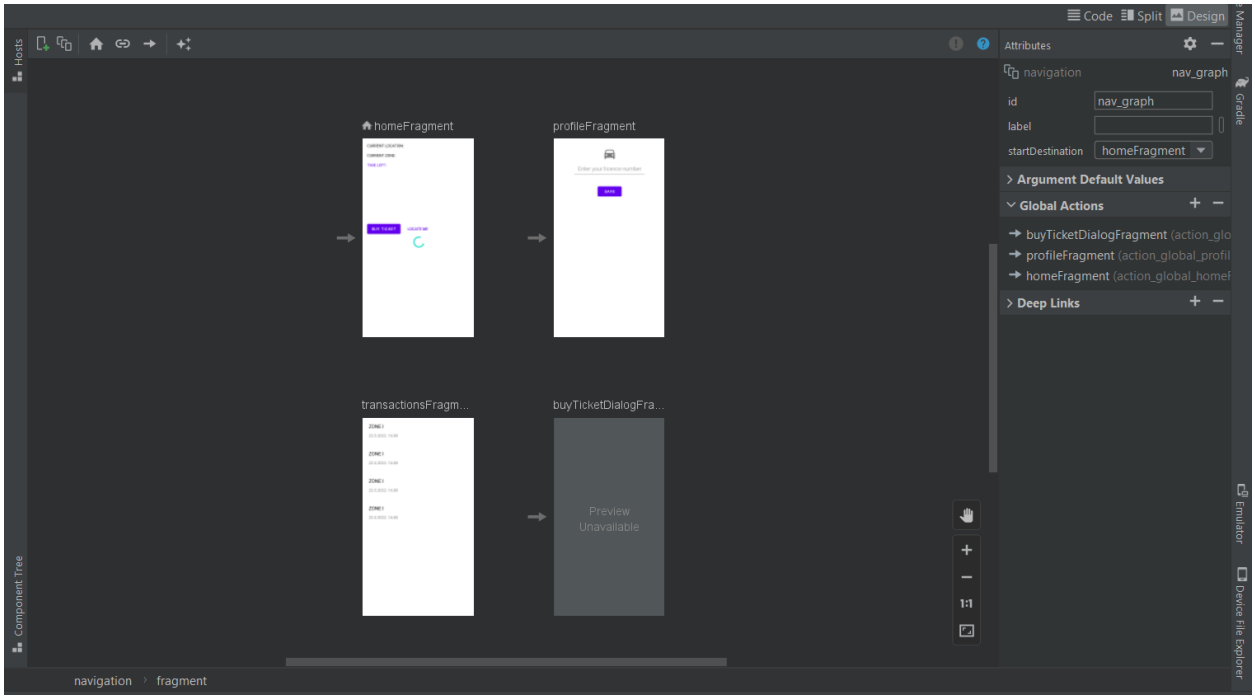
3.9. *Dependency za Navigation*

te se zatim u *activity* definira *NavHost* koji će biti zadužen za prikazivanje svih fragmenata(Slika 3.10.),

```
<androidx.fragment.app.FragmentContainerView
    android:id="@+id/nav_host_fragment"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    app:defaultNavHost="true"
    app:layout_constraintBottom_toTopOf="@id/bottom_navigation"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:navGraph="@navigation/nav_graph" />
```

Slika 3.10. *Definiranje NavHost-a*

a nakon toga je sve spremno za kreiranje *nav-grapha* s različitim destinacijama unutar *Navigation editora*(Slika 3.11.).



Slika 3.11. *nav_graph.xml*

3.3.6. Room

Room je još jedna *Android Jetpack* komponenta čije se korištenje preporučuje od strane *Googlea* kada god unutar aplikacija postoji potreba za lokalnom pohranom veće količine strukturiranih podataka (objekata) što je vrlo korisno kada korisnik nema trenutno pristup internetu, a ranije je učitao određenu količinu podataka. *Room* biblioteka predstavlja apstrakciju nad SQLite bazom podataka i omogućava korištenje funkcionalnosti koje pruža SQLite te se preferira korištenje ove biblioteke umjesto API solucija. *Room* pruža programerima mogućnost *compile-time* verifikacije SQL upita što znači da će u trenutku pisanja upita *compiler* prepoznati pogreške te će povratno upozoriti osobu koja piše programski kōd što je izuzetno snažna i korisna funkcionalnost jer se eliminiraju pogreške koje su vezane za pisanje upita. Osim toga, *Room* sadrži prikladne anotacije za generiranje različitih segmenata koji su nužni za uspješno kreiranje baze podataka što također znatno smanjuje pisanje *boilerplate* kōda. U pozadini svake anotacije su mnogobrojne linije kōda koje su unaprijed definirane te spremne da ih programer iskoristi bez ikakvih komplikacija[14].

Svaka *Room* baza podataka se sastoji od 3 ključna segmenta, odnosno 3 klase: klasa za generiranje baze podataka, DAO sučelje (*Data Access Object*) i entiteti, odnosno tablice unutar baze podataka. DAO sučelje sadrži različite metode koje se koriste za operacije nad bazom podataka, kao na primjer umetanje, brisanje i ažuriranje podataka. Ove tri navedene metode su predefinirane unutar *Room* biblioteke te ih se vrlo jednostavno uključuje pomoću već spomenutih anotacija, no osim ovih metoda, moguće je napisati proizvoljan SQL upit koji će odraditi određenu manipulaciju s bazom podataka.

Na slici 3.12. se nalazi primjer DAO sučelja s dvije metode: umetanje karte i dohvaćanje svih karata iz tablice poredano s obzirom na vrijeme kreiranja.

```
@Dao
interface TicketDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(ticket: Ticket)

    @Query("SELECT * FROM Ticket ORDER BY timestamp DESC")
    fun getAllTickets(): LiveData<List<Ticket>>

}
```

Slika 3.12. *TicketDao.kt*

Može se primijetiti da za prvu metodu nije potrebno „ručno“ pisati upit jer *Room* ima ovu metodu predefiniranu unutar svoje biblioteke te se treba samo napisati odgovarajuća anotacija, dok je za drugu metodu to potrebno jer podatke iz baze podataka možemo dohvaćati na mnogo različitih načina i prema različitim pravilima. *Room* baza podataka se koristi u navedenoj aplikaciji u svrhu spremanja informacija o prethodno kupljenim parking kartama.

3.3.7 Dependency Injection

U objektno-orijentiranom programiranju je neizostavno korištenje objekata te je vrlo čest slučaj da jedan objekt ovisi o drugom objektu ili nizu drugih objekata, stoga je potrebno kroz konstruktor klase prvog objekta umetnuti druge objekte o kojima on ovisi. Ovaj koncept se naziva *dependency injection*. Kada u aplikaciji postoje ovakvi objekti koji ovise o drugim objektima, a ti objekti također ovise o nekim drugim objektima, dolazi do problema gdje se na više mjesta pojavljuje velika količina ponavljajućeg kōda čija je isključiva funkcija instanciranje jednog objekta. Kako bi se riješio ovaj problem, na raspolaganje se nudi nekoliko različitih biblioteka za *dependency injection*, a neke od najpoznatijih su: *Hilt, Dagger* i *Koin.* Za realizaciju aplikacije *Simple parking* je korištena *Hilt* biblioteka koju je razvio *Google[15].* Pomoću ove biblioteke moguće je na jednom mjestu unutar aplikacije definirati pravila za generiranje objekata, a *Hilt* će se u pozadini pobrinuti za generiranje potrebnih *dependencyja* za vrijeme kompajliranja programa. Slika 3.13. prikazuje objekt unutar kojeg se definiraju različite *provide* funkcije pomoću *Hilt* biblioteke.

```
@Module
@InstallIn(SingletonComponent::class)
object AppModule {
    @Singleton
    @Provides
    fun provideDatabase(
        app: Application
    ) = Room.databaseBuilder(app, TicketDatabase::class.java, name: "ticket_database")
        .build()

    @Provides
    fun provideTaskDao(db: TicketDatabase) = db.getTicketDao()

    @Provides
    fun provideGeocoder(
        app: Application
    ) = Geocoder(app, Locale.getDefault())
```

Slika 3.13. *AppModule objekt za dependency injection*

3.4. Opis implementacije

Implementacija aplikacije u određenoj mjeri je spomenuta kroz prethodna poglavlja u vidu korištenja *Room* baze podataka, *navigation* komponente za navigaciju između fragmenata, prikazan je modul za *dependency injection* i komunikacija od *modela* do *view* sloja.

U ovom potpoglavlju bit će navedene i opisane najvažnije metode i klase koje su implementirane kako bi se postigao željeni rad aplikacije. Kako bi se omogućilo dohvaćanje lokacije korisnika, potrebno je postaviti *click listener* na tipku koja će okidati lociranje. Ova funkcija se nalazi unutar klase *HomeFragment* i unutar *onViewCreated* funkcije koja se poziva svaki puta kada se na zaslon pozove navedeni fragment. Slika 3.14. prikazuje postavljanje *click listenera* na tipku.

```
binding.apply {  this: FragmentHomeBinding
    btnLocate.setOnClickListener {  it: View!
        permissionHelper.withPermission(permissionHelper.locationPermission,
            { viewModel.fetchLocation() },
            { locationPermissionLauncher.launch(arrayOf(permissionHelper.locationPermission)) }
        )
        loadingIcon.isVisible = true
    }
```

Slika 3.14. *onClickListener na tipku za lociranje*

Pritiskom korisnika na tipku prvo se provjerava jesu li odobreni nužni zahtjevi za rukovanje lokacijom te, ukoliko jesu, poziva se funkcija *fetchLocation* koja se nalazi unutar *LocationHelper* klase(Slika 3.15.).

```
fun fetchLocation() {
    val criteria = Criteria()
    criteria.accuracy = Criteria.ACCURACY_FINE
    val provider = locationManager.getBestProvider(criteria,  enabledOnly: true)
    val minTime = 1000L
    val minDistance = 1.0f
    try {
        locationManager.requestLocationUpdates(
            provider!!,
            minTime,
            minDistance,
            locationListener
        )
```

Slika 3.15. *fetchLocation funkcija*

Navedena funkcija koristi *LocationManager* objekt za dohvaćanje lokacije pomoću *LocationListener* sučelja koje se koristi za okidanje svaki puta kada se registrira nova lokacija. Dobivena lokacija se zatim prosljeđuje prema *updateAddress* funkciji koja šalje lokaciju u *geocoder* kako bi se dobila adresa(Slika 3.16.). Nakon toga se dobivena adresa prosljeđuje *updateZone* funkciji koja je odgovorna za određivanje zone na način da uspoređuje lokaciju s definiranim listama zona u kojima se nalaze ulice(Slika 3.17.).

```
private fun updateAddress(location: Location) {
    val lat = location.latitude
    val long = location.longitude

    val addressList = geocoder.getFromLocation(lat, long,  maxResults: 1)
    address = if (addressList.size > 0) {
        addressList[0].getAddressLine( index: 0)
    } else "Address couldn't be found,"
    updateZone(address)
    locationManager.removeUpdates(locationListener)
}
```

Slika 3.16. *updateAddress funkcija*

```
private fun updateZone(address: String) {
    val streetNameWithNumber = address.split( ...delimiters: ",")[0]
    val streetName = streetNameWithNumber.substring(0, streetNameWithNumber.lastIndexOf( string: " "))

    zone = when {
        firstZone.contains(streetName) -> "1"
        secondZone.contains(streetName) -> "2"
        thirdZone.contains(streetName) -> "3"
        else -> "none"
    }
    isLocationInZone()
}
```

Slika 3.17. *updateZone funkcija*

Ukoliko se dobivena lokacija nalazi unutar bilo koje zone, tipka „Buy ticket“ na početnom zaslonu će postati klikabilna, a u suprotnom će biti onemogućena.

Pritiskom na tipku „Buy ticket“ i potvrđivanjem transakcije na dijaloškom okviru, okidaju se funkcije *onConfirmClick*(Slika 3.18.) i *sendSMS*(Slika 3.19.).

```
fun onConfirmClick(ticket: Ticket) {
    viewModelScope.launch {  this: CoroutineScope
        ticketRepository.insert(ticket)
    }
    alarmHelper.setAlarm(System.currentTimeMillis() + 3300000)
    prefs.addTimeToFinish( time: System.currentTimeMillis() + 3600000)
    startTimer()

}
```

Slika 3.18. *onConfirmClick funkcija*

Unutar *onConfirmClick* funkcije poziva se metoda iz repozitorija unutar korutine koja je zadužena za spremanje podataka o karti u bazu podataka. Osim toga, ova funcija zakazuje slanje obavijesti na uređaj nakon 55 minuta i pokreće *timer* u trajanju od 60 minuta koji se također prikazuje i na početnom zaslonu.

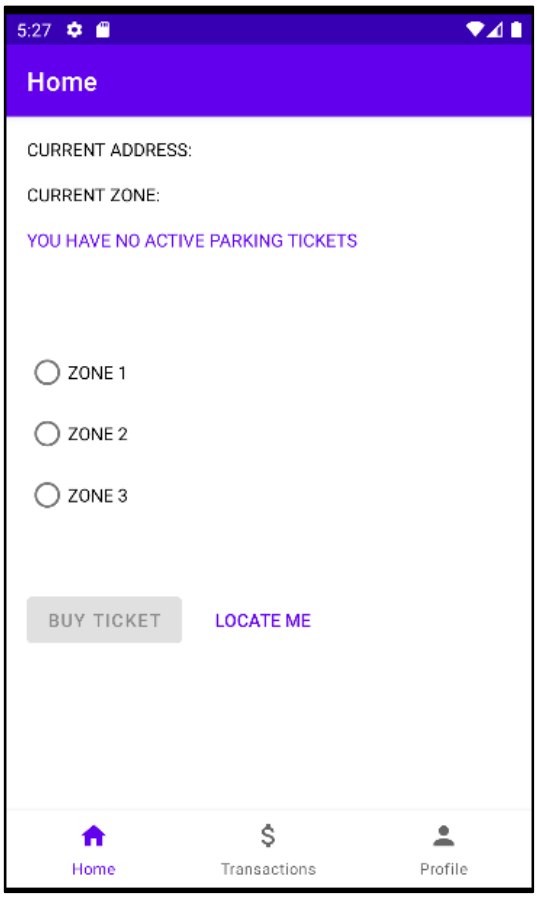
```
fun sendSMS(zone: String) {
    val licenceNumber = prefs.getLicence()
    val destinationNumber = when (zone) {
        "1" -> "708311"
        "2" -> "708312"
        "3" -> "708313"
        else -> "0"
    }
    smsManager.sendTextMessage(destinationNumber,  scAddress: null, licenceNumber,
}
```

Slika 3.19. *sendSMS funkcija*

Funkcija *sendSMS* koristi *SmsManager* klasu za upravljanje operacijama poput slanja tekstualnih poruka. Metoda *sendTextMessage* kao parametre prima odredišni telefonski broj i sadržaj poruke. Za potrebe aplikacije, sadržaj poruke je registracijska oznaka automobila koja se dohvaća iz *prefs* pomoćne klase, a odredišni broj ovisi o zoni za koju se kupuje parkirna karta.

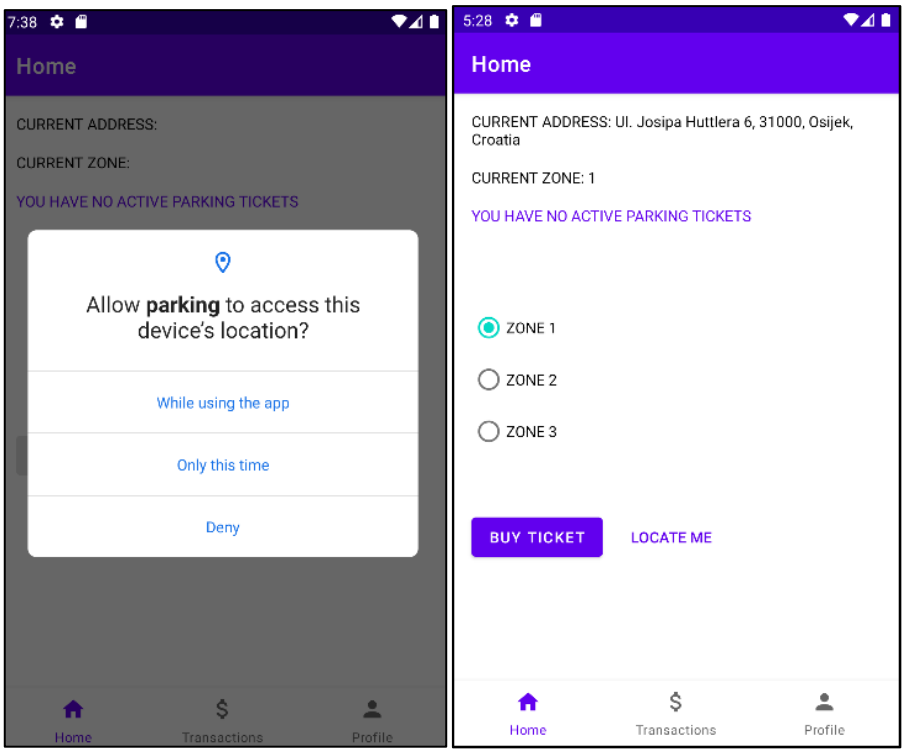
4. DEMONSTRIRANJE I ISPITIVANJE FUNKCIONALNOSTI

Pokretanjem aplikacije otvara se početni zaslon prikazan na slici 4.1. Na početnom zaslonu se nalaze informacije o trenutnoj adresi, trenutnoj zoni, preostalo vrijeme (ukoliko je kupljena karta) i dvije tipke za lociranje i kupovinu karte.



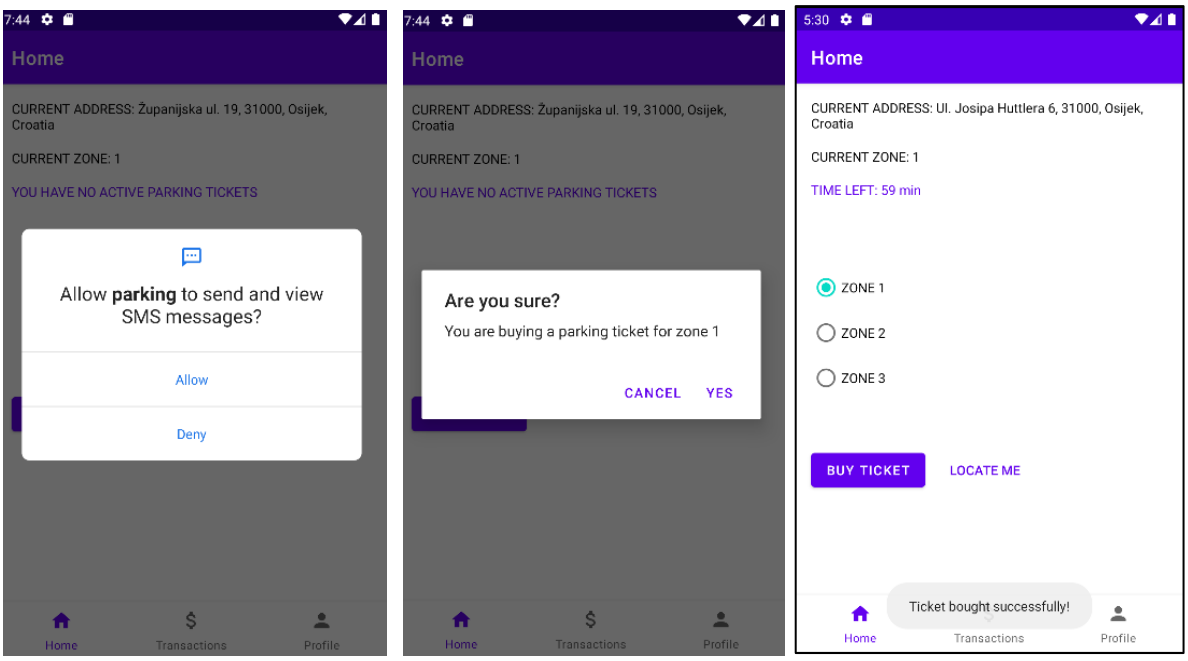
Slika 4.1. *Početni zaslon*

Pritiskom na tipku „Locate me“, sustav traži dopuštenje za praćenje lokacije te se zatim prikazuje lokacija korisnika(Slika 4.2.) i odgovarajuća zona (ako je adresa unutar zone). Dijaloški okvir s traženjem dopuštenja se prikazuje samo prvi put prije nego što ih korisnik prihvati.



Slika 4.2. *Dohvaćanje lokacije*

Nakon toga, korisnik može kupiti kartu (ako se nalazi unutar određene zone) pritiskom na tipku „Buy ticket“ koja također zahtjeva prethodno dopuštenje za slanje SMS-a jer se za kupovinu karte koristi klasa *SmsManager* koja u pozadini šalje SMS na unaprijed definirani broj. Kupnjom karte sustav pokreće *timer* od sat vremena. Isto tako, korisnik može i sam odabrati zonu za koju želi kupiti kartu pritiskom na jednu od ponuđenih zona. Nakon potvrde, prikazuje se *toast* poruka o uspješnoj kupovini karte. Postupak kupovine karte prikazan je na slici 4.3.

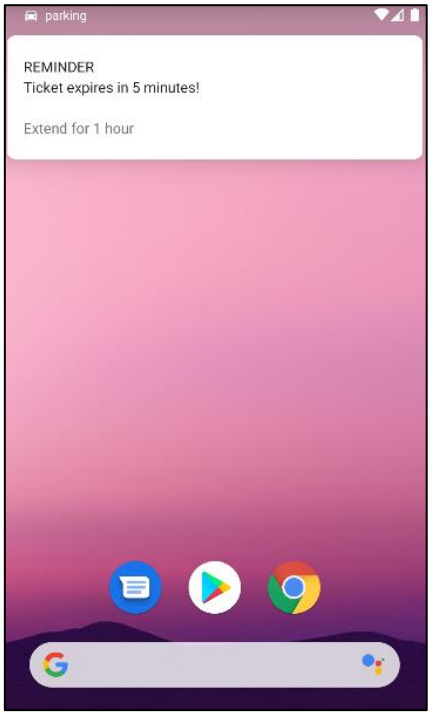


Slika 4.3. *Kupovina karte za parking*

Svaka kupljena karta se može vidjeti u „Transactions“ fragmentu(Slika 4.4.).

Slika 4.4. *Pregled transakcija*

Pet minuta prije isteka valjanosti parking karte, sustav korisniku šalje notifikaciju koja nudi mogućnost produljenja dodatnih sat vremena ako se pritisne tipka „Extend for 1 hour“ (Slika 4.5.).



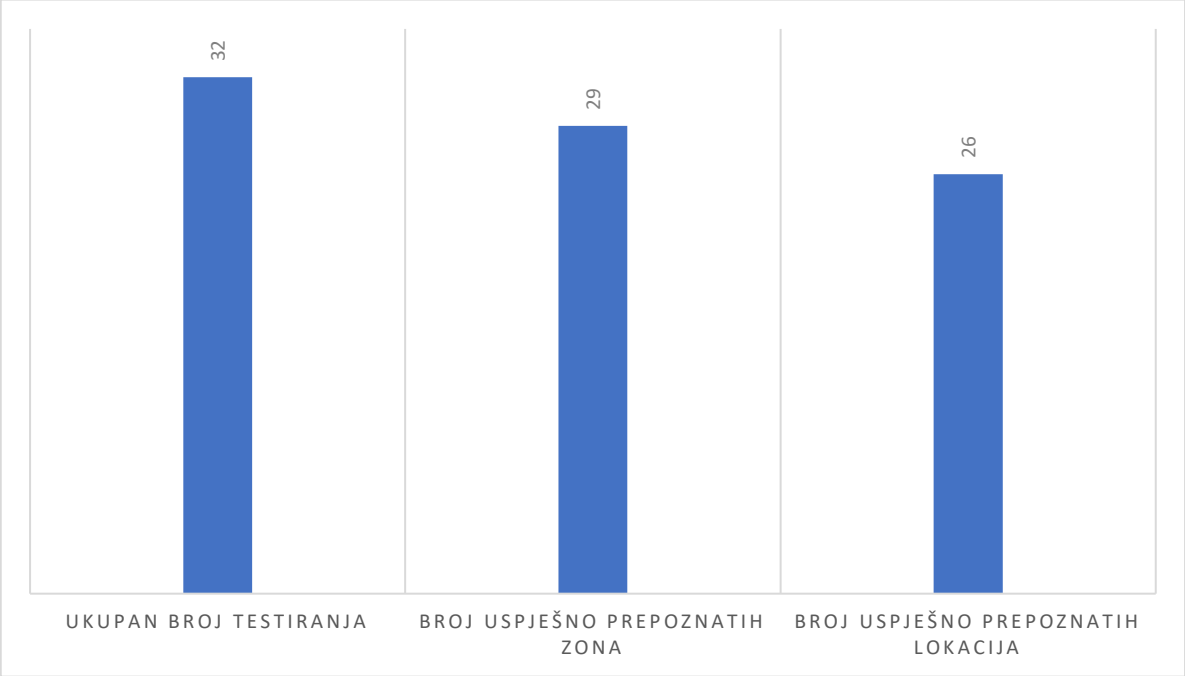
Slika 4.5. *Obavijest o isteku karte*

Kako bi se provjerila ispravnost rada aplikacije, napravljen je niz testiranja na različitim lokacijama unutar grada Osijeka i to tako što su rađeni testovi u normalnim okolnostima (sredina ulice) i u otežanim okolnostima (rubovi ulice i blizina križanja s drugim ulicama) gdje se mogu očekivati neispravna očitavanja. Napravljena su 32 različita testiranja te su rezultati prikazani u sljedećoj tablici 4.1.

Tablica 4.1. *Rezultati testiranja*

TEST #	LOKACIJA PARKINGA	ZONA	PREPOZNATA LOKACIJA	PREPOZNATA ZONA	USPJEŠNOST (ZONA)	USPJEŠNOST (LOKACIJA)
1	Istarska	2	Istarska	2	DA	DA
2	Kralja Zvonimira	2	Istarska	2	DA	NE
3	Istarska ulica	2	Kamila Firingera	0	NE	NE
4	Ulica kralja Zvonimira	2	Kralja Zvonimira	2	DA	DA
5	Kralja Zvonimira	2	Dobriše Cesarića	2	DA	NE
6	Otokara Keršovanija	2	Otokara Keršovanija	2	DA	DA
7	Kralja Zvonimira	2	Zagrebačka	2	DA	NE
8	Zagrebačka	2	Zagrebačka	2	DA	DA
9	Zagrebačka	2	Zagrebačka	2	DA	DA
10	Zagrebačka	2	Zagrebačka	2	DA	DA
11	Reisnerova	1	Reisnerova	1	DA	DA
12	Reisnerova	1	Reisnerova	1	DA	DA
13	Reisnerova	1	Radićeva	1	DA	NE
14	Zrinjevac	1	Zrinjevac	1	DA	DA
15	Zrinjevac	1	Zrinjevac	1	DA	DA
16	Županijska	1	Županijska	2	NE	DA
17	Gundulićeva	2	Gundulićeva	2	DA	DA
18	Gundulićeva	2	Gundulićeva	2	DA	DA
19	Stepinčeva	1	Stepinčeva	1	DA	DA
20	Stepinčeva	1	Stepinčeva	1	DA	DA
21	Stepinčeva	1	Vukovarska	0	NE	NE
22	Franje Šepera	2	Franje Šepera	2	DA	DA
23	Franje Šepera	2	Franje Šepera	2	DA	DA
24	Vjekoslava Hengla	1	Vjekoslava Hengla	1	DA	DA
25	Vjekoslava Hengla	1	Vjekoslava Hengla	1	DA	DA
26	Petra Preradovića	1	Petra Preradovića	1	DA	DA
27	Petra Preradovića	1	Petra Preradovića	1	DA	DA
28	Petra Preradovića	1	Petra Preradovića	1	DA	DA
29	Trg bana Jelačića	2	Trg bana Jelačića	2	DA	DA
30	Jove Jovanovića Zmaja	2	Jove Jovanovića Zmaja	2	DA	DA
31	Jove Jovanovića Zmaja	2	Jove Jovanovića Zmaja	2	DA	DA
32	Trg bana Jelačića	2	Trg bana Jelačića	2	DA	DA

Na slici 4.6. su grafički prikazani rezultati i uspješnost provedbe testiranja.

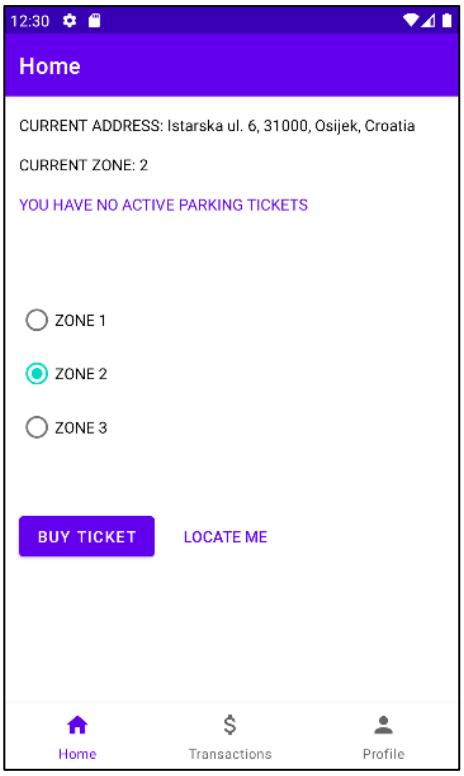


Slika 4.6. *Rezultati testiranja*

Za ispravnu interpretaciju dobivenih rezultata potrebno je pobliže objasniti algoritam koji je korišten za dobivanje adrese na temelju lokacije mobilnog uređaja. ***Reverse geocoding*** je postupak pri kojem se lokacija na mapi prevodi u ljudima čitljivu adresu[16], a za ovu aplikaciju je implementiran *Google-ov reverse geocoding* API koji je dostupan za slobodno korištenje. Za rad API-ja potrebno je proslijediti geografske koordinate lokacije, a API će na temelju koordinata pokušati pronaći najbližu adresnu oznaku zadanim koordinatama te će kao rezultat vratiti navedenu adresu. Potrebno je naglasiti da *reverse geocoding* nije egzaktna znanost te da će *geocoder* pokušati vratiti najbližu adresabilnu lokaciju unutar određene tolerancije na pogrešku[20] što znači da ako se u *geocoder* proslijedi koordinata za neku lokaciju u čijoj blizini nema građevinskih objekata s adresom, vrlo je velika vjerojatnost da će algoritam kao rezultat dati adresu koja ne odgovara navedenoj lokaciji. Na temelju ovih razmatranja se može zaključiti da će za određene lokacije *geocoder* vratiti najbližu adresu koja se može nalaziti čak i u susjednoj ulici. Prethodni zaključak daje objašnjenje za krivo očitavanje određenih lokacija tijekom testiranja aplikacije, odnosno za ispravno očitavanje većine lokacija. Naime, u ispravno prepoznatim testnim slučajevima, mobilni uređaj se nalazio ili u sredini ulice ili na rubnim

lokacijama blizu druge ulice, ali u čijoj blizini se nalazio objekt s adresom unutar promatrane ulice, dok se u krivo očitanim lokacijama uređaj nalazio na križanjima dviju ulica, a da je pri tome najbliži objekt smješten u susjednoj ulici prema *Google-ovim* kartama. U većini slučajeva gdje je ulica krivo prepoznata, zona je svejedno ispravna što je s korisničke strane pozitivan ishod.

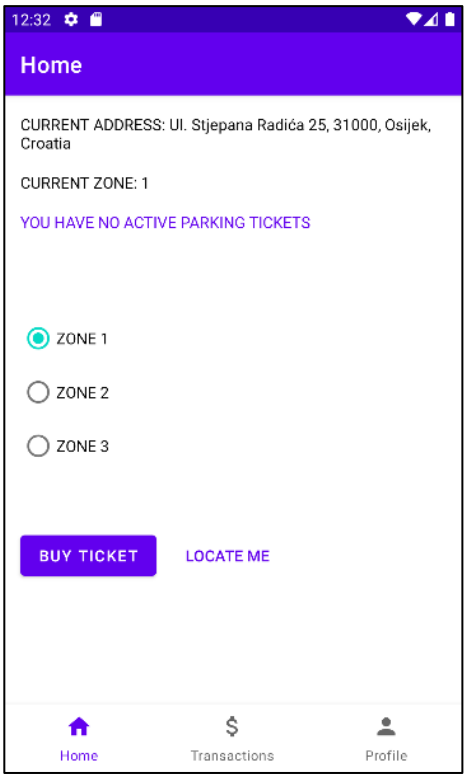
Slika 4.6. *Lokacija 1. testa* [17]



Slika 4.7. *Prepoznata lokacija 1. testa*

Na slikama 4.6. i 4.7. je prikazano ispravno prepoznavanje ulice iako se lokacija uređaja nalazi u blizini križanja s drugom ulicom, no stambena zgrada na adresi Istarska 6 je najbliža adresa za traženu lokaciju.

Slika 4.8. *Lokacija 13. testa* [21]



Slika 4.9. *Prepoznata lokacija 13. testa*

Na slikama 4.8. i 4.9. je prikazana tipična situacija u kojoj se navedeni sustav ne ponaša očekivano i u takvim situacijama se može očekivati pogreška u prepoznavanju ulice. Testni uzorak je napravljen na rubnoj lokaciji parkinga u Reisnerovoj ulici na križanju s ulicom Stjepana Radića pri čemu je najbliži objekt s adresom smješten u Radićevoj ulici (Učilište Studium) te *geocoder* zbog svog algoritma dohvaćanja lokacije smješta korisnika na neispravnu lokaciju. Treba naglasiti da, gledajući iz perspektive korisnika, ova situacija će imati pozitivan ishod jer se obje ulice nalaze u istoj parkirnoj zoni, stoga iako je korisnik lociran u krivoj ulici, karta koju će kupiti je ispravna. Za razliku od ove situacije, testiranje 16. iz tablice 4.1. je rezultiralo krivom zonom zbog toga što je Županijska ulica specifična po tome što se nalazi u dvije zone, a sustav nema mogućnost korekcije ovog problema jer je pretpostavka da se svaka ulica nalazi u samo jednoj zoni. Zbog toga je korisniku omogućeno da sam odabere ispravnu zonu u situaciji kada dođe do krivog prepoznavanja zone.

5. ZAKLJUČAK

Cilj ovog diplomskog rada bio je izraditi *Android* aplikaciju koja će korisnicima olakšati i ubrzati proces plaćanja parkirne karte i uz to omogućiti dodatne funkcionalnosti poput pregleda prethodnih transakcija i slanje podsjetnika o isteku karte kako bi se izbjeglo plaćanje kazne. Tehnologije za razvoj mobilnih aplikacija su u konstantnoj promjeni i evoluiraju na gotovo dnevnoj bazi, stoga je potrebno biti u korak s najboljim praksama za razvoj aplikacija. Zbog ovog razloga je postavljen još jedan cilj, a to je koristiti najnovije i najbolje preporuke od strane *Google* tima poput primjene adekvatne arhitekture, strukturiranje kōda, korištenje *Jetpack* biblioteka i općenito primjene dobre programerske prakse.

Aplikacija u trenutnom stanju nalazi primjenu na području grada Osijeka, no uz minimalna proširenja i dodavanje zona, lako se može prilagoditi za upotrebu i u drugim gradovima. Aplikacija za determinaciju parking zone koristi adresu, dok se adresa dohvaća preko geografskih koordinata koristeći *reverse geocoder* čija je funkcionalnost opisana u prethodnom poglavlju te je jasno da ovakav pristup nalazi određene nedostatke u specifičnim situacijama i uvjetima. Zbog tih nedostataka postoji mjesto za napredak i poboljšanje u vidu korištenja drugačije metode za dohvaćanje adrese ili sofisticiranijeg API-ja, no u ovom slučaju korišteni su besplatni alati koji su dostupni od strane *Google-a*.

Iako postoje određeni problemi, kroz testiranje je jasno pokazano da se aplikacija u velikoj većini situacija ponaša u skladu s očekivanjima te apsolutno pronalazi primjenu u realnim situacijama gdje pomaže korisnicima i ispunjava cilj koji je zadan na početku izrade ovog rješenja.

LITERATURA

[1] KEKS Pay, Erste banka d.d., dostupno na: https://www.kekspay.hr/ [kolovoz 2022.]

[2] Aircash, Aircash d.o.o., dostupno na: https://aircash.eu/ [kolovoz 2022.]

[3] ZgPark, Zagrebparking d.o.o., dostupno na: https://www.zagrebparking.hr/ [kolovoz 2022.]

[4] SMSParking, NACHEV development, dostupno na: https://smsparking.app/hr/ [kolovoz 2022.]

[5] Bmove, Bmove d.o.o., dostupno na: https://www.bmove.com/ [kolovoz 2022.]

[6] Guide to app architecture, Google LLC, dosupno na: https://developer.android.com/topic/architecture [lipanj 2022.]

[7] Meet Android Studio, Google LLC, dostupno na: https://developer.android.com/studio/intro [lipanj 2022.]

[8] Kotlin: A modern programming language that makes developers happier, JetBrains, dostupno na: https://kotlinlang.org/ [lipanj 2022.]

[9] M.Heller, What is Kotlin? The Java alternative explained, InfoWorld, 2020., dostupno na: https://www.infoworld.com/article/3224868/what-is-kotlin-the-java-alternative-explained.html [lipanj 2022.]

[10] L.R.E. Quin, XML Essentials, W3C, 2016., dostupno na: https://www.w3.org/standards/xml/core [lipanj 2022.]

[11] Android Jetpack, Google, dostupno na: https://developer.android.com/jetpack [lipanj 2022.]

[12] Get started with the Navigation component, Google LLC, dostupno na: https://developer.android.com/guide/navigation/navigation-getting-started [lipanj 2022.]

[13] Pass data between destinations, Google LLC, dostupno na: https://developer.android.com/guide/navigation/navigation-pass-data [lipanj 2022.]

[14] Save data in a local database using Room, Google LLC, dostupno na: https://developer.android.com/training/data-storage/room [lipanj 2022.]

[15] Dependency injection with Hilt, Google LLC, dostupno na: https://developer.android.com/training/dependency-injection/hilt-android, [lipanj 2022.]

[16] Geocoding Service, Google LLC, dostupno na: https://developers.google.com/maps/documentation/javascript/geocoding#ReverseGeocoding [kolovoz 2022.]

[17] Google Maps, Google LLC, dostupno na: https://www.google.com/maps/ [kolovoz 2022.]

SAŽETAK

Ovaj diplomski rad obrađuje temu izrade *Android* mobilne aplikacije koja pomaže pri plaćanju parkirne karte. Glavni problem bio je osmisliti način na koji će se pridružiti odgovarajuća parkirna zona trenutnoj lokaciji korisnika, a to je riješeno tako da je u aplikaciji korišten *Google reverse geocoder* API koji pretvara koordinate u ljudski čitljivu adresu. Nakon toga radi se usporedba dobivene adrese s adresama iz različitih nizova ulica, gdje svaki niz predstavlja jednu zonu. Funkcionalnost aplikacije je testirana, objašnjeni su postignuti rezultati te je zaključeno da aplikacija nalazi primjenu u većini situacija i zadovoljava zadanim zahtjevima.

Ključne riječi: *Android,* aplikacija, *geocoder,* lokacija, parking

ABSTRACT

Title: Application for parking payment assistance

This thesis discusses the topic of developing an Android mobile application that helps with buying a parking ticket. The main problem was to find a way to assign a proper parking zone to a current user's location and this was handled in a way that application uses Google reverse geocoder API that transforms coordinates into a human-readable address. The address that is returned is than compared with the addresses from different arrays of streets, where every array represents a single parking zone. Functionality of the application is tested, achieved results are than explained and it was concluded that the application finds usage in majority of situations and it satisfies the given requirements.

Keywords: Android, application, geocoder, location, parking

ŽIVOTOPIS

Luka Kurtović rođen je 18. kolovoza 1998. u Osijeku gdje završava Osnovnu školu Jagode Truhelke. Nakon završetka srednjoškolskog obrazovanja u III. gimnaziji Osijek, 2017. godine upisuje Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek, smjer elektrotehnika.