Mobilna aplikacija za potporu u praćenju dijabetesa

Lušičić, Katarina

SVEUČILIŠTE JURJA DOBRILE U PULI

Fakultet informatike u Puli

Katarina Lušičić

MOBILNA APLIKACIJA ZA POTPORU U PRAĆENJU DIJABETESA

Diplomski rad

Pula, 2022.

SVEUČILIŠTE JURJA DOBRILE U PULI

Fakultet informatike u Puli

Katarina Lušičić

MOBILNA APLIKACIJA ZA POTPORU U PRAĆENJU DIJABETESA

Diplomski rad

JMBAG: -, redoviti student

Studijski smjer: Informatika

Kolegij: Mobilne aplikacije

Znanstveno područje: Društvene znanosti

Znanstveno polje: Informacijske i komunikacijske znanosti

Znanstvena grana: Informacijski sustavi i informatologija

Mentor: doc.dr.sc Siniša Sovilj

Komentor: prof.dr.sc. Ratko Magjarević

Pula, rujan 2022.

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani Katarina Lušičić, kandidat za magistra informatike ovime izjavljujem da je ovaj Diplomski rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Diplomskog rada nije napisan na nedozvoljeni način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

_____

U Puli, rujan 2022.

IZJAVA O KORIŠTENJU AUTORSKOG DJELA

Ja, Katarina Lušičić dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj diplomski rad pod nazivom „Mobilna aplikacija za potporu u praćenju dijabetesa“ koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

Student

_____

U Puli, rujan 2022.

SADRŽAJ

1. Uvod

U ovom diplomskom radu opisan je postupak izrade i upotreba aplikacije za potporu oboljelima od dijabetesa kojom bi se pratile životne navike dijabetičara kao što su unos terapije, unos hrane te unos fizičke aktivnosti, u cilju poboljšanja svakodnevnog života.

Rad se sastoji od dva dijela. U prvom dijelu objašnjeno je što modeliranje sustava pomoću UML-a, korištene tehnologije: Java, Android, Android Studio te su objašnjene osnovne aplikacijske komponente i komponente Android arhitekture. Definirani su noviji AndroidX koncepti u razvoju mobilnih aplikacija: *ViewModel*, *LiveData* i *Navigation Components*. Opisana je aplikacijska arhitektura te je pojašnjeno pohranjivanje na Oblaku i testiranje Android aplikacije.

U drugom dijelu, izvršena je analiza sustava za potporu u praćenju dijabetesa i razrada funkcionalnosti same aplikacije. Dokumentiran je sustav, što uključuje izradu korisničkog sučelja, korisničke scenarije, dijagram slučaja uporabe, sekvencijalni dijagram i dijagram klasa. Nakon toga opisana je implementacija u programski kod, odnosno izrada aplikacije. Na kraju su prikazane kratke korisničke upute.

2. Modeliranje sustava pomoću UML jezika

Za proces razvoja sustava važno je imati dobro definiranu i izražajnu notaciju. Standardna notacija omogućava opis scenarija i formuliranje arhitekture te nedvosmislen prikaz sustava drugima. Jedna od mogućnosti modeliranja sustava je pomoću UML notacije. (Booch, 2007)

UML (eng. *Unified Modelling Language*) definira se kao ujedinjeni jezik za modeliranje koji omogućava analizu, specifikaciju i dizajn softverskog sustava. Smatra se standardom za dokumentiranje objektno orijentiranog sustava. (Miles i Hamilton, 2006)

UML sadrži više različitih vrsta dijagrama, a svaki od njih pruža određen pogled na sustav. UML dijagrami mogu se klasificirati u dvije skupine (Booch, 2007):

- **strukturni dijagrami** – koriste se za prikaz statičke strukture elemenata. Mogu prikazivati arhitektonsku organizaciju sustava, fizičke elemente u sustavi i slično. Strukturni dijagrami su sljedeći:
    - dijagram paketa
    - dijagram klasa
    - dijagram komponenti
    - dijagram postavljanja
    - dijagram objekata
    - dijagram kompozitne strukture
- **dijagrami ponašanja** – koriste se za prikaz dinamičke bihevioralne semantike problema ili za prikaz njegove implementacije kroz sljedeće dijagrame:
    - dijagram slučajeva uporabe
    - dijagram aktivnosti
    - dijagram stanja sustava
    - dijagrami interakcije:
        - sekvencijalni dijagram

- komunikacijski dijagram
- dijagram pregleda interakcija
- vremenski dijagram

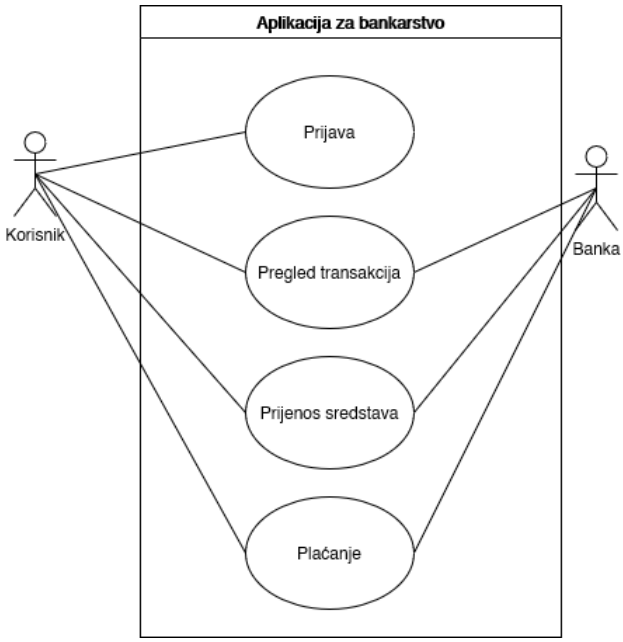
Činjenica da je UML detaljna specifikacija ne znači da se mora koristiti svaki aspekt za dokumentiranje sustava. Uzimajući to u obzir, u daljnjem tekstu bit će opisani neki od dijagrama koji su dovoljni za jednostavnu analizu i dizajn, odnosno modeliranje sustava.

Dijagrami slučajeva korištenja koriste se za opisivanje konteksta sustava koji se gradi i opis funkcionalnosti koje sustav nudi. Sastoji se od (Booch, 2007):

- aktera - entiteta koji komuniciraju sa sustavom
- slučajeva korištenja - predstavljaju što akteri žele da sustav radi za njih

Dijagrami slučajeva korištenja vizualno specificiraju odgovornosti entiteta, no ne pokazuje kako sustav obrađuje zahtjev. (Booch, 2007)

Slika 1: Primjer dijagrama slučajeva korištenja



(Izvor: autor)

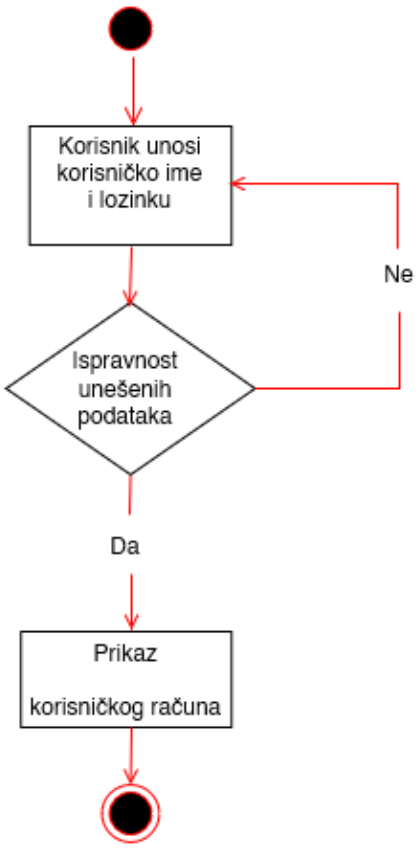
Dijagrami aktivnosti prikazuje tok aktivnosti, bilo u sustavu, poslovanju, tijeku rada ili nekom drugom procesu. Dijagrami aktivnosti su usredotočeni na aktivnosti koje se izvode i tko (ili što) je odgovoran za izvođenje tih aktivnosti. Omogućuje određivanje načina kako će sustav ostvariti svoje ciljeve. (Miles i Hamilton, 2006)

Dijagram aktivnosti sastoji se od (Booch, 2007):

- čvorova radnji - predstavljaju funkcionalnost u aktivnosti
- kontrolnih čvorova - koordiniraju tok kontrole u aktivnosti
- objektnih čvorova - pomažu u definiranju toka objekta u aktivnosti

Kontrolni čvorovi se dijele na početni i završni, čvorove odluke i sjedinjena, te dijeljenja i spajanja. (Booch, 2007)

Slika 2: Primjer dijagrama aktivnosti



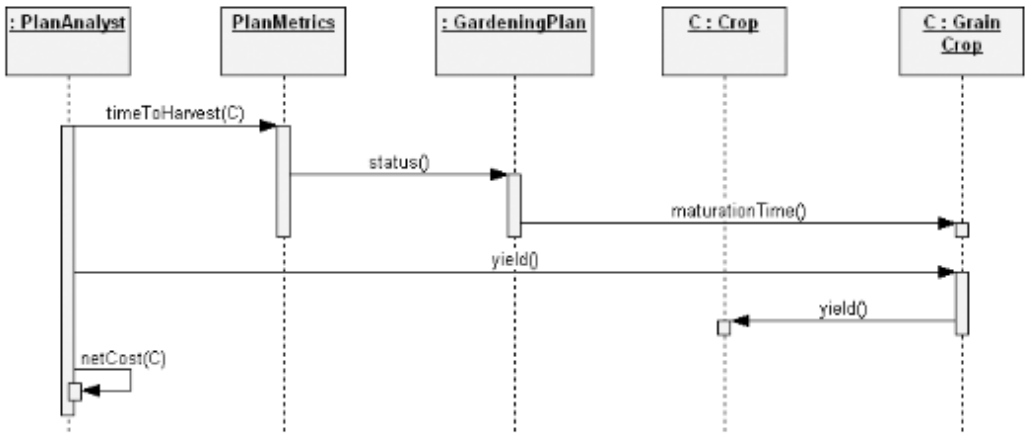
(Izvor: autor)

Sekvencijalni dijagram prikazuje kako su objekti međusobno povezani te koje poruke prenose dok sudjeluju u određenoj interakciji. Pomoću sekvencijalnog dijagrama, moguće je opisati koje će se interakcije pokrenuti kada se izvrši određen slučaj upotrebe i kojim će se redoslijedom te interakcije dogoditi. Iako pokazuju mnogo različitih informacija o interakciji, njihova snaga je jednostavan i učinkovit način na koji prikazuju redoslijed događaja unutar interakcije. (Miles i Hamilton, 2006)

Osnove sekvencijalnog dijagrama su (Miles i Hamilton, 2006):

- objekti (sudionici)
- događaji – gradivni blokovi za prijenos poruke
- životne crte – ukazuju na postojanje podataka
- poruke – označavaju događaje ili pozivanje operacija. Poruke mogu biti sinkrone, asinkrone i povratne.

Slika 3: Primjer sekvencijalnog dijagrama



(Izvor: Object-Oriented Analysis and Design with Applications, 2007)

Klasni dijagram koristi se za prikaz klasa i njihovih veza u logičkom pogledu na sustav. Pomoću njega se opisuje struktura sustava. Osnovni elementi dijagrama su klase i njihove veze. (Miles i Hamilton, 2006)

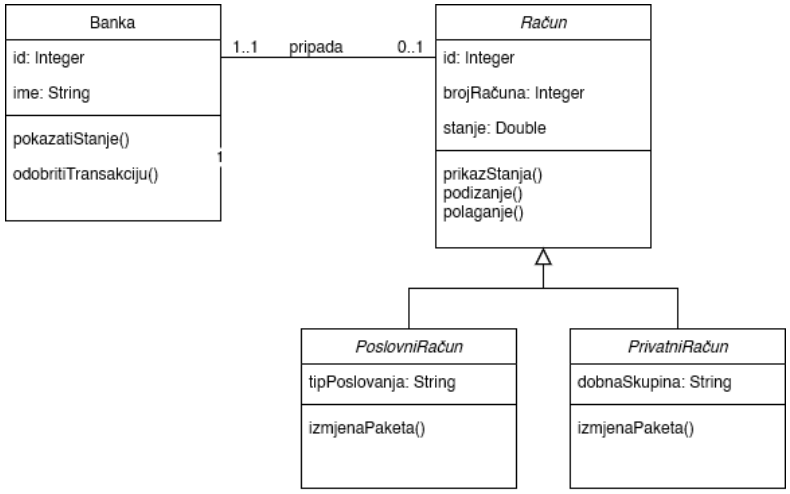
Klase opisuju različite vrste objekata koje sustav može imati. Sastoji se od (Miles i Hamilton, 2006):

- atributa – informacije koje klasa sadrži
- operacija – predstavljaju ponašanje koje klasa pokazuje

Vezama se prikazuje povezanost između klasa. Vrste veza su (Miles i Hamilton, 2006):

- ovisnost (eng. *dependency*) – objekt klase ovisi o objektima druge klase
- asocijacija (eng. *association*) – veza između klasa koja pokazuje da između objekata iz dotičnih klasa postoji poveznica
- agregacija (eng. *aggregation*) – kada jedna klasa posjeduje, ali dijeli referencu objektima druge klase
- kompozicija (eng. *composition*) – pokazuje da jedna klasa sadrži objekte druge klase
- nasljeđivanje (eng. *inheritance*) – pokazuje da jedna klasa proširuje drugu klasu

Slika 4: Primjer klasnog dijagrama



(Izvor: autor)

3. Tehnologije

Tehnologija je izraz za skup praksi, alata, tehnika i metoda. Neke od osnovnih tehnologija potrebne za razvoj aplikacije su Java, Android i Android Studio koji sadrži Android SDK.

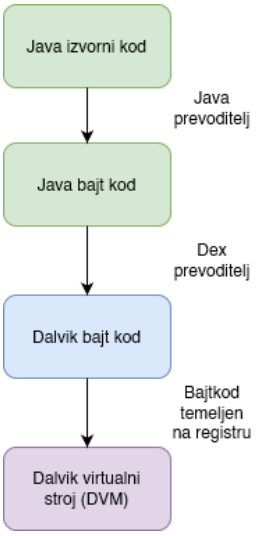
3.1. Java

Java je objektno orijentirani jezik temeljen na klasama. Osmišljen je tako da bude dovoljno jednostavan da mnogi programeri budu tečni u njemu. Programski jezik Java povezan je sa programskim jezicima C i C++, ali je organiziran prilično drugačije; izostavljeni su brojni aspekti iz C i C++-a te su implementirane neke ideje iz drugih jezika. (Gosling, 2015)

Java pruža stabilnu, čvrstu osnovu za razvoj poslovno kritičnih aplikacija. Namijenjen je da bude proizvodni jezik, a ne istraživački jezik. Zato kao programski jezik, ima relativno konzervativan dizajn i sporo doživljava promjene. (Evans i Flanagan, 2015)

Android aplikacije napisane su u Javi prije nego se pretvore u Android-ov vlastiti format datoteke DEX. Na dijagramu ispod prikazana je transformacija Java izvornog koda (.java) u Java bajt kod (.class) te zatim u Dalvik izvršnu datoteku. (Darwin, 2017)

Slika 5: Transformacija Java koda



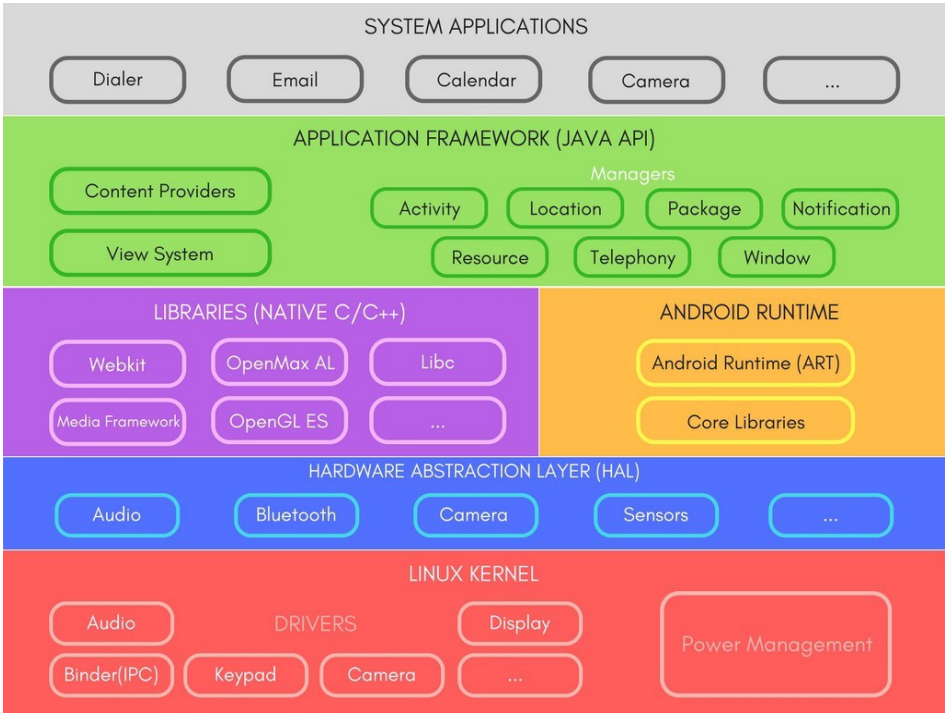
(Izvor: autor)

3.2. Android

Android je operacijski sustav otvorenog koda, temeljen je na jezgri Linux. Namijenjen je velikom rasponu uređaja, neki od njih su pametni telefoni i tablet računala. (Android, 2021c)

Android se sastoji od raznih slojeva, a svaki sloj ima svoje osobine i svrhu. Slojevi nisu strogo odvojeni te se često nadovezuju. Prije kodiranja Android aplikacije, razumijevanje izgleda sustava pomaže u shvaćanju što se može ili ne može lako učiniti. (Gargenta, 2011)

Slika 6: Arhitektura Android-a



Izvor: (Android, 2021c)

Android platforma sastoji se od (Android, 2021c):

- **Linux Kernel** – temelj Android platforme. Neke od osnovnih funkcija su upravljanje memorijom niske razine i upravljanje procesima.
- **Hardverski Apstraktni Sloj (HAL)** – pruža standardna sučelja koja izlažu hardverske mogućnosti uređaja Java API okviru više razine. Kada API okvir uputi poziv za pristup hardveru uređaja, sustav Androida učita modul biblioteke za tu hardversku komponentu (npr. kamera, bluetooth) .

- **Android Runtime** – napisan je kako bi vrtio više virtualnih strojeva na uređajima s malo memorije izvršavanjem DEX datoteka. Alati za izgradnju, kao što je d8, prevode Java izvore u DEX bajt kod, koji se zatim može izvoditi na Android-u. Osnovne biblioteke pružaju većinu funkcionalnosti Java programskog jezika (uključujući značajke Java 8) koje Java API okvir koristi.
- **C/C++ biblioteke** – mnoge izvorne komponente i usluge, kao što su ART i HAL, izgrađene su od izvornog koda koji zahtijeva izvorne biblioteke napisane u C i C++. Može se koristiti Android NDK za pristup bibliotekama.
- **Java API** – građevni blokovi potrebni za izradu Android aplikacije. Pojednostavljuju korištenje osnovnih, modularnih komponenti sustava i usluga.
- **Aplikacije sustava** – osnovne aplikacije Androida: za e-poštu, SMS, kameru itd. Korisnik može instalirati alternativne aplikacije.

3.3. Android studio

Android Studio je službeno integrirano razvojno okruženje za razvoj Android aplikacija, temeljen na integriranom razvojnom okruženju IntelliJ IDEA. Android Studio ima odgovornost pružiti sučelje za izradu aplikacija i u pozadini voditi složeno upravljanje datotekama. (Darwin, 2017)

Instalacija Android Studio-a sadrži Android SDK - alate potrebni za izgradnju Android aplikacije kao što su razne knjižice, razotkrivač i otklanjač problema (eng. *debugger*), emulator (za kreiranje i testiranje aplikacija na virtualnim uređajima) i slično. (Philips, 2017)

Neke od značajke Android Studio-a su što je sustav gradnje baziran na *Gradle*-a (sustav automatizacije gradnje), brz emulator bogat značajkama te sadrži robusne mehanizme testiranja. (Darwin, 2017)

4. Struktura Android aplikacije

Tipična Android aplikacija sastoji se od više aplikacijskih komponenti kao što su aktivnosti, fragmenti, usluge, prijemnici emitiranja i pružatelji sadržaja. Potrebno je razumjeti osnovne gradivne elemente aplikacije za definiranje arhitekture aplikacije u kojoj se organiziraju i povezuju komponente kako bi na kraju aplikacija bila dobro konstruirana što omogućava lakše testiranje, širenje i održavanje aplikacije. (Android, 2022a)

4.1. Aplikacijske komponente

Aplikacijske komponente su osnova Android aplikacije. To su konceptualne stavke na kojima se može samostalno raditi a zatim se sastavljaju kako bi se stvorila veća cjelina, odnosno aplikacija. Svaka komponenta ima svoju jedinstvenu funkciju te životni ciklus. (Gargenta, 2011.)

Četiri osnovne komponente su (Android, 2022a):

- aktivnosti
- usluge
- prijemnici emitiranja
- pružatelji sadržaja.

Postoje dodatne komponente koje se koriste u izgradnji gore navedenih entiteta, njihove logike i povezivanja između njih. Jedni od bitnijih dodatnih komponenti su fragmenti koji predstavljaju modularni dio korisničkog sučelja unutar aktivnosti.

4.1.1. Aktivnost

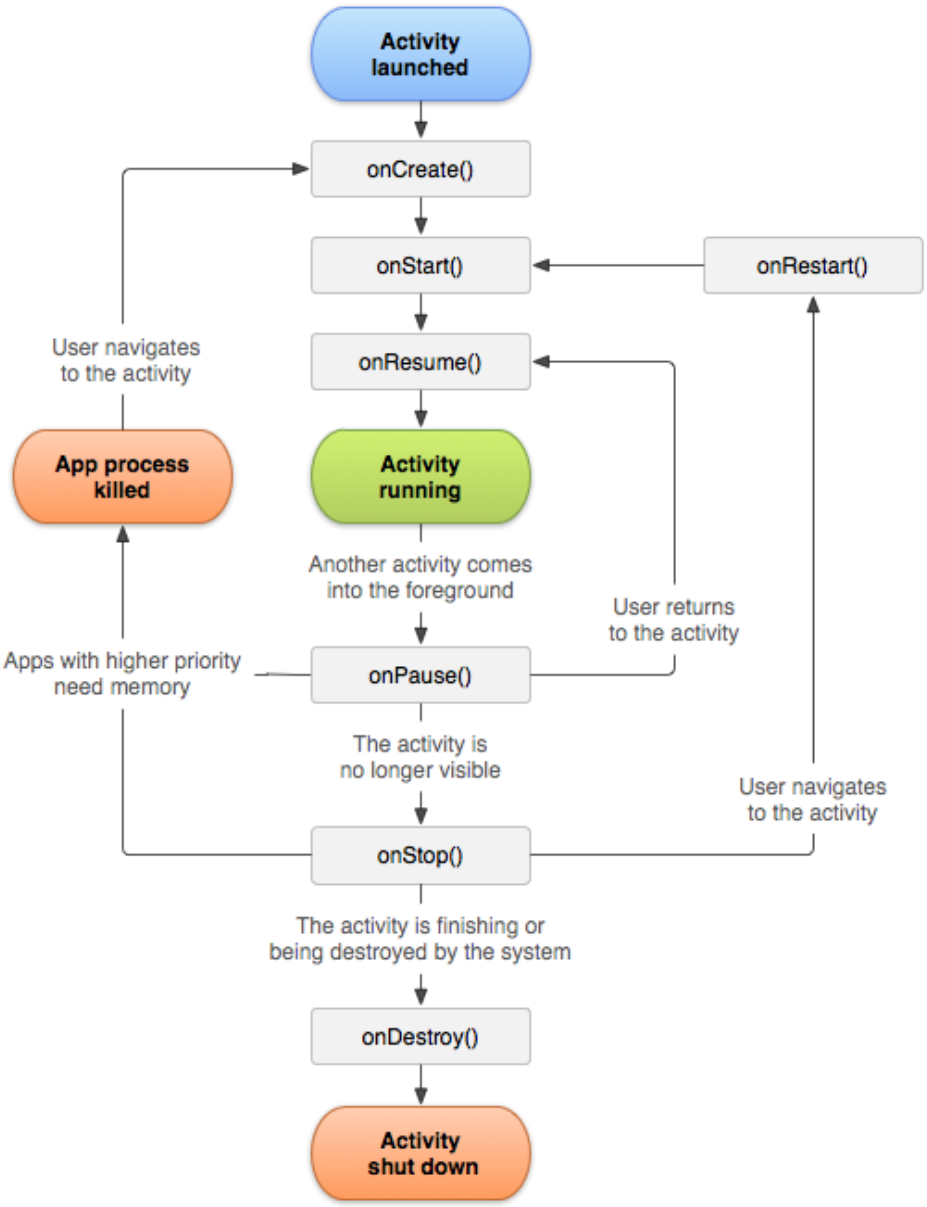
Aktivnost (eng. *Activity*) predstavlja zaslon sa korisničkim sučeljem. Na primjer, aplikacija za elektroničku poštu ima jednu aktivnost u kojoj je prikazana lista nove e-pošte, drugu aktivnost za sastavljanje e-pošte te još jednu aktivnost za čitanje individualne e-pošte. Svaka ta aktivnost je nezavisna od druge, tako da druga aplikacija može pokrenuti neku od tih aktivnosti ukoliko aplikacija za e-poštu dopusti. Za primjer se može uzeti aplikacija za fotografiranje koja može pokrenuti u aplikaciji za e-poštu aktivnost za sastavljanje nove e-pošte kako bi korisnik mogao podijeliti fotografiju. (Android, 2022d)

Aktivnost kroz životni ciklus prolazi različit broj stanja i za prijelaz između stanja mogu se koristiti određeni povratni pozivi (eng. *callbacks*). Stanja kroz koja prolazi su aktivno (eng. *active*), vidljiv (eng. *visible*), zaustavljen (eng. *stopped*) i sakriven (eng. *hidden*) te uništen (eng. *destroyed*). Pomoću metoda povratnih poziva mogu se određivati kako će se aktivnost ponašati tijekom određenih radnji. (Android, 2022d)

Šest osnovnih metoda koje služe kao povratni pozivi su (Android, 2022d):

- onCreate() – izvodi se osnovna logika pokretanja aplikacije
- onStart() – omogućuje da je aktivnost vidljiva korisniku
- onResume() – poziva se kada korisnik ponovo započne interakciju s aplikacijom
- onPause() – metoda za pauziranje ili podešavanje operacija koje se ne bi trebale nastaviti izvoditi u aktivnosti, poziva se kao prva indikacija da korisnik napušta aktivnost
- onStop() – poziva se kada aktivnost više nije vidljiva korisniku, npr. prilikom poziva nove aktivnosti koja zauzme cijeli zaslon
- onDestroy() – poziva se prije nego se aktivnost uništi (zatvaranjem aktivnosti ili zbog sustava i konfiguracijskih promjena)

Slika 7: Životni ciklus aktivnosti



(Izvor: Android, 2022d)

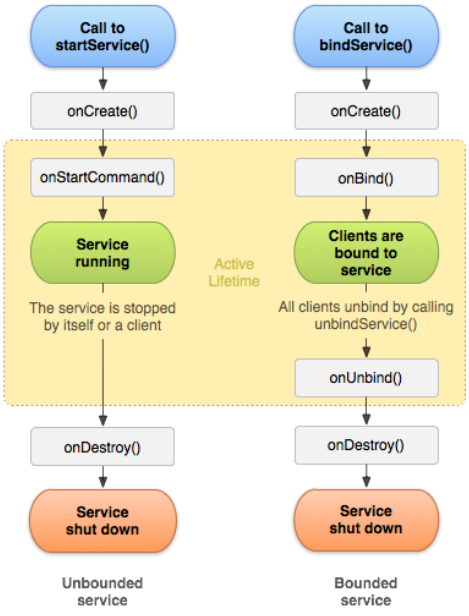
4.1.2. Usluga

Usluga (eng. *service*) je komponenta bez korisničkog sučelja koja u pozadini obavlja određene operacije za koje nije potrebna komunikacija sa korisnikom te mogu izvršavati operacije čak i ako se izađe iz aplikacije. Usluga može biti duže pokrenuta od aktivnosti. Jedan od primjera korištenja usluga je reprodukcija glazbe u pozadini dok korisnik koristi druge aplikacije. (Darwin, 2017.) Usluga može biti u dva stanja (Android, 2022h):

- pokrenut (eng. *started*) kada se pokrene pomoću aplikacijske komponente kao što je aktivnost pozivom metode *startService()*. Pokrenuta usluga može biti u pozadini neodređeno, čak i kada se aplikacijska komponenta uništi.
- vezan (eng. *bound)* je kada se aplikacijska komponenta veže na njega pozivom *bindService()*. Vezana usluga nudi sučelje klijent-poslužitelj koje komponentama omogućava komunikaciju sa uslugom, slanje zahtjeva, dobivanje rezultata i slično.

Usluga ima životni ciklus te metode pomoću kojih se može pratiti stanje usluge te prema tome izvršavati željene operacije. Pomoću dijagrama na lijevoj strani prikazan je životni ciklus usluga kada je usluga započeta pomoću *startService()* i na desnoj strani je prikazan dijagram životnog ciklusa usluge kada se usluga započne sa *bindService()*. (Android, 2022h)

Slika 8: Životni ciklus usluge



(Izvor: Android, 2022h)

Usluga kao klasa sadrži razne metode, no neke su bitnije te iako se ne moraju sve implementirati, važno je znati svaku od njih i implementirati one koje omogućavaju da se aplikacija ponaša kako korisnik očekuje prilikom određenog događaja. (Android, 2022h)

Neke od tih metoda su (Android, 2022h):

- onStartCommand() – sustav poziva ovu metodu kada druga komponenta, kao što je aktivnost, zatraži da usluga bude započeta pomoću *startService()*. Ukoliko se usluga započne na ovakav način, potrebno je prekinuti rad pomoću metode *stopSelf()* ili *stopService()*.
- onBind() – sustav poziva ovu metodu kada druga komponenta želi biti povezana sa uslugom pozivom *bindService()*. Implementacijom ove metode, potrebno je pružiti sučelje kako bi klijenti mogli komunicirati sa uslugom.
- onUnBind() – sustav poziva ovu metodu kada se svi klijenti isključe od određenog sučelja prikazanog od strane usluge
- onRebind() – sustav poziva ovu metodu kada se novi klijenti spajaju na uslugu, nakon što je prethodno obavješten da su svi odspojeni pomoću *onUnbind()*
- onCreate() – sustav poziva ovu metodu kada se usluga prvi put kreira
- onDestroy() – sustav poziva ovu metodu kada se usluga više neće koristiti te će biti uništena

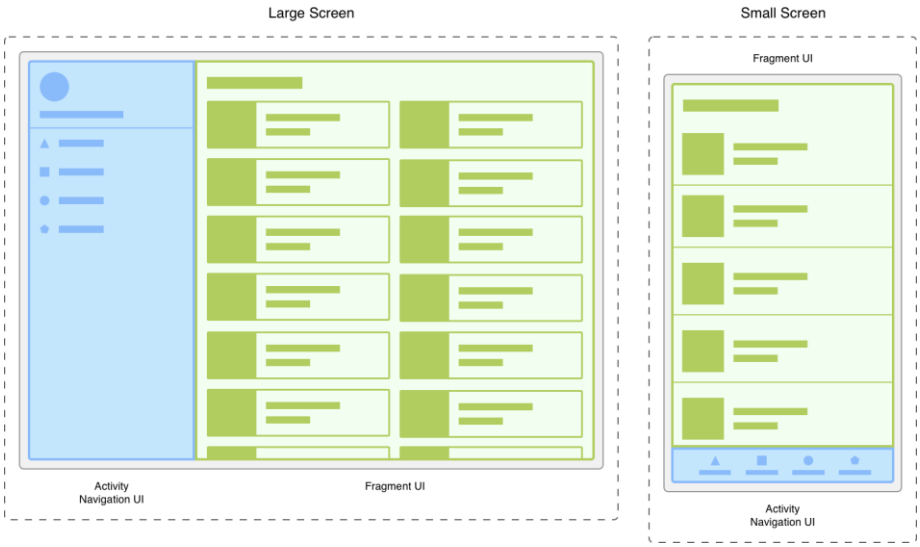
4.1.3. Fragment

Fragment predstavlja dio korisničkog sučelja u aktivnosti. Može se smatrati mini aktivnošću, uzimajući u obzir da ima svoj životni ciklus, događaje i shemu (eng. *layout*). Međutim, fragment ne može postojati samostalno - veže se za aktivnost ili drugi fragment. (Darwin, 2017.)

Kada se kaže da je fragment vezan uz aktivnost to znači da je životni ciklus fragmenta direktno pod utjecajem aktivnosti. Na primjer, kada se aktivnost uništi, svi fragmenti unutar te aktivnosti će također biti uništeni, ako je aktivnost zaustavljena – svi fragmenti su zaustavljeni te ponovim pokretanjem aktivnosti pokrenu se i svi fragmenti. Međutim, kada je aktivnost pokrenuta, svaki se fragment individualno može manipulirati po želji i potrebi. (Android, 2021a)

Iako fragmenti imaju razne slučajeve upotrebe, poput ponovnog korištenja u različitim aktivnostima kao što su fragmenti specijalizirani za prikaz liste; glavni je zbog podrške dinamičnijem i fleksibilnijem dizajnu korisničkog sučelja, na velikim zaslonima poput tableta. S obzirom da je zaslon tableta veći od ekrana telefona, sadrži više prostora za kombiniranje i izmjenu komponenti. Fragmenti omogućavaju drugačiji dizajn za zaslone mobitela i tableta bez potrebe za složenim promjenama hijerarhije dizajna. (Android, 2021a)

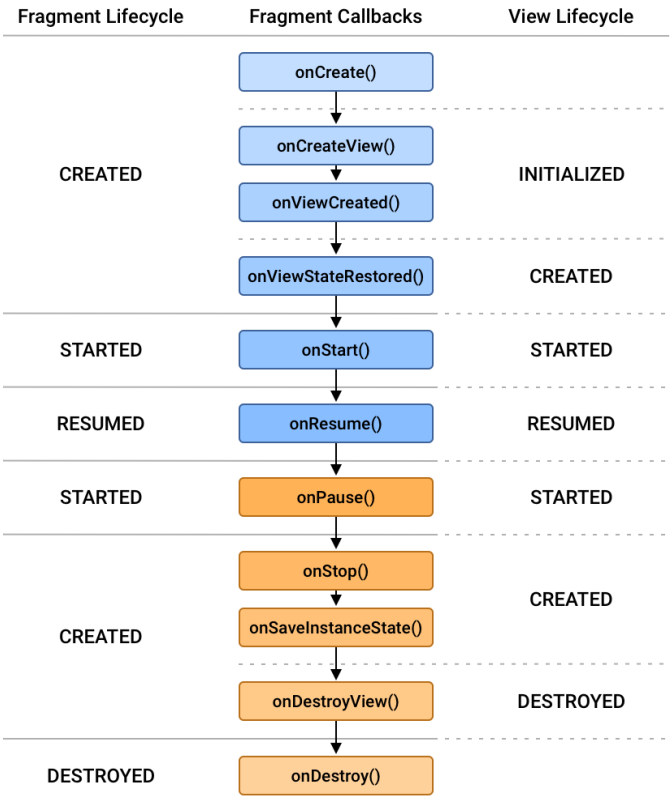
Slika 9: Raspored navigacije i fragmenta na velikom i malom zaslonu



(Izvor: Android, 2021a)

Stanja životnog ciklusa su: stvoren (eng. *created*), započet (eng. *started*), nastavljen (eng. *resumed*), uništen (eng. *destroyed*). Praćenje životnog ciklusa moguće je pomoću implementacije *LifecycleObserver*-a, no također je moguće pratiti metodama kao što su onCreate(), onStart(), onResume(), onStop(), onDestroy(). Na slici ispod prikazan je životni ciklus fragmenta uz metode fragmenata i životni ciklus prikaza. (Android, 2021b)

Slika 10: Životni ciklus fragmenta



(Izvor: Android, 2021a)

U arhitekturi aplikacije baziranoj na fragmentima, kao što je to u arhitekturi „jedna aktivnost“ (eng. *single-activity architecture*), aktivnost se koristi kao navigacijski upravljač za kretanje kroz aplikaciju te prijenos podataka, dok fragmenti sadrže prikaze i logiku vezanu uz stanja prikaza, događaje te dohvat i spremanje podataka. (Android, 2021a)

4.1.4. Prijemnik emitiranja

Prijemnik emitiranja (eng. *broadcast receivers*) daju odgovor na emitirane poruke iz drugih aplikacija ili iz samog sustava. Te se poruke nazivaju događajima (eng. *events*) ili namjerama (eng. *intents*). Na primjer, ukoliko je postotak baterije nizak te aplikacija konzumira veliku količinu, sustav javlja da je baterija skoro prazna, a prijemnik u aplikacji može pokrenuti odgovarajuću radnju kao što je recimo prekid funkcije praćenja koraka u fitnes aplikaciji. (Darwin, 2017.)

4.1.5. Pružatelj sadržaja

Pružatelji sadržaja (eng. *content providers*) definiraju se kao sučelja za dijeljenje podataka između aplikacija na zahtjev. Zahtjevi se izvršavaju pomoću metoda klase *ContentResolver*. Pružatelj sadržaja koristi različite načine za spremanje podataka te se podaci uglavnom spremaju u bazu podataka. Pružatelji sadržaja ponašaju se veoma slično kao baza podataka gdje se mogu izvršiti upiti, uređivanje, dodavanje ili brisanje sadržaja pomoću metoda insert(), update(), delete(), query(). Primjer korištenja pružatelja sadržaja jest pristup imeniku u svrhu izrade alternativnog imenika za aplikaciju. (Gargenta, 2011.)

4.2. Komponente Android arhitekture

Komponente Android arhitekture (AAC) su kolekcija biblioteka koje pomažu u dizajniranju robusnijih aplikacije koje se mogu testirati i održavati. Pomažu u upravljanju životnih ciklusa komponenti korisničkog sučelja te u rukovanju dosljednosti podataka. (Android, 2022c)

Korištenjem ovih komponenti mogu se riješiti uobičajeni problemi kao što su promjene konfiguracije, curenje memorije, pisanja lako testirajućih aplikacija, smanjivanje ponavljanja koda i to uz zadržavanje arhitekture. (Android, 2022c)

Komponente Android arhitekture su (Android, 2022c):

- **Data binding** – pomaže u deklarativnom povezivanju elemenata korisničkog sučelja u shemi sa izvorima podataka
- **Lifecycles** – upravlja životnim ciklusima aktivnosti i fragmenata
- **LiveData** – obavještava poglede o promjenama u bazi podataka
- **Navigation** – koristi se za lakšu navigaciju unutar aplikacije

- **Paging** – pomaže u postupnom učitavanju informacija na zahtjev iz izvora podataka
- **Room** – pruža sloj apstrakcije preko SQLite-a. Omogućava lakši pristup bazi podataka.
- **ViewModel** – upravlja podacima povezanim s korisničkim sučeljem svjestan životnog ciklusa.
- **WorkManager** – upravlja svim pozadinskim poslovima u Androidu u skladu s odabranim okolnostima.

Na sljedećim stranicama bit će objašnjene komponente koje su korištene u izradi aplikacije: *ViewModel, LiveData, Navigation Component*, *Room*.

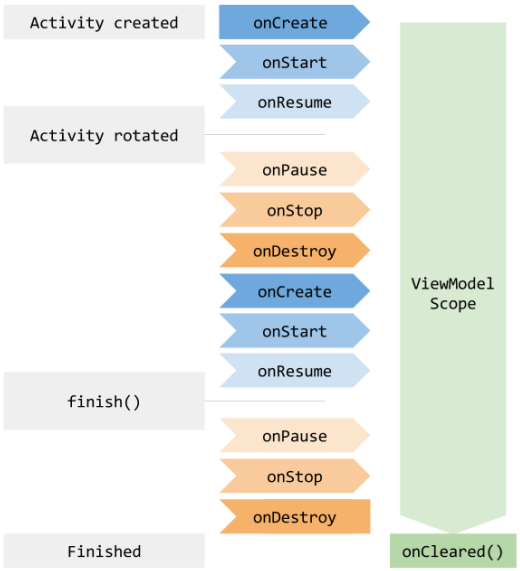
4.2.1. ViewModel

ViewModel klasa namijenjena je za spremanje i upravljanje podacima vezanim za korisničko sučelje, uzimajući u obzir životni ciklus. *ViewModel* omogućava preživljavanje podataka nakon konfiguracijskih promjena. Konfiguracijske promjene uzrokuju ponovno kreiranje aktivnosti i gubljenje podataka ukoliko nisu spremljeni na pravilan način, uz to je moguće javljanje greški ili čak rušenje u aplikaciji. Primjer konfiguracijske promjene je rotacija zaslona. Kako se tijekom takve situacije ne bi izgubili podaci, potrebno je koristiti *ViewModel*. (Android, 2022i)

Osim što je *ViewModel* odgovoran za pripremu i upravljanje podataka za aktivnost ili fragment, on također upravlja komunikacijom aktivnosti ili fragmenta sa ostatkom aplikacije (točnije poziva klase za poslovnu logiku). (Android, 2022i)

Ilustracija na sljedećoj stranici prikazuje razna stanja životnog ciklusa aktivnosti tijekom rotacije zaslona te kraj aktivnosti. Također prikazuje životni vijek *ViewModel*-a usporedno sa životnim ciklusom aktivnosti s kojim je povezan.

Slika 11: Životni ciklus *ViewModel*-a



(Izvor: Android, 2022i)

Implementacija *ViewModel*-a može se prikazati kroz primjer za prikaz liste korisnika. Samoj listi korisnika neće se pristupiti kroz aktivnost ili fragment, već kroz ViewModel što je vidljivo u kodu ispod.

Slika 12: Primjer *ViewModel*-a

```
public class MyViewModel extends ViewModel {

    private MutableLiveData<List<User>> users;

    public LiveData<List<User>> getUsers() {
        if (users == null) {
            users = new MutableLiveData<List<User>>();
            loadUsers();
        }
        return users;
    }

    private void loadUsers() {
        // Asinkrona operacija za dohvat korisnika.
    }
}
```

(Izvor: Android, 2022i)

Moguće je pristupiti listi u aktivnosti tako da se kreira *ViewModel* prvi put kada sustav poziva metodu onCreate(). Ponovo kreirane aktivnosti dobivaju istu instancu MyViewModel-a kreirane od prve aktivnosti.

Slika 13: Primjena ViewModel-a u aktivnosti

```
public class MyActivity extends AppCompatActivity {
    public void onCreate(Bundle savedInstanceState) {

        MyViewModel model = new
ViewModelProvider(this).get(MyViewModel.class);

        model.getUsers().observe(this, users -> {
            // Ažuriranje UI-a.
        });
    }
}
```

(Izvor: Android, 2022i)

4.2.2. LiveData

LiveData je klasa koja drži podatke sljedeći obrazac promatrača, što znači da klasa može biti promatrana. *LiveData* klasa svjesna je životnog ciklusa. *LiveData* olakšava sinkronizaciju ekrana sa podacima. Na primjer, uz korisničko sučelje postoji i *LiveData* objekt za koji se želi da je prikazan na ekranu. U tom slučaju, korisničko sučelje promatra *LiveData* objekt. To omogućuje da korisničko sučelje bude obavješteno za moguće promjene. Točnije, kada se *LiveData* promjeni, korisničko sučelje bude obavješteno i zatim se korisničko sučelje prilagodi promjenama, odnosno prikaže nove podatke. (Android, 2022e)

LiveData svjestan je relevantnih promjena statusa životnog ciklusa te automatski upravlja zaustavljanjem i nastavkom. To znači da razumije je li korisničko sučelje aktivno, neaktivno ili uništeno. *LiveData* zna koje je stanje korisničkog sučelja jer se prosljeđuje pozivom promatrača. Također je važno napomenuti da *LiveData* ažurira jedino promatrače koji su u aktivnom životnom ciklusu. (Android, 2022e)

Neke prednosti korištenja *LiveData* objekta su (Android, 2022e):

- odgovarajuće ažuriranje korisničkog sučelja – uz korištenje *LiveData*, korisničko sučelje aplikacije promijenit će se samo ako dođe do promjene u podacima

- bez rušenja zbog zaustavljenih aktivnosti – promatrači su obavješteni da su u aktivnom stanju. Zbog toga neće doći do rušenja u aplikaciji kada je aktivnost zaustavljena ili pauzirana.
- ispravne konfiguracijske promjene – ako se aktivnost ili fragment ponovo kreiraju zbog konfiguracijskih promjena kao što je rotacija zaslona, odmah dobiva zadnje stanje podataka.
- ažurnost podataka – promatrač dobiva najnovije podatke čak i ako se nalazi u pozadini ili ako je u stanju pauze. Kada promatrač nastavi sa svojim radom, promatraču se dostave posljednji ažurirani podaci.
- bez curenja memorije – promatrači su vezani za objekte životnog ciklusa (eng. *lifecycle objects*) i čiste se kada se uništi životni ciklus za koji su vezani. Iz tih razloga nema curenja memorije.

Implementacija *LiveData* započinje kreiranjem instance *LiveData* kako bi se držao određeni tip podataka. Kreiranje se obično radi u *ViewModel*-u.

Slika 14: Kreiranje instance *LiveData*

```
public class NameViewModel extends ViewModel {

private MutableLiveData<String> currentName;

    public MutableLiveData<String> getCurrentName() {
        if (currentName == null) {
            currentName = new MutableLiveData<String>();
        }
        return currentName;
    }

// Ostatak ViewModel-a...

}
```

(Izvor: Android, 2022e)

Zatim se kreira objekt promatrač (engl. *Observer*) koji definira metodu onChanged(), koja kontrolira što se događa kada se čuvani podaci *LiveData* objekta promjene. Objekt promatrač se kreira u kontroleru korisničkog sučelja, kao što je aktivnost ili fragment.

Slika 15: Kreiranje promatrača

```
final Observer<String> nameObserver = new Observer<String>() {
    @Override
    public void onChanged(@Nullable final String newName) {
        // Ažuriranje UI-a, u ovom slučaju TextView.
        nameTextView.setText(newName);
    }
};
```

(Izvor: Android, 2022e)

Nakon kreiranja objekta promatrača, objekt promatrač veže se na *LiveData* objekt koristeći metodu observe(). Metoda observe() uzima *LifecycleOwner* objekt. Time se objekt promatrač pretplaćuje na *LiveData* objekt kako bi bio obavješten o promjenama. Objekt promatrač uglavnom se povezuje u kontroleru korisničko sučelja, kao što je aktivnost ili fragment.

Slika 16: Vezanje promatrača na *LiveData*

```
model.getCurrentName().observe(this, nameObserver);
```

(Izvor: Android, 2022e)

MutableLiveData koristi se umjesto *LiveData* kada se želi ažurirati spremljene podatke. *MutableLiveData* klasa sadrži dvije dodatne metode koje omogućavaju određivanje vrijednosti *LiveData* objekta: setValue(T) i postValue(T). *MutableLiveData* najčešće se koristi u *ViewModel*-u, *ViewModel* izlaže samo nepromjenjive *LiveData* objekte promatračima. Nakon što se poveže *LiveData* sa promatračem, moguće je ažurirati vrijednost *LiveData* objekta. Idući primjer prikazuje promjenu vrijednosti *LiveData* objekta, koja se događa nakon što korisnik pritisne gumb. (Android, 2022e)

Slika 17: Promjena vrijednosti LiveData

```
button.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        String anotherName = "John Doe";
        model.getCurrentName().setValue(anotherName);
    }
});
```

(Izvor: Android, 2022e)

Transformacija *LiveData* objekta moguća je pomoću klase *Transformations* koja se nalazi u *Lifecycle* paketu. Klasa sadrži pomoćne metode koje podržavaju scenarije kao što su slučaj kada se želi napraviti promjena vrijednosti pohranjene u *LiveData* objektu prije nego se pošalje promatračima ili kada se želi vratiti druga instanca *LiveData* temeljenoj na vrijednosti neke druge(Android, 2022e)

- **Transformations.map() - p**rimjenjuje funkciju na vrijednost pohranjenu u objektu *LiveData* i prenosi rezultat nizvodno.

Slika 18: Primjer Transformations.map()

```
LiveData<User> userLiveData = ...;
LiveData<String> userName =
Transformations.map(userLiveData, user -> {
    user.name + " " + user.lastName
});
```

(Izvor: Android, 2022e)

- **Transformations.switchMap() - s**lično kao map(), primjenjuje funkciju na vrijednost pohranjenu u objektu *LiveData* te razmotava i šalje rezultat nizvodno.

Slika 19: Primjer Transformations.switchMap()

```
private LiveData<User> getUser(String id) {
  ...;
}

LiveData<String> userId = ...;
LiveData<User> user = Transformations.switchMap(userId, id ->
getUser(id) );
```

(Izvor: Android, 2022e)

Primjer koda ispod prikazuje implementaciju pretraživanja poštanskog koda kao transformaciju upisane adrese. Polje poštanskog koda (eng. *postalCode*) definiran je kao transformacija upisane adrese (eng. *adressInput*). Sve dok aplikacija ima aktivan promatrač koji je povezan sa poljem poštanskog koda, svaki puta kada se unese nova adresa izmjeni se i poštanski broj ukoliko adresa pripada drugom mjestu.

Slika 20: Implementacija pretraživanja poštanskog koda

```
class MyViewModel extends ViewModel {
    private final PostalCodeRepository repository;
    private final MutableLiveData<String> addressInput = new
MutableLiveData();
    public final LiveData<String> postalCode =
            Transformations.switchMap(addressInput, (address) -> {
                return repository.getPostCode(address);
             });

  public MyViewModel(PostalCodeRepository repository) {
      this.repository = repository
  }

  private void setInput(String address) {
      addressInput.setValue(address);
  }
}
```

(Izvor: Android, 2022e)

4.2.3. Navigation

Navigacija se odnosi na interakcije koje omogućavaju korisniku da se kreće kroz različite dijelove sadržaja unutar aplikacije. Komponenta *Navigation* pomaže u implementaciji navigacije, od jednostavnog klika na gumb do kompleksnijih obrazaca, kao što su trake aplikacije i navigacijske ladice. (Android, 2022f)

Komponenta osigurava dosljedno i predvidljivo ponašanje korisnika pridržavajući se već utvrđenog skupa načela navigacije (Android, 2022f):

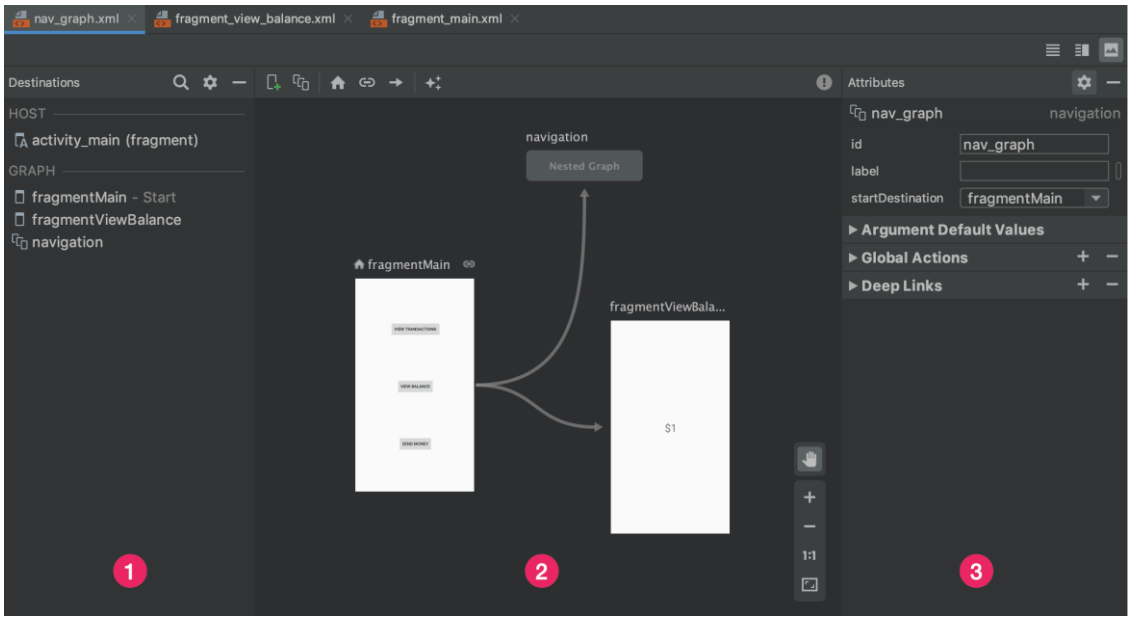
- fiksna početna destinacija
- stanje navigacije predstavljeno je kao stog destinacija
- gumb za povratak u *action bar*-u (traka na vrhu ekrana) te gumb za povratak u navigacijskoj traci na dnu ekrana imaju istu funkciju
- gumb za povratak u *action bar*-u ne izlazi upotpunosti iz aplikacije
- dubinsko povezivanje (eng. *deep linking*) simulira ručnu navigaciju.

Navigation se sastoji od tri ključna dijela (Android, 2022f):

- **navigacijski grafikon** – XML koji sadrži sve vezane informacije za navigaciju u jednoj centraliziranoj lokaciji. To uključuje sva pojedinačna mjesta sadržaja unutar aplikacije, odnosno destinacije (eng. *Destinations*), kao i moguće puteve kojima korisnik može prolaziti kroz aplikaciju.
- **NavHost** – Prazni spremnik koji prikazuje odredišta iz navigacijskog grafikona. Komponenta *Navigation* sadrži zadanu *NavHost* implementaciju, *NavHostFragment* koji prikazuje odredišta fragmenata.
- **NavController** – objekt koji upravlja navigacijom aplikacije unutar *NavHost*-a. *NavController* upravlja izmjenom odredišnog sadržaja u *NavHost* kako korisnici prolaze kroz aplikaciju.

Za implementaciju navigacije, prvo se mora kreirati navigacijski graf koji se kreira kao Android datoteka resursa (eng. *Android resource File*), nakon toga Android studio otvara graf u editoru navigacije (eng. *Navigation Editor*).

Slika 21: Editor navigacije



(Izvor: Android, 2022f)

Sastoji se od:

1. **panela destinacija** gdje je vidljiv domaćin navigacije (eng. *navigation host*) i sve destinacije koje su dodane u editor grafa
2. **editor grafa** koji sadrži vizualni prikaz navigacijskog grafa. Sastoji se od prikaza dizajna i XML reprezentacije.
3. **atributa** – koji prikazuju atribute za trenutno označenu stavku u navigacijskom grafu.

Izgled XML datoteka navigacije je sljedeći:

Slika 22: XML datoteka navigacije

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
            xmlns:app="http://schemas.android.com/apk/res-auto"
            android:id="@+id/nav_graph">

</navigation>
```

(Izvor: Android, 2022f)

Nakon kreiranja navigacijskog grafa potrebno je dodati domaćina navigacije (eng. *navigation host*) u glavnu aktivnost. Domaćin navigacije je prazan spremnik u kojoj se destinacije izmjenjuju kako se korisnik kreće kroz aplikaciju. Glavna implementacija

NavHost-a iz kojeg proizlazi domaćin navigacije je *NavHostFragment* koji upravlja zamjenom destinacijama fragmenata. Ispod se nalazi primjer XML koda gdje je prikazan *NavHostFragment* kao dio glavne aktivnosti aplikacije:

Slika 23: NavHostFragment u glavnoj aktivnosti

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    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"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <androidx.appcompat.widget.Toolbar
        .../>

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/nav_host_fragment"
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="0dp"
        android:layout_height="0dp"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"

        app:defaultNavHost="true"
        app:navGraph="@navigation/nav_graph" />

    <com.google.android.material.bottomnavigation.BottomNavigationView
        .../>

</androidx.constraintlayout.widget.ConstraintLayout>
```

(Izvor: Android, 2022f)

Potom je moguće dodavati destinacije koje se kreiraju iz već postojećih fragmenata ili aktivnosti, a dodaju se pomoću opcije za dodavanje nove destinacije.

Sljedeći kod podebljanim slovima prikazuje primjer destinacije u navigacijskom grafu:

Slika 24: Primjer destinacije u navigacijskom grafu

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android"
    app:startDestination="@id/blankFragment">
    <fragment
        android:id="@+id/blankFragment"
        android:name="com.example.cashdog.cashdog.BlankFragment"
        android:label="@string/label_blank"
        tools:layout="@layout/fragment_blank" />
</navigation>
```

(Izvor: Android, 2022f)

Sastoji se od identifikacije koja je jedinstvena te upućuje na destinaciju, imena koje sadrži ime cijele klase fragmenta, oznake koja predstavlja riječ za identificiranje fragmenta. Destinacije olakšavaju odrediti početnu točku u aplikaciji i kretanje kroz aplikaciju. (Android, 2022f)

Kretanje kroz aplikaciju vrši se pomoću *NavController*-a, objekta koji upravlja navigacijom unutar *NavHost*-a. Svaki *NavHost* ima pripadajući *NavController* koji se može dohvatiti sljedećim metodama u Javi (Android, 2022f):

- NavHostFragment.findNavController(Fragment)
- Navigation.findNavController(Activity, @IdRes int viewId)
- Navigation.findNavController(View)

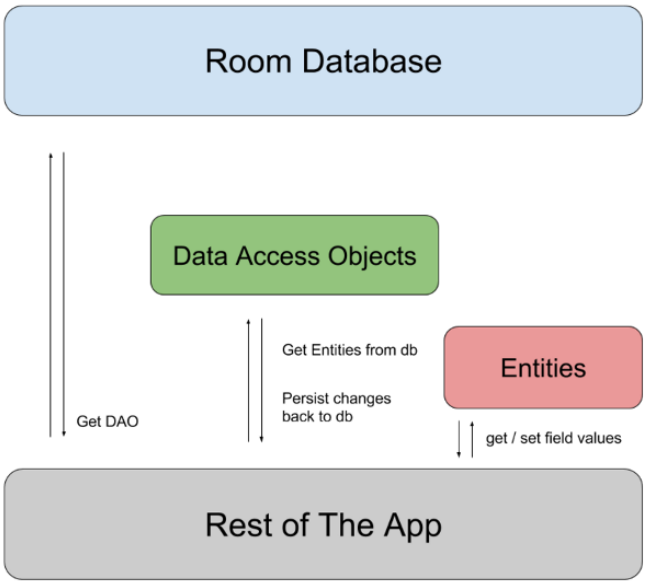
Komponenta pruža velik broj pogodnosti kao što su rukovanje transakcijama fragmenata, ispravno rukovanje radnjama naprijed i natrag, pruža standardizirane resurse za animacije i prijelaz, implementaciju i rukovanje dubinskim povezivanjem, uključivanje obrasce navigacije korisničkog sučelja (kao što su navigacijske ladice i donji meni), omogućava dijeljenje podataka vezanih za korisničko sučelje unutar destinacija pomoću *ViewModel*-a. (Android, 2022f)

4.2.4. Room

Room je biblioteka koja omogućava sloj apstrakcije iznad SQLite-a za robusniji pristup bazi podataka pritom iskorištavajući punu snagu SQLite-a. (Android, 2022g)

Neke od prednosti korištenja Room baze su pogodne anotacije koje minimiziraju kod sklon pogreškama i kod sa ponavljajućim linijama (eng. *boilerplate code*), omogućava jednostavan rad sa *LiveData* i *RxJava* za promatranje podataka, omogućava verifikaciju SQL upita tijekom kompiliranja. (Android, 2022g)

Slika 25: Prikaz komunikacije *Room* baze sa ostatkom aplikacije



(Izvor: Android, 2022g)

Tri glavne komponente Room baze su:

- entitet
- objekt za pristup podacima (DAO)
- baza podataka

Entitet – predstavlja tablice u bazi podataka aplikacije.

Slika 26: Primjer entiteta

```
@Entity
public class User {
    @PrimaryKey
    public int uid;

    @ColumnInfo(name = "username")
    public String firstName;
}
```

(Izvor: Android, 2022g)

Objekt za pristup podacima (DAO) – sadrži metode kojima se u aplikaciji mogu vršiti upiti, umetanje, ažuriranje i brisanje u bazi podataka

Slika 27: Primjer Dao-a

```
@Dao
public interface UserDao {

    @Insert
    void insertAll(User... users);

    @Query("SELECT * FROM user")
    List<User> getAll();
}
```

(Izvor: Android, 2022g)

Baza podataka – klasa koja drži bazu podataka i služi kao glavna pristupna točka za povezivanje sa postojanim podacima.

Slika 28: Primjer baze podataka

```
@Database(entities = {User.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
    public abstract UserDao userDao();
}
```

(Izvor: Android, 2022g)

Nakon definiranja entiteta, DAO-a i objekta baze, kreira se prvo instanca baze te se zatim koriste apstraktne metode iz *AppDatabase* za dohvat instance DAO-a iz kojeg se koriste metode za interakciju sa bazom podataka.

Slika 29: Primjer dohvata podataka iz *Room* baze

```
AppDatabase db = Room.databaseBuilder(getApplicationContext(),
        AppDatabase.class, "database-name").build();

UserDao userDao = db.userDao();
List<User> users = userDao.getAll();
```

(Izvor: Android, 2022g)

4.3. Arhitektura aplikacije

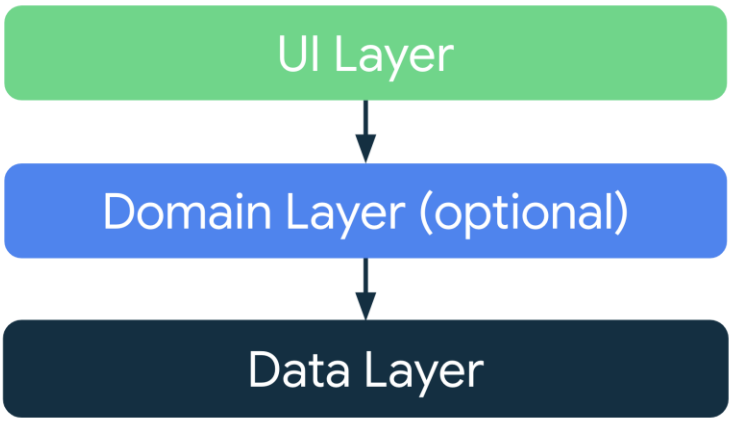
Važno je definirati arhitekturu koja omogućuje skaliranje aplikacije, povećava robusnost aplikacije i olakšava testiranje aplikacije. Arhitektura aplikacije definira granice između dijelova aplikacije i odgovornosti koje bi svaki dio aplikacije trebao imati. (Android, 2022c)

Aplikacija bi trebala imati minimalno dva sloja:

- **Sloj korisničkog sučelja** – prikazuje podatke aplikacije na ekranu
- **Podatkovni sloj** - sadrži poslovnu logiku aplikacije i izlaže podatke aplikacije

Može se dodati dodatni sloj koji se naziva sloj domene. To je sloj koji se nalazi između sloja korisničkog sučelja i podatkovnog sloja te je odgovaran za enkapsulaciju složene poslovne logike ili jednostavne poslovne logike koju više puta koriste više *ViewModel*-a. (Android, 2022c)

Slika 30: Dijagram arhitekture aplikacije



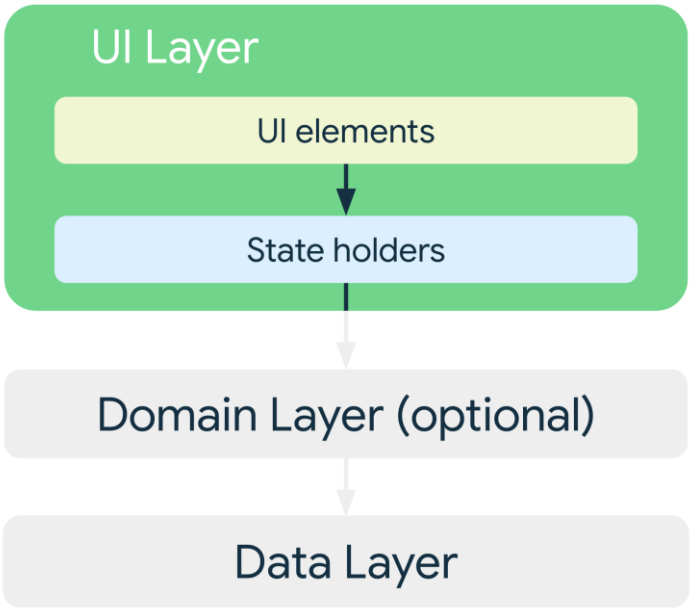
(Izvor: Android, 2022c)

Sloj korisničkog sučelja (ili prezentacijski sloj) ima ulogu prikazivanja podataka aplikacije na zaslon. Kada se podaci promjene, bez obzira radi li se o korisničkog interakciji (npr. pritisak gumba) ili vanjskog unosa (kao što je odgovor mreže), korisničko sučelje trebalo bi se ažurirati kako bi se odrazile promjene. (Android, 2022c)

Sloj korisničkog sučelja sastoji se od dva dijela (Android, 2022c):

- **Elemenata korisničkog sučelja** koji prikazuju podatke na zaslonu. Ti elementi se grade korištenjem pogleda (eng. *Views*) ili *Jetpack Compose* funkcija.
- **Držača stanja** (eng. *state holders*) kao što su *ViewModel* klase, koji drže podatke, izlažu ih korisničkom sučelju i rukuju logikom.

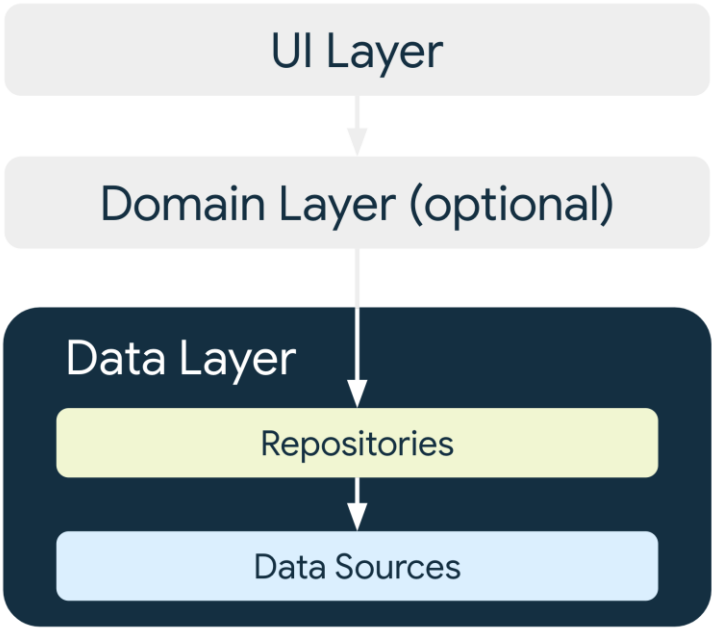
Slika 31: Sloj korisničkog sučelja



(Izvor: Android, 2022c)

Podatkovni sloj sadrži poslovnu logiku. Poslovna logika je ono što daje vrijednost aplikaciji – sadrži pravila koja određuju kako aplikacija stvara, sprema te mijenja podatke. Podatkovni sloj sastoji se od spremišta (eng. *repositories*) od kojih svaki može sadržavati nula i više izvora podataka. Sve različite vrste podataka kojima se rukuje u aplikaciji trebale bi imati svoju klasu repozitorija. Na primjer, klasa FilmoviRepository za podatke povezanim s filmovima ili klasu PlacanjaRepository za podatke povezanim s plaćanjima. (Android, 2022c)

Slika 32: Prikaz podatkovnog sloja



(Izvor: Android, 2022c)

Repozitoriji su odgovorni izlaganje podataka ostatku aplikacije, centraliziranje promjena podataka, rješavanje konflikta između više izvora podataka, apstrahiranje izvora podataka od ostatka aplikacije, posjedovanje poslovne logike. (Android, 2022c)

Svaki klasa izvora podataka trebala bi raditi samo s jednim izvorom podataka, na primjer datoteka, mrežni izvor ili lokalna baza podataka. Klase izvora podataka služi kao most između aplikacije i sustava za rad s podacima. (Android, 2022c)

4.4. Pohrana podataka na Oblak

Pohrana podataka na Oblak omogućava izradu sigurnosne kopije podataka u slučaju katastrofe (kao što je kvar hardvera, gubitak uređaja i slično) te pristup podacima sa različitih uređaja. Firebase je platforma tvrtke Google koja se koristi za razvoj aplikacija. Neke od istaknutih mogućnosti platforme su baza podataka, autentifikacija, analitika, pohrana podataka. Za pohranjivanje podataka na Oblak može se koristiti *Firebase Realtime Database*. (Firebase, 2022)

Firebase Realtime Database je NoSQL baza podataka smještena u oblaku. Podaci se pohranjuju kao JSON i sinkroniziraju u stvarnom vremenu sa svakim povezanim klijentom. (Firebase, 2022)

Kako bi se mogao koristiti *Firebase Realtime Database* potrebno je spajanje Firebase-a na Android projekt. To je moguće je kreiranjem novog projekta na konzoli Firebase-a i dodavanja ovisnosti (eng. *dependencies*) i Google konfiguracijske datoteke, no može se i jednostavno dodati Firebase podrška Android projektu izravno iz Android Studio-a. Detaljnija konfiguracija opisana je u službenoj dokumentaciji Firebase-a. (Kumar, 2018)

Prije pisanja ili čitanja podataka, potrebna je referenca na bazu podataka. Svaki Firebase projekt ima svoje stavke *Realtime* baze podataka koje mogu biti pregledane otvaranjem projekta unutar Firebase konzole te odabira opcije za prikaz baze podataka. Unutar konzole, nalaze se paneli koji mogu biti odabrani za prikaz stabala podataka u bazi podataka, pravila za dohvat podataka, procjene upotrebe baze podataka i ostalih mogućnosti. *Firebase Realtime Database* obično su krajnja točka REST (reprezentacijsko stanje prijenosa) referenci koje se koriste za dodavanje podataka. (Kumar, 2018)

Slika 33: Kreiranje reference na bazu podataka

```
FirebaseDatabase mDatabase = FirebaseDatabase.getInstance();
DatabaseReference mDbRef = mDatabase.getReference("User/Name");
```

(Izvor: Mastering Firebase for Android Development, 2018)

Za pisanje u bazu potrebno je dohvatiti instancu baze podataka koristeći getInstance() te referencirati lokaciju gdje treba pisati. Moguće je pisati većinu primitivnih tipova podataka, s obzirom da oni također uključuju Java objekte. (Kumar, 2018) U primjeru ispod prikazano je dodavanje novog korisnika, sadržaj Java objekta (u ovom slučaju korisnika) preslikava se na podređene lokacije na ugniježđeni način. (Firebase, 2022)

Slika 34: Pisanje podataka u Realtime bazu podataka

```
  public void writeNewUser(String userId, String name, String email) {
      User user = new User(name, email);

      mDatabase.child("users").child(userId).setValue(user);
  }
```

(Izvor: Firebase, 2022)

S obzirom da Realtime baza podataka sinkronizira sve podatke u stvarnom vremenu na svim platformama i uređajima, za čitanje podataka koristi se povratni poziv onDataChange*()*, metoda slušača za promjene ValueEventListener-a. (Firebase, 2022)

Slika 35: Čitanje podataka iz Realtime baze podataka

```
ValueEventListener postListener = new ValueEventListener() {
    @Override
    public void onDataChange(DataSnapshot dataSnapshot) {
        Post post = dataSnapshot.getValue(Post.class);
    }

    @Override
    public void onCancelled(DatabaseError databaseError) {
        Log.w(TAG, "loadPost:onCancelled", databaseError.toException());
    }
};
mPostReference.addValueEventListener(postListener);
```

(Izvor: Firebase, 2022)

Korištenje *Firebase Realtime Database* omogućava pohranu podataka na Oblak bez upravljanja serverima te pristup podacima sa bilo kojeg uređaja. *Realtime Database* se integrira sa Firebase autentifikacijom što omogućava jednostavnu i intuitivnu autentifikaciju za programere.

4.5. Testiranje aplikacije

Testiranje je sastavni dio procesa razvoja aplikacije. Testiranjem Android aplikacije testiraju se njene funkcionalnosti, upotrebljivost i kompatibilnost aplikacije na Android uređajima. Aplikacije sklone pogreškama mogu značajno utjecati na korisničko iskustvo i dovesti do pada zadovoljstva i ocjene aplikacije te se iz toga razloga preporučuje dostatno testiranje aplikacije prije nego se pusti u pogon. (Kong, 2018)

Testiranje aplikacije se može podijeliti na manualno i automatizirano testiranje.

Manualno testiranje vrši se kretanjem kroz aplikaciju i pritom prolaze različiti korisnički tokovi, generiraju moguće pogreške, koriste različiti uređaji i emulatori, međutim takvo testiranje može biti naporno, dugotrajno i sklono pogreškama. (Kong, 2018; Android, 2022b)

Automatizirano testiranje uključuje korištenje alata koji sami izvode testiranje koje je efektivnije, brže, snažnije i pouzdanije. (Android, 2022b)

Postoje razne vrste testova koji ovise o tome što je subjekt testiranja: testiranje funkcionalnosti, performansi, testiranje pristupačnosti i testiranje kompatibilnosti. (Android, 2022b)

Također, testovi mogu varirati prema veličini, odnosno stupnju izolacije (Android, 2022b):

- **Jedinični testovi** (eng. *unit tests*) – provjeravaju mali dio aplikacije, kao što je klasa ili metoda.
- **Krajnji** (eng. *end-to-end*) ili **veliki testovi** – provjeravaju velike dijelove aplikacije u isto vrijeme, kao što je cijeli ekran ili korisnički tok
- **Srednji testovi** (eng. *medium tests*) - nalaze se u sredini te provjeravaju integraciju između dvije ili više jedinica.

Testovi za Android aplikacije dijele se na lokalne i instrumentalne testove.

Lokalni testovi - izvršavaju se na razvojnom stroju ili serveru. Takvi testovi često su manji i brzi, izolira se subjekt testiranja od ostatka aplikacije. Lokalni testovi mogu biti jedinični testovi, integracijski testovi i simulatori. (Android, 2022b)

Slika 36: Primjer dijela jediničnog testa za ViewModel

```
// instanca MyViewModel-a
MyViewModel viewModel = new MyViewModel(fakeDataRepository);

// kad se učitaju podaci
viewModel.loadData();

// onda bi podaci trebali biti vidljivi
assertTrue(viewModel.data() != null);
```

(Izvor: Android, 2022b)

Instrumentalni testovi – izvode se na uređaju ili emulatoru. U pozadini se instalira aplikacija, te se zatim i testna aplikacija instalira koja kontrolira aplikaciju, pokreće je i izvodi testove korisničkog sučelja. Instrumentalni testovi se također mogu koristiti i za testiranje logike bez korisničkog sučelja. Korisni su kada se treba testirati kod koji ovisi o kontekstu aplikacije. (Android, 2022b)

Slika 37: Komunikacija sa korisničkim sučeljem u instrumentalnom testu

```
// Kada se klikne na gumb Dalje
onView(withText("Dalje"))
    .perform(click())

// Prikaže se zaslon dobrodošlice
onView(withText("Dobro došli"))
    .check(matches(isDisplayed()))
```

(Izvor: Android, 2022b)

U idealnom svijetu testirao bi se svaki redak koda u aplikaciji na svakom uređaju s kojim je aplikacija kompatibilna, međutim takav proces je prespor i skup te je potrebno odrediti što se treba testirati. Određivanje što se treba testirati ovisi o više čimbenika: tipu aplikacije, razvojnom timu, količini koda te korištenoj arhitekturi. Testiranje je bitan element ukoliko se želi graditi kvalitetna aplikacija. (Android, 2022b)

5. Analiza i razrada funkcionalnosti aplikacije

Mobilna aplikacija za potporu u praćenju dijabetesa služila bi osobama oboljelima od dijabetesa kako bi na dnevnoj bazi mogli pratiti svoje aktivnosti kao što su unos terapije, hrane i fizičke aktivnosti.

Dio aplikacije omogućava ručni unos terapije inzulinom (vrijeme, doza i tip unosa: oralno/intravenski/kontinuirano) te omogućava pregled unesenih terapija na dnevnoj bazi.

Drugi dio aplikacije omogućava unos hrane: tip hrane, količina i vrijeme unosa. Na temelju prethodno unesenih stavki korisniku se prikazuju i separirani ugljikohidrati i proračun kalorijske vrijednosti obroka.

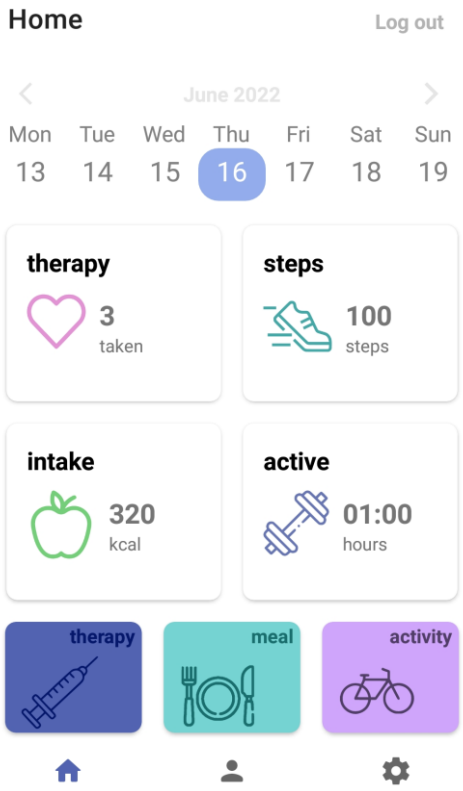
Te jedan dio aplikacije omogućava korisniku praćenje fizičke aktivnosti ručnim unosom tipičnih sportskih aktivnosti kao što su vožnja biciklom, trčanje, vježbanje, joga i slično te praćenje korištenjem pedometra. Korištenje pedometra omogućava dnevno brojanje koraka koje korisnik učini. Opcija se može aktivirati automatski ulaskom u aplikaciju ili ručnim pokretanjem opcije praćenja koraka. Na temelju prethodnog unosa podataka korisniku se omogućava dnevni prikaz broja koraka i ukupno vrijeme ostale fizičke aktivnosti.

Sa aspekta korisnika sustava, aplikacijom će se koristiti korisnik kojemu je potrebno praćenje aktivnosti vezane uz dijabetes na dnevnoj bazi.

5.1. Korisničko sučelje mobilne aplikacije

Početni prozor sadrži pregled terapija, učinjenih koraka, unosa kalorija hrane te vrijeme fizičke aktivnosti po danima u tjednu. Klikom na određeni pregled otvara se novi prozor sa detaljnijim prikazom. Na primjer, klikom na pregled terapije otvara se prozor za detaljniji prikaz terapije. Postoje tri gumba s kojima korisnik ima mogućnost otvaranja novih prozora za unos terapije, hrane ili fizičke aktivnosti. Na dnu se nalazi navigacija za početni zaslon, profil korisnika te postavke.

Slika 38: Prikaz korisničkog sučelja mobilne aplikacije

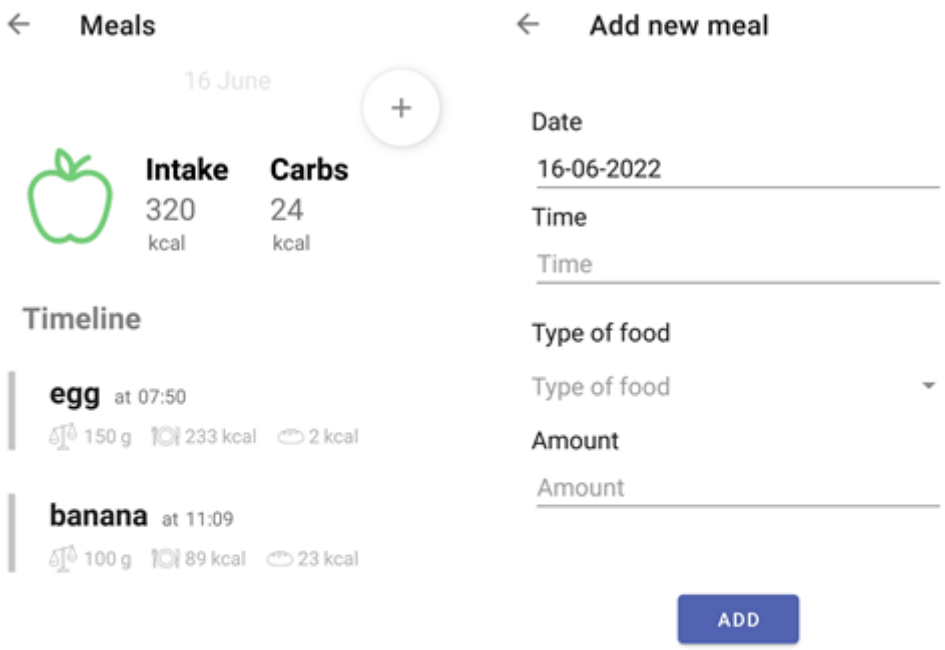


(Izvor: autor)

Prozor za detaljniji prikaz, na primjer obroka, sadrži ukupan dnevni unos kalorija i količinu ugljikohidrata te pregled obroka vremenskom linijom; uključujući tip hrane, vrijeme unosa, količina, kalorijska vrijednost te postotak ugljikohidrata.

Prozor za dodavanje nove stavke, kao što je obrok, sadrži polja za unos datuma, vrijeme, tip hrane te količinu. Nakon dodavanja novog obroka, generiraju se dodatni podaci: kalorijska vrijednost te postotak ugljikohidrata te se kreirani obrok može vidjeti na prozoru za detaljniji prikaz obroka.

Slika 39: Prozori za detaljniji prikaz i dodavanje nove stavke



(Izvor: autor)

5.2. Scenariji

Scenarij je niz koraka koji se izvršavaju otpočetka do kraja pojedinog slučaja korištenja. Scenariji u ovom slučaju ilustriraju bilježenje unosa podataka hrane koji se pohranjuju lokalno te unos podataka terapije i slanje podataka na Cloud.

5.2.1. Scenarij bilježenja unosa podataka

Korisnik želi zabilježiti novi unos podataka o obroku. Kako bi podaci bili zabilježeni, korisnik prvo unosi osnovne podatke kao što su vrijeme unosa i količina te odabire tip hrane. Nakon što je korisnik potvrdio unos podataka o obroku, generiraju se dodatni podaci (izračun kalorija, separacija ugljikohidrata) i svi podaci se pohranjuju u bazu.

Slika 40: Dijagram aktivnosti bilježenja unosa hrane

(Izvor: autor)

5.2.2. Scenarij bilježenja unosa podataka i slanja na Cloud

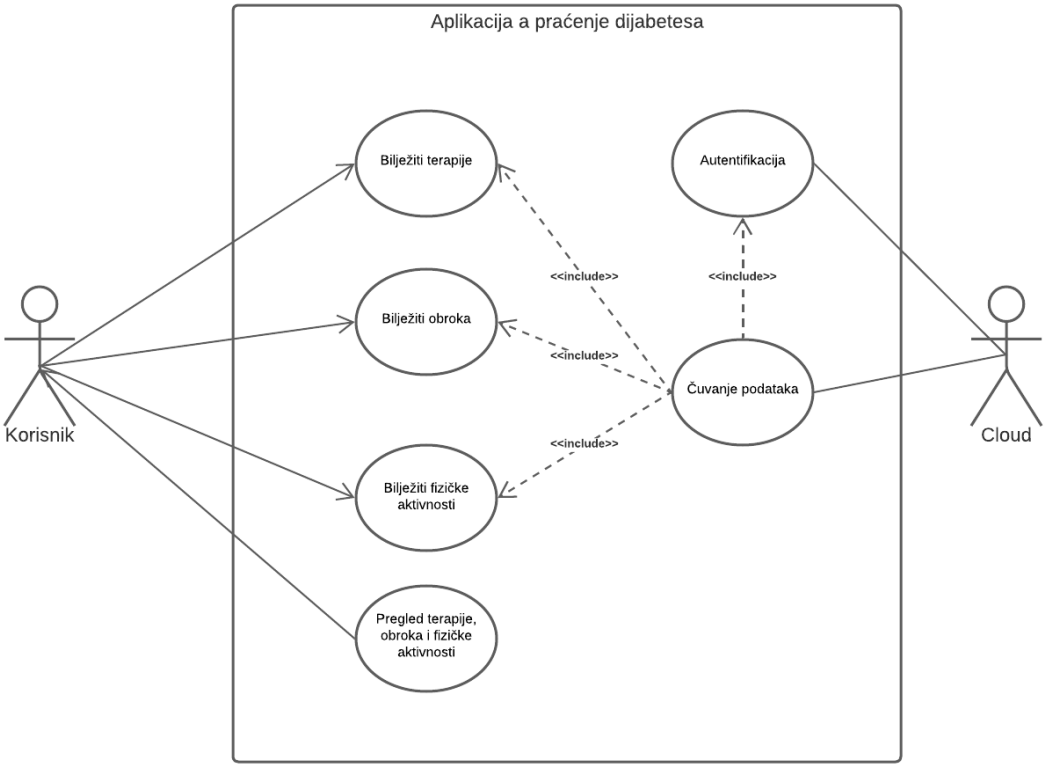
Korisnik želi unesti podatke o terapiji i pritom ih spremiti na Cloud. Korisnik se prvo prijavljuje u svoj račun, zatim unosi podatke o terapiji kao što su vrijeme unosa, doza i tip unosa. Nakon što korisnik unese potrebne podatke, korisnik potvrđuje unos te se zatim podaci šalju na Cloud i pospremaju lokalno u bazu.

Slika 41: Dijagram aktivnosti unosa podataka za slanje na Cloud

(Izvor: autor)

5.3. Dijagram slučaja uporabe

Slika 42: Dijagram slučajeva korištenja mobilne aplikacije za potporu u praćenju dijabetesa



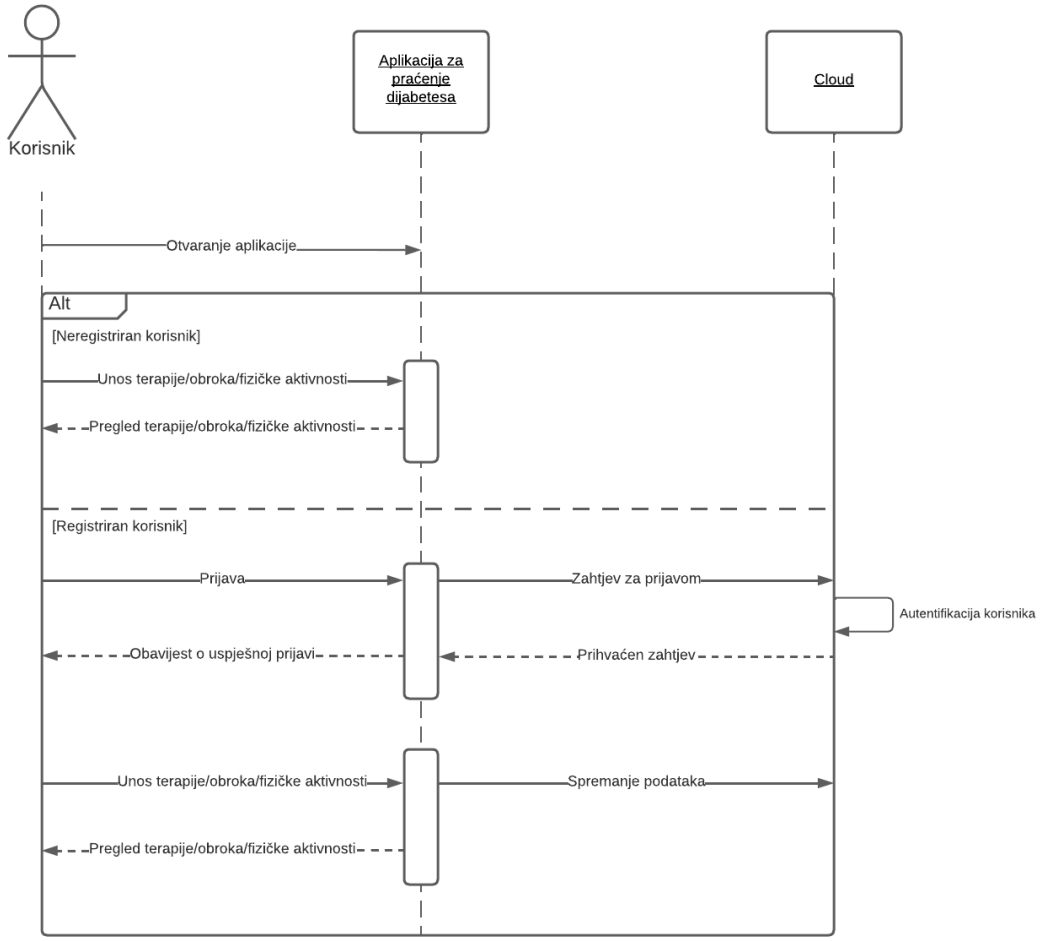
(Izvor: autor)

Na prikazanoj slici dijagrama slučajeva korištenja prikazane su funkcionalnosti koje će korisnik moći izvršiti unutar aplikacije. Funkcionalnosti koje korisnik ima su: bilježenje terapija, obroka, fizičkih aktivnosti te pregled terapija, obroka i fizičke aktivnosti. Za funkcionalnost čuvanja podataka potrebna je autentifikacija.

5.4. Sekvencijalni dijagram

Sekvencijalnim dijagramom prikazano je kada i kako objekti međusobno komuniciraju.

Slika 43: Sekvencijalni dijagram mobilne aplikacije za potporu u praćenju dijabetesa



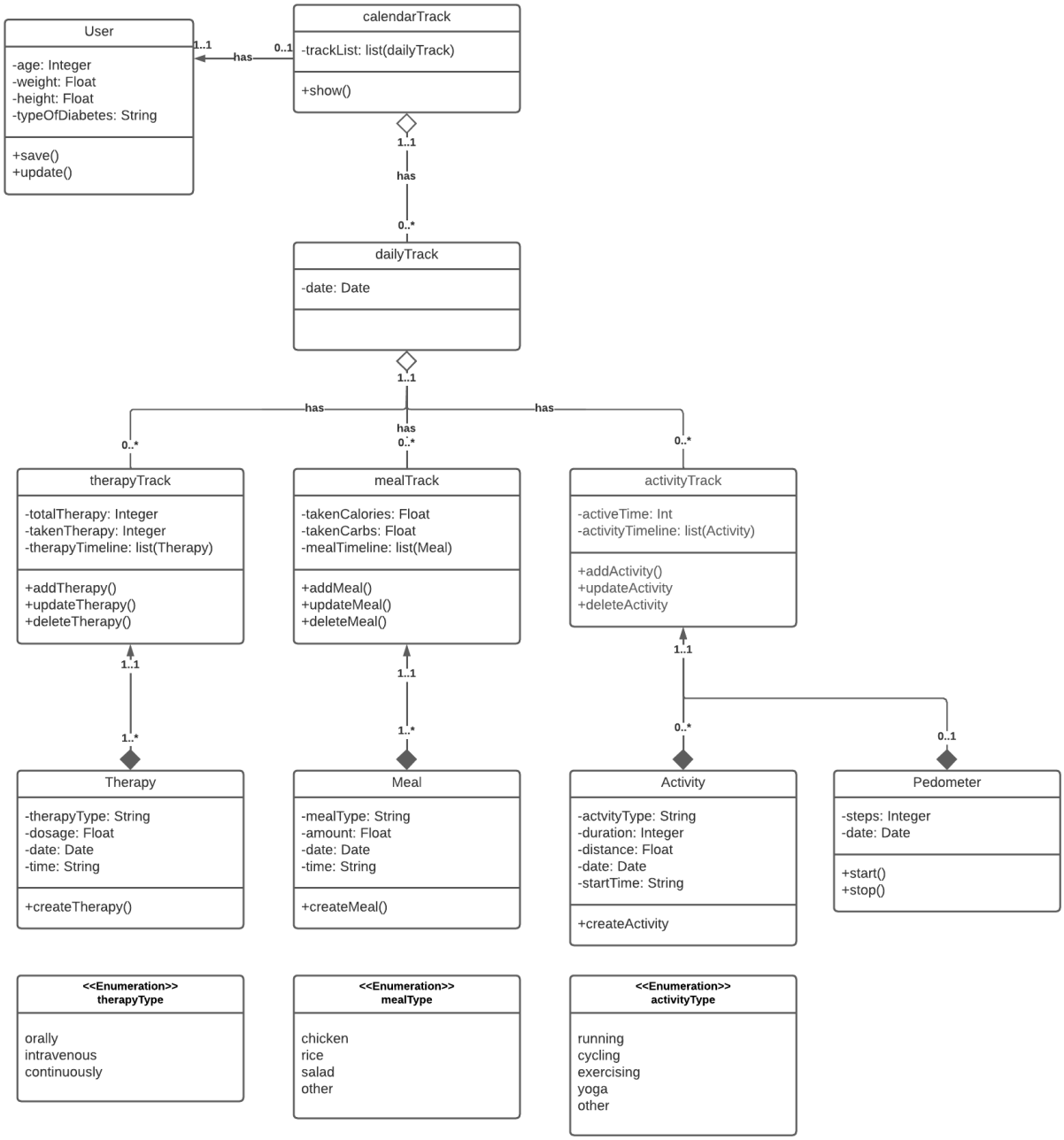
(Izvor: autor)

Nakon otvaranja aplikacije, postoje dva scenarija za korisnika. U prvom scenariju neregistrirani korisnik unosi terapiju, obrok ili fizičku aktivnost i zatim ih može pregledati. U drugom scenariju, registrirani korisnik se prijavljuje u aplikaciji i šalje se zahtjev za prijavom na *Cloud*, nakon autentifikacije korisnika šalje se povratna informacija da je zahtjev prihvaćen te korisnik dobiva obavijest o uspješnoj prijavi. Zatim registrirani korisnik unosi terapiju, obrok ili fizičku aktivnost te se podaci spremaju na *Cloud* i korisnik može pregledati terapije, obroke ili fizičke aktivnosti.

5.5. Dijagram klasa

Dijagramom klasa opisane su klase i veze između njih kako bi se objasnio dizajn i model aplikacije.

Slika 44: Dijagram klasa mobilne aplikacije za potporu u praćenju dijabetesa



(Izvor: autor)

Svaka klasa ima dio s pretpostavljenim funkcijama koje bi trebalo koristiti u samoj implementaciji. Dijagram klasa sadrži i enumeracije koje označavaju podatke koje će korisnik moći unositi odabirom nekih od zadanih podataka. Korisnik neće moći unesti drugi podatak koji nije na popisu u enumeraciji. Na primjer, unutar enumeracije TherapyType korisnik će moći unesti samo vrijednosti oralno, intravenozno i kontinuirano koje označavaju način unosa terapije.

Dijagram se sastoji od sedam klasa koje su međusobno povezane agregacijskim i kompozicijskim vezama. Agregacijska veza je slaba veza i u takvoj vezi podklasa može postojati i bez nadklase. Agregacijska veza na strani podklase označena je bijelim rombom. Kompozicijska veza je jaka veza te u toj vezi podklasa ne može postojati bez nadklase. Kompozicija u nadklasi označena je crnim rombom na strani podklase.

Prikazom interakcija između objekata omogućeno je bolje razumijevanje odgovornosti objekata i ponašanja koja trebaju imati.

6. Implementacija u programski kod

U ovom poglavlju bit će objašnjena implementacija same aplikacije. Prikazan je i objašnjen kod za dodavanje, uređivanje i brisanje podataka terapije, praćenje broja koraka, učitavanje te promjena jezika.

Za umetanje, uređivanje i brisanje koristi se više klasa koje su međusobno povezane.

Prva klasa koja je definirana je POJO (eng. *plain old Java object*) klasa koja je običan Java objekt i predstavlja entitet. Potrebno ga je označiti kao entitet te definirati primarni ključ. Polja koja su definira predstavljaju redove tablice u bazi podataka.

Slika 45: Entitet za terapiju

```
@Entity(tableName = "therapy")
public class Therapy {

    @PrimaryKey(autoGenerate = false)
    private int id;

    private String type;
    private Double dosage;
    private String time;
    private String date;

    public Therapy(String type, Double dosage, String time, String date) {
        this.type = type;
        this.dosage = dosage;
        this.time = time;
        this.date = date;
    }

    public void setId(int id) { this.id = id; }

    public int getId() { return id; }

    public String getType() { return type; }

    public Double getDosage() { return dosage; }

    public String getTime() { return time; }

    public String getDate() { return date; }
}
```

(Izvor: autor)

Zatim se definira objekt za pristup podacima, skraćeno DAO (eng. *data access object*) koji je ustvari sučelje anotiran sa „Dao“. Ovdje se definiraju operacije baze podataka, točnije CRUD (eng. *create, read, update, delete*) operacije za kreiranje, čitanje, ažuriranje i brisanje podataka. Osim anotacije sučelja sa Dao, potrebno je anotirati i CRUD operacije.

Slika 46: DAO za terapiju

```
@Dao
public interface TherapyDao {
    @Insert
    void insert(Therapy therapy);

    @Update
    void update(Therapy therapy);

    @Delete
    void delete(Therapy therapy);

    @Query("SELECT * FROM therapy ORDER BY time DESC")
    LiveData<List<Therapy>> getAllTherapies();

    @Query("SELECT * FROM therapy WHERE date==:date ORDER BY time DESC")
    LiveData<List<Therapy>> getDayTherapies(String date);
}
```

(Izvor: autor)

Potom se definira klasa za bazu podataka u kojoj se definira ime baze podataka uz reference na tablice i DAO. Ova klasa proširuje se na klasu RoomDatabase, osnovnu klasu za sve *Room* baze. Potrebno je proširenje na RoomDatabase, nakon što se klasa anotira kao baza podataka.

Slika 47: Baza podataka

```
@Database(entities = Therapy.class, version = 1)
public abstract class AppDatabase extends RoomDatabase {
    public abstract TherapyDao therapyDao();
    private static AppDatabase instance;
    public static synchronized AppDatabase getInstance(Context context) {
        if (instance == null) {
            instance = Room.databaseBuilder(context.getApplicationContext(),
                    AppDatabase.class, "app_database").addCallback(roomCallback)
                    .build();
        }
        return instance;
    }
    private static RoomDatabase.Callback roomCallback = new RoomDatabase.Callback()    {
        @Override
        public void onCreate(@NonNull @NotNull SupportSQLiteDatabase db) {
            super.onCreate(db);
            new PopulateDbAsyncTask(instance).execute();
        }
    };
}
```

(Izvor: autor)

Kada se završi sa kreiranjem klase za bazu podataka, potrebno je definirati klasu koja služi kao repozitorij. Glavna funkcija je spajanje *ViewModel*-a sa izvorom podataka, u ovom slučaju *Room* modelom. Operacije se izvršavaju pomoću izvršitelja (eng. *executors*).

Slika 48: Repozitorij za terapiju

```
public class TherapyRepository {

    private TherapyDao therapyDao;
    private LiveData<List<Therapy>> allTherapies;

    private Executor executor = Executors.newSingleThreadExecutor();

    public TherapyRepository(Application application){
        AppDatabase database = AppDatabase.getInstance(application);
        therapyDao = database.therapyDao();
        allTherapies = therapyDao.getAllTherapies();
    }

    public void insertTherapy(Therapy therapy){
        executor.execute(new Runnable() {
            @Override
            public void run() {
                therapyDao.insert(therapy);
            }
        });
    }


    public void updateTherapy(Therapy therapy){
        executor.execute(new Runnable() {
            @Override
            public void run() {
                therapyDao.update(therapy);
            }
        });
    }

    public void deleteTherapy(Therapy therapy){
        executor.execute(new Runnable() {
            @Override
            public void run() {
                therapyDao.delete(therapy);
            }
        });
    }

    public LiveData<List<Therapy>> getAllTherapies(){
        return allTherapies;
    }

    public LiveData<List<Therapy>> getDayTherapies(String date) {
        return therapyDao.getDayTherapies(date);
    }

}
```

(Izvor: autor)

Nakon kreiranja repozitorija, kreira se *ViewModel* klasa koja je odgovorna za pripremu i upravljanje podacima za aktivnost ili fragment. Također upravlja komunikacijom aktivnosti ili fragment s ostatkom aplikacije. Kreira se klasa koja proširuje *AndroidViewModel*, proširuje se zbog potrebe za korištenjem konteksta aplikacije. Zatim se kreira objekt klase repozitorij. Definira se konstruktor kojemu je parametar aplikacija.

Slika 49: ViewModel sa dijelom koda za terapiju

```
public class DayViewModel extends AndroidViewModel {

    private TherapyRepository therapyRepo;
    private LiveData<List<Therapy>> dayTherapies;
    private SingleLiveEvent<Therapy> sTherapy = new SingleLiveEvent<>();

    private MutableLiveData<String> date = new MutableLiveData<>();

    public DayViewModel(@NonNull @NotNull Application application) {
        super(application);

        therapyRepo = new TherapyRepository(application);

        dayTherapies = Transformations.switchMap(date,
                date -> therapyRepo.getDayTherapies(date));
    }

    public void setDate(String newDate) { date.setValue(newDate);}
    public LiveData<String> getDate() { return date;}

    public LiveData<List<Therapy>> getDayTherapies() { return dayTherapies; }

    public void setSTherapy(Therapy therapy) { sTherapy.setValue(therapy); }
    public SingleLiveEvent<Therapy> getSTherapy() { return sTherapy; }

    public void insertTherapy(Therapy therapy) { therapyRepo.insertTherapy(therapy);}
    public void updateTherapy(Therapy therapy){ therapyRepo.updateTherapy(therapy); }
    public void deleteTherapy(Therapy therapy){ therapyRepo.deleteTherapy(therapy); }

}
```

(Izvor: autor)

Konačno se može referencirati *ViewModel* u prikazu (aktivnost ili fragment) te pozvati funkcije za određene operacije.

Slika 50: Referenciranje ViewModela i poziv funkcija u prikazu

```
dayViewModel = new ViewModelProvider(requireActivity()).get(DayViewModel.class);


dayViewModel.getDayTherapies().observe(getActivity(), new Observer<List<Therapy>>() {
    @Override
    public void onChanged(List<Therapy> therapies) {
        therapyAdapter.setTherapies(therapies);
    }
});


buttonAdd.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        dayViewModel.insertTherapy(therapy);
    }
});
```

(Izvor: autor)

Brojanje koraka izvršava se pomoću dvije metode: onStartCommand() i onSensorChanged().

Metodu onStartCommand() poziva sustav pozivanjem metode startService() kada druga komponenta (u ovom slučaju aktivnost) zahtjeva pokretanje usluge. Dohvaća se senzor za brojanje koraka te ukoliko postoji senzor, registrira se slušač i pokrene notifikacija kojom usluga, odnosno praćenje koraka postaje vidljivo korisniku.

Slika 51: Metoda za registriranje slušača senzora

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
    sensor = sensorManager.getDefaultSensor(Sensor.TYPE_STEP_COUNTER);

    if(sensor != null){
        sensorManager.registerListener(this, sensor,
SensorManager.SENSOR_DELAY_NORMAL);
        showNotification();
    }

    return START_STICKY;
}
```

(Izvor: autor)

Metoda onSensorChanged(), metoda je koja se koristi za praćenje broja koraka. Metoda prvo dohvaća ukupan broj koraka otkako je mobitel upaljen, zatim u prvom uvjetu provjerava je li današnji datum jednak privremeno pospremljenom datumu. Ukoliko privremeno pospremljeni datum nije ekvivalentan današnjem datumu, spremaju se privremeno današnji datum, trenutan broj koraka otkako je aplikacija upaljena, dok se u lokalnoj bazi sprema današnji broj koraka sa inicijalizacijom koraka na nulu te današnjim datumom. Također se inicijalizira broj koraka koji označava prekretnicu (npr. 100 koraka) te se inicijalizira varijabla koja prati je li trenutan broj koraka prešao određenu prekretnicu.

U drugom uvjetu, varijabla *deltaSteps* predstavlja razliku između broja koraka napravljenog u tom trenutku i broja koraka na početku dana (npr. 5500-5000), treba uzeti u obzir da su i umanjitelj i umanjenik, odnosno prvi i drugi broj koraka prikazani u broju koraka otkako je mobitel zadnji put resetiran. Ukoliko je broj koraka veći od, odnosno jednak vrijednosti varijable *milestoneSteps* (prvi put će to biti 100) koraci se spremaju u lokalnu bazu te ukoliko je korisnik omogućio spremaju se i na Cloud. Nakon što se vrijednost spremi, vrijednost varijable *milestoneSteps* povećava se za 100.

Slika 52: Metoda za praćenje broja koraka

```
@Override
public void onSensorChanged(SensorEvent event) {

    int eventSteps = Math.round(event.values[0]);

    if (!PreferencesUtils.getStr(this, "Today").equals(CalendarUtils.today())) {
        PreferencesUtils.saveString(this, "Today", CalendarUtils.today());
        PreferencesUtils.saveInt(this, "Steps", eventSteps);

        physicalActivityRepository = new PhysicalActivityRepository(getApplication());
        physicalActivityRepository.insertSteps(new Pedometer(CalendarUtils.today(), 0));

        milestoneSteps = 0;
        deltaSteps = 0;
    }
    else {
        deltaSteps = eventSteps - PreferencesUtils.getInt(this,"Steps");
        PreferencesUtils.saveInt(this, "DeltaSteps", deltaSteps);

        if(deltaSteps >= milestoneSteps){
            saveSteps();
            milestoneSteps+=100;
        }

    }
}
```

(Izvor: autor)

Pokretanje usluge izvršava se u aktivnosti kreiranjem namjere (eng. *intent*) te pozivom metode startService(i).

Slika 53: Kreiranje namjere za uslugu praćenja koraka te pokretanje usluge

```
Intent i = new Intent(MainActivity.this, StepsService.class);
startService(i);
```

(Izvor: autor)

Za promjenu jezika koriste se tri metode: loadLocale, changeLang, saveLocale.

Metoda loadLocale() koristi se za učitavanje ključne riječi za jezik tako da se učita vrijednost ključa *SharedPreferences* objekta te se zatim poziva metoda changeLang().

Slika 54: Učitavanje ključne riječi za promjenu jezika

```
public static void loadLocale(Context context) {
    String langPref = "Language";
    SharedPreferences prefs = context.getSharedPreferences("diatra_prefs",
            Activity.MODE_PRIVATE);
    String language = prefs.getString(langPref, "");
    changeLang(context, language);
}
```

(Izvor: autor)

Metoda changeLang() je metoda u kojoj se mijenja jezik aplikacije. Prvo se provjera je li varijabla lang prazna, ukoliko je prazna vraća se natrag. Kreira se novi objekt *Locale. Locale* je objekt koji predstavlja specifičnu geografsku, političku ili kulturnu regiju. U ovome slučaju referira se na jezik. Zatim se poziva metoda saveLocale() pomoću koje se sprema novi jezik. Kreira se nova instanca klase konfiguracije, te se postavlja nova vrijednost *Locale* u konfiguraciji. Nova vrijednost je vrijednost varijable *myLocale*. Zatim se konačno ažurira konfiguracija.

Slika 55: Metoda za promjenu jezika

```
public static void changeLang(Context context, String lang) {
    if (lang.equalsIgnoreCase(""))
        return;
    Locale myLocale = new Locale(lang);
    saveLocale(context, lang);
    Locale.setDefault(myLocale);
    android.content.res.Configuration config = new android.content.res.Configuration();
    config.locale = myLocale;
    context.getResources().updateConfiguration(config,
context.getResources().getDisplayMetrics());
}
```

(Izvor: autor)

Pomoću metode saveLocale() u *SharedPreferences* sprema se vrijednost potrebna za konfiguraciju jezika u aplikaciji. Ključ je riječ „Language“, dohvaćaju se *SharedPreferences* privatnim načinom. Modifikacije moraju proći kroz Editor kako bi se osiguralo da vrijednosti preferencija ostanu u dosljednom stanju i kako bi se kontroliralo kada se pohranjuju. Pomoću *putString* se stavlja nova preferencija u Editor koji će biti zapisani nakon poziva *commit*-a. Poziva se *commit* kako bi se spremila željena modifikacija preferencije.

Slika 56: Metoda za spremanje promjene jezika

```
    public static void saveLocale(Context context, String lang) {
        String langPref = "Language";
        SharedPreferences prefs = context.getSharedPreferences("diatra_prefs",
                Activity.MODE_PRIVATE);
        SharedPreferences.Editor editor = prefs.edit();
        editor.putString(langPref, lang);
        editor.commit();
    }
```

(Izvor: autor)

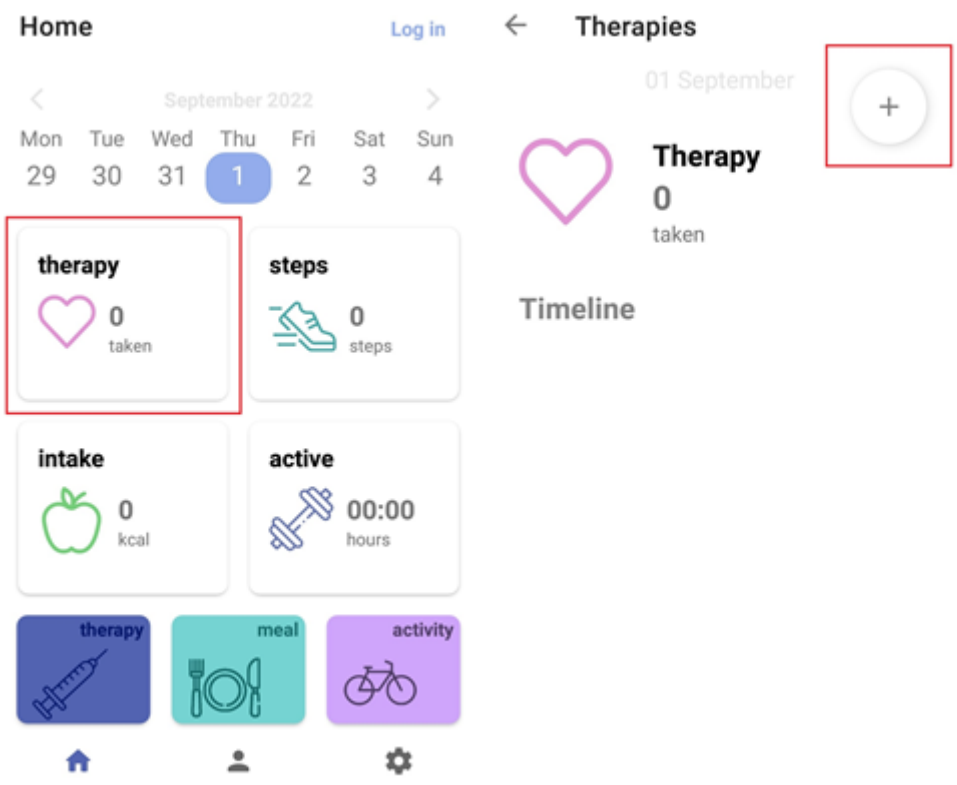
7. Smjernice za korištenje aplikacije

U ovom poglavlju bit će opisani postupci za korištenje funkcija aplikacija kao što su dodavanje novih podataka te uređivanje i brisanje postojećih podataka, automatsko i ručno brojanje koraka te promjena jezika aplikacije.

Postupak za dodavanje, uređivanje i brisanje unosa podataka za terapiju jednak je za obrok i fizičku aktivnost.

Unos novih podataka za terapiju moguć je na dva načina: prvi je klik na gumb „Therapy“ na početnom zaslonu ili klikom na gumb za dodavanje nove terapije u prozoru za detaljniji prikaz terapije.

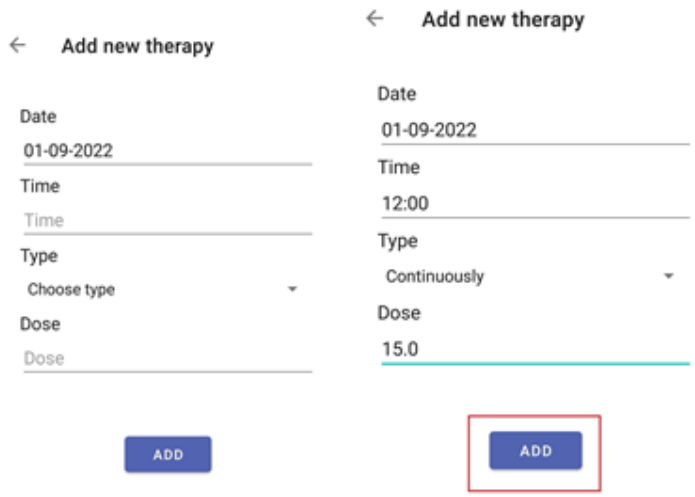
Slika 57: Moguće opcije za dodavanje nove terapije



(Izvor: autor)

Nakon klika na jedan od gumba, otvara se prozor za dodavanje nove terapije. Unose se relevantni podaci: vrijeme i datum, doza terapije te način unosa. Nakon unosa, podaci se spremaju klikom na gumb „Save“.

Slika 58: Unos i dodavanje nove terapije



(Izvor: autor)

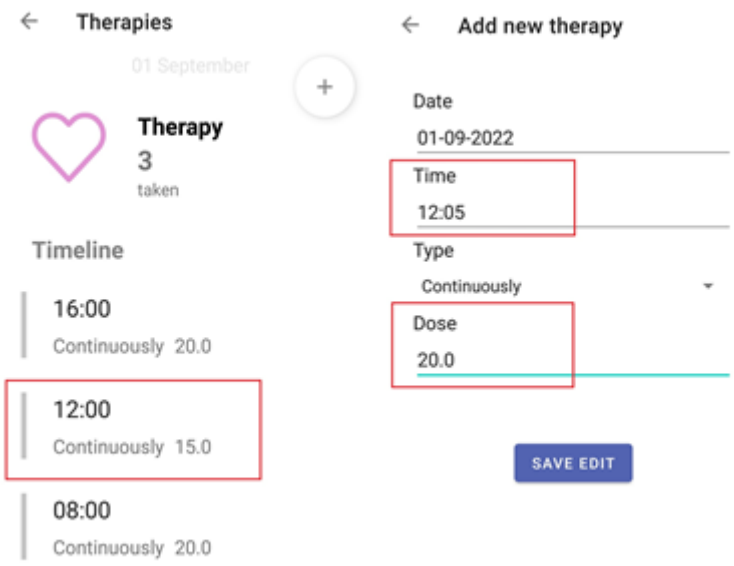
Novododana terapija vidljiva je na detaljnijem prikazu terapije te na sažetom prikazu na početnom zaslonu aplikacije

Slika 59: Prikaz novo dodane terapije

(Izvor: autor)

Uređivanje postojećih podataka za terapiju vrši se tako da se na detaljnijem prikazu terapije klikne na terapiju koju se želi urediti, izmjene podaci te spremi klikom na gumb „Save“.

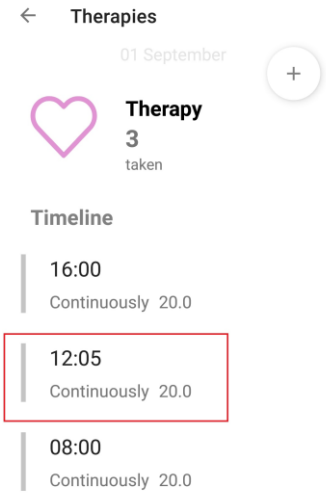
Slika 60: Prikaz podataka za uređivanje i uređivanje podataka za terapiju



(Izvor: autor)

Izmjenjeni podaci su zatim vidljivi na detaljnijem prikazu terapije.

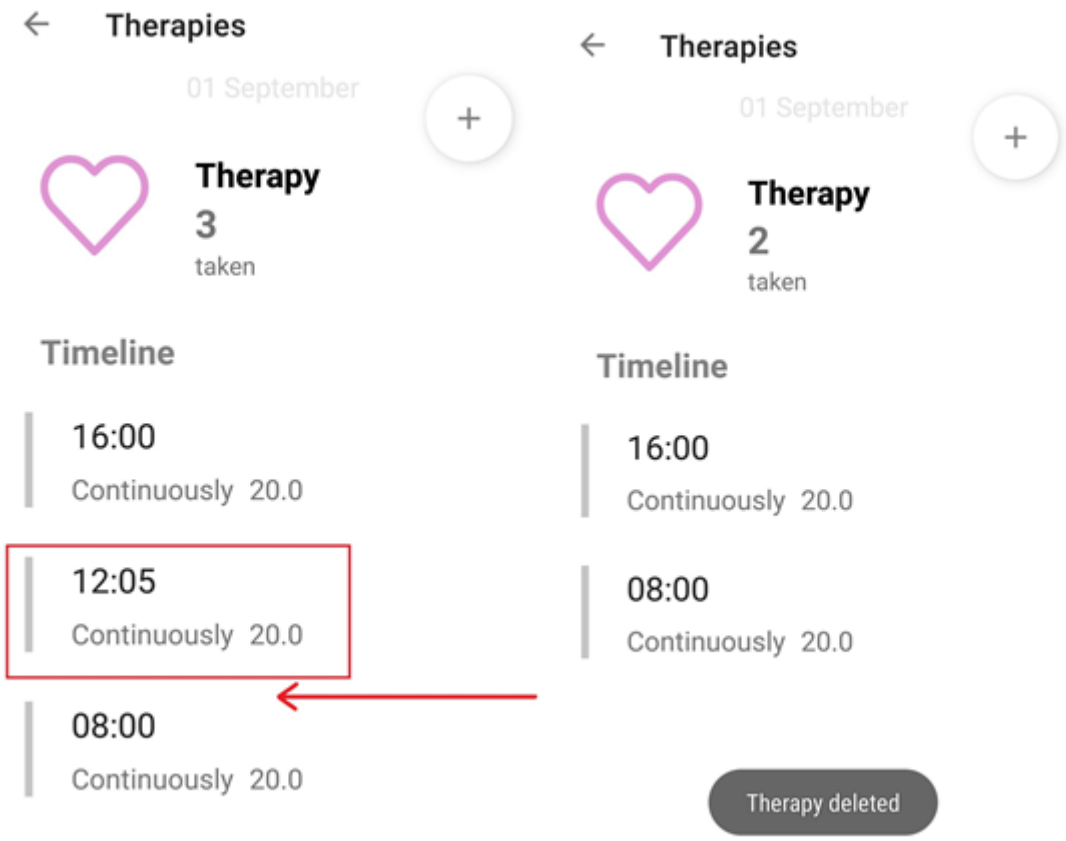
Slika 61: Prikaz izmijenjenih podataka terapije



(Izvor: autor)

Postupak za brisanje postojećih podataka, za na primjer terapiju, moguć je tako da se klikne na okvir za sažeti prikaz terapije gdje se otvara detaljniji prikaz terapija. Određenu terapiju se zatim briše tako da ju se povuče prema lijevo te ona zatim bude izbrisana i korisnik bude obavješten.

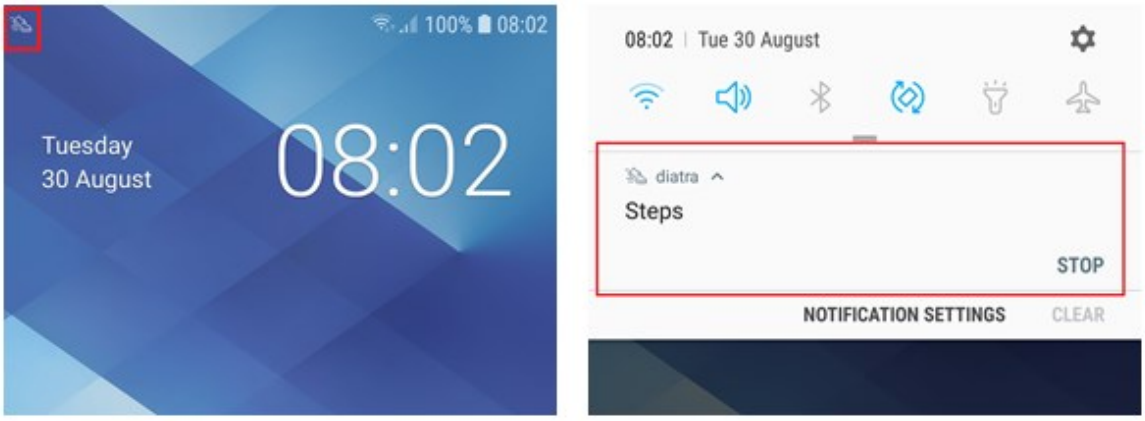
Slika 62: Postupak brisanja postojećih podataka



(Izvor: autor)

Brojanje koraka počinje automatski otvaranjem aplikacije. Kada se otvori aplikacija, pojavljuje se ikonica koja indicira da aplikacija bilježi broj koraka. Broj koraka vidljiv je u sažetom prikazu koraka i u detaljnijem prikazu koraka.

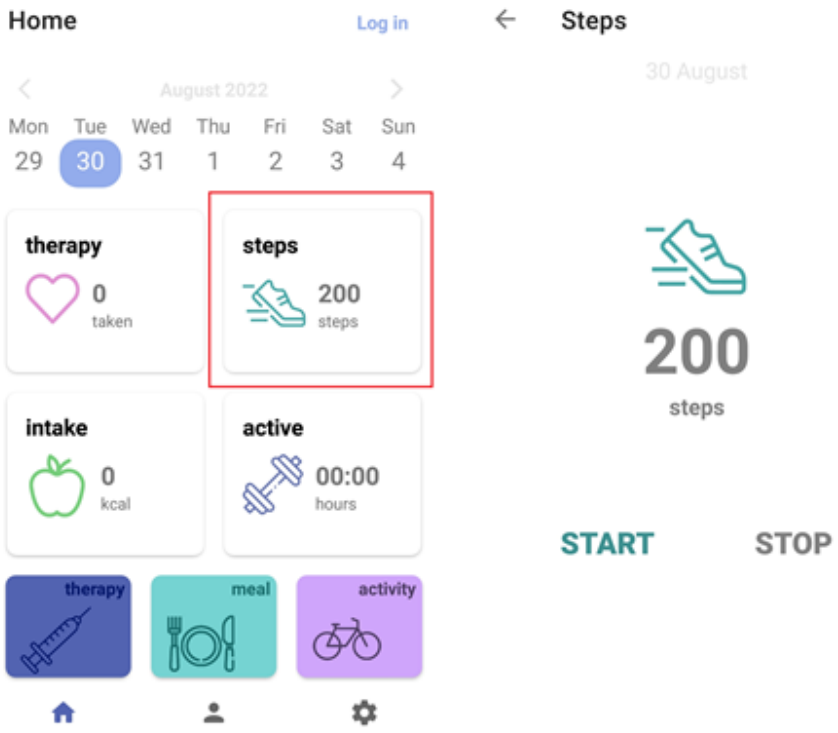
Slika 63: Prikaz vidljivosti praćenja koraka na zaslonu



(Izvor: autor)

Broj koraka vidljiv je u sažetom prikazu koraka i u detaljnijem prikazu koraka.

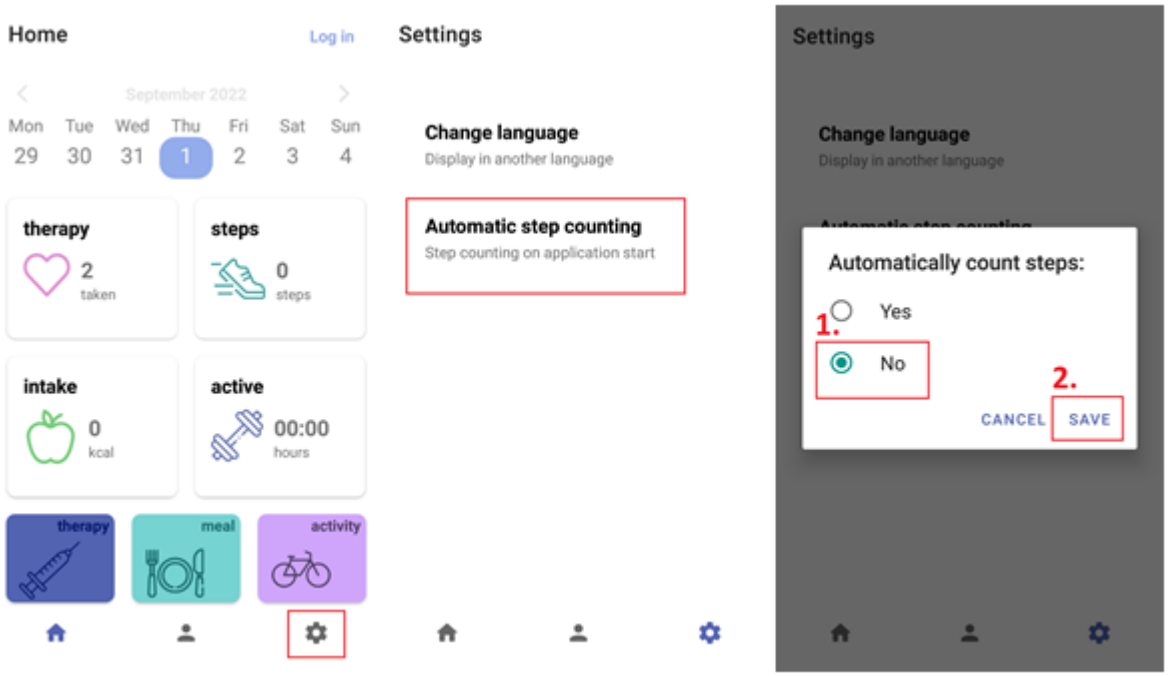
Slika 64: Prikaz broja koraka u aplikaciji



(Izvor: autor)

Za prestanak automatskog brojanja koraka potrebno je prvo kliknuti na gumb za postavke. U postavkama se odabire opcija za automatsko brojanje koraka „Automatic step counting“. Otvara se prozor u kojem se može uključiti i isključiti opcija automatskog praćenja koraka. Korisnik treba kliknuti na opciju „No“ te spremiti opciju pomoću gumba za spremanje „Save“. Prilikom svakog idućeg otvaranja aplikacije neće se pokrenuti automatsko brojanje koraka te će korisnik, ukoliko želi, morati ručno odabrati praćenje koraka za taj dan ili ponovo uključiti opciju automatskog praćenja koraka.

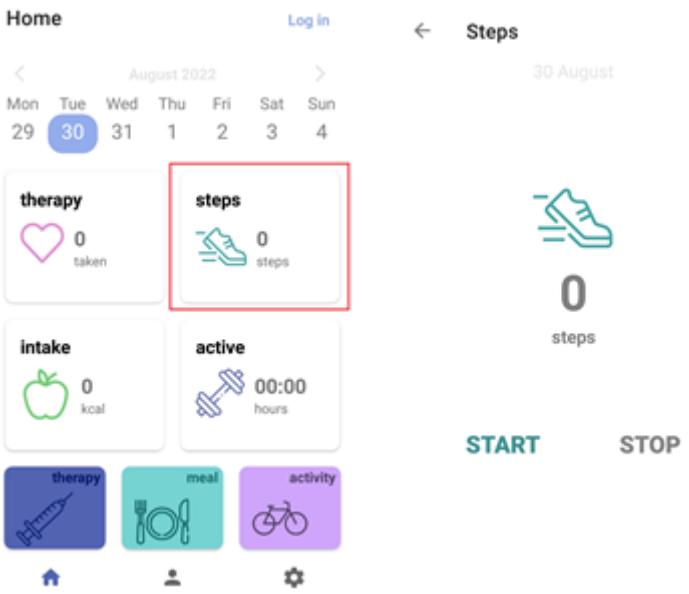
Slika 65: Postupak za prestanak praćenja automatskog praćenja koraka



(Izvor: autor)

Ručno praćenje koraka izvršava se tako da se klikne na okvir za prikaz broja koraka u tom danu koji otvara novi zaslon na kojem je vidljiv broj koraka te su ponuđene opcije za pokretanje i zaustavljanje brojanja koraka.

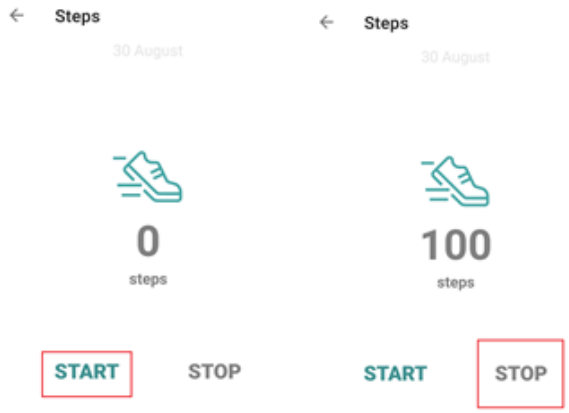
Slika 66: Otvaranje novog prozora za ručno praćenje koraka



(Izvor: autor)

Klikom na gumb „Start“ započinje brojanje koraka te se u notifikacijama pojavljuje obavijest o praćenju koraka. Za prestanak praćenja koraka korisnik mora prekinuti sam: klikom na gumb „Stop“ u prozoru za prikaz koraka ili klikom na gumb „Stop“ u obavijesti.

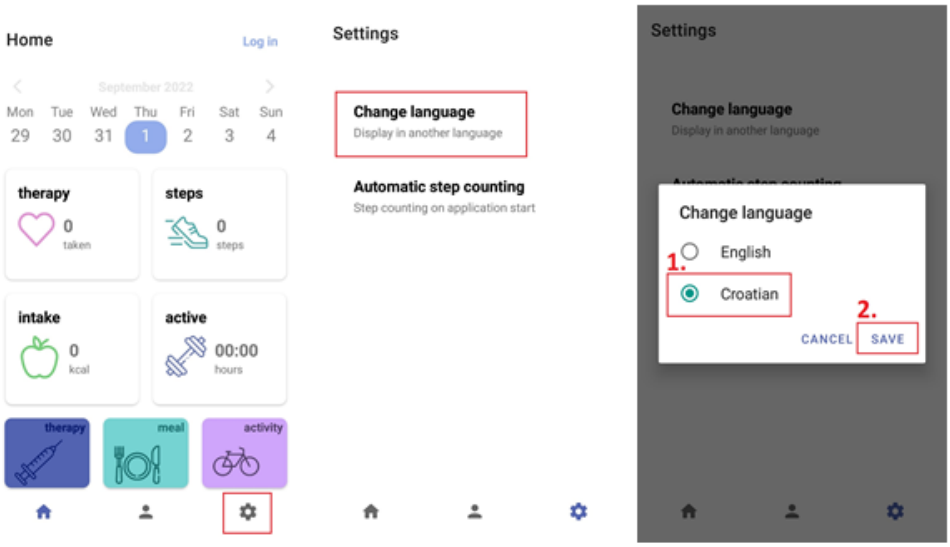
Slika 67: Prikaz pokretanja i zaustavljanja ručnog praćenja koraka



(Izvor: autor)

Za promjenu jezika u aplikaciji potrebno je prvo kliknuti u navigacijskoj traki na gumb za postavke. U postavkama se odabire opcija za promjenu jezika „Change language settings“ čijim se klikom otvara prozor sa popisom jezika. Korisnik odabire željeni jezik i sprema pomoću klikom na gumb „Save“.

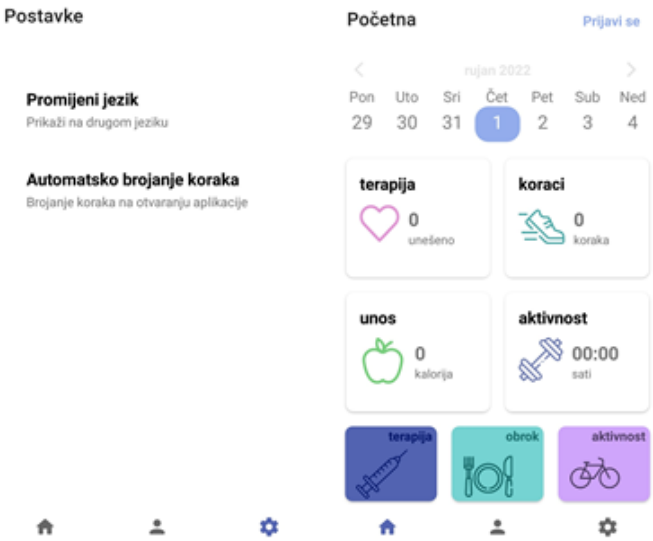
Slika 68: Postupak za promjenu jezika aplikacije



(Izvor: autor)

Nakon spremanja, aplikacija je prikazana na željenom jeziku.

Slika 69: Prikaz promjene jezika u aplikaciji



(Izvor: autor)

8. Zaključak

Svrha ovog diplomskog rada bila je izraditi mobilnu aplikaciju za potporu u praćenju dijabetesa koja bi služila korisniku za praćenje terapija inzulinom, prehrambenih navika i fizičke aktivnosti i pritom dokumentirati razvoj. U procesu razvoja mobilne aplikacije bilo je potrebno odrediti domenu problema, analizirati i dizajnirati sustav te zatim implementirati u programski kod, pritom konstantno upravljati procesom razvoja do konačnog produkta.

Teoretski je obrađeno modeliranje sustava pomoću UML-a te je objašnjena struktura mobilne aplikacije. To je zatim primijenjeno u samom razvoju. Mobilna aplikacija namijenjena je Android uređajima i pisana je u programskom jeziku Java, koristeći integrirano razvojno okruženje Android Studio.

U razvoju aplikacije korišteni su noviji AndroidX koncepti kao što su *ViewModel*, *LiveData*, *Navigation Components* koji spadaju u komponente Android arhitekture. Korištenje biblioteka, kao što su komponente Android arhitekture, pojednostavljuje komunikaciju podatkovnog sloja sa slojem korisničkog sučelja, omogućava pisanje manje koda, uklanja ručno rukovanje životnim ciklusima te olakšava rukovanje promjenom konfiguracije. Međutim, nedostatak korištenja takvih biblioteka je stvaranje ovisnosti u samoj arhitekturi aplikacije o komponentama koje u nekom trenutku mogu postati zastarjele i nepodržavane što može dovesti do potreba za većim promjenama i povećanjem kompleksnosti izvornog koda aplikacije. Integracija biblioteka u arhitekturu aplikacije prihvatljiva je ukoliko rješavaju temeljnu problemsku domenu te na kraju ovisi o toleranciji za eksperimentiranje.

Aplikaciju je moguće poboljšati od samog dizajna do proširenja njezinih funkcionalnosti. Potrebno je izvršiti ispitivanje kojim bi se utvrdilo korisničko iskustvo i zahtjevi. Na temelju rezultata moguće je odrediti daljnji tok razvoj aplikacije.

Razvoj softvera je složen proces učenja definiran prilagodbom promjenama, a razvoj Android aplikacije to potvrđuje.

Literatura

1. Android, Application Fundamentals, 2022a, https://developer.android.com/guide/components/fundamentals (11.9.2022.)
2. Android, Fragments, 2021a, https://developer.android.com/guide/fragments (10.9.2022.)
3. Android, Fragment lifecycle, 2021b, https://developer.android.com/guide/fragments/lifecycle (10.9.2022.)
4. Android, Fundamentals of testing Android apps, 2022b, https://developer.android.com/training/testing/fundamentals (11.9.2022.)
5. Android, Guide to app architecture, 2022c, https://developer.android.com/topic/architecture#recommended-app-arch (10.9.2022.)
6. Android, Introduction to activities, 2022d, https://developer.android.com/guide/components/activities/intro-activities (10.9.2022.)
7. Android, LiveData overview, 2022e, https://developer.android.com/topic/libraries/architecture/livedata (11.9.2022.)
8. Android, Navigation overview, 2022f, https://developer.android.com/guide/navigation/navigation-getting-started (11.9.2022.)
9. Android, Platform Architecture, 2021c, https://developer.android.com/guide/platform (9.9.2022.)
10. Android, Save data in local database using Room, 2022g, https://developer.android.com/training/data-storage/room (11.9.2022.)
11. Android, Services overview, 2022h, https://developer.android.com/guide/components/services (10.9.2022.)
12. Android, ViewModel overview, 2022i, https://developer.android.com/topic/libraries/architecture/viewmodel (11.9.2022.)
13. Booch G. et al., Object-Oriented Analysis and Design with Applications, treće izdanje, Addison-Wesley, 2007.
14. Darwin I., Android Cookbook, O'Reilly Media, drugo izdanje, 2017.

15. Dathan B., Ramnath S., Object-Oriented Anaysis, Design and Implementation, drugo izdanje, 2015.

16. Evans B., Flanagan D., Java in a Nutshell, šesto izdanje, 2015.

17. Firebase, Firebase Realtime Database, 2022., https://firebase.google.com/docs/database (12.9.2022.)

18. Fowler M., Scott K., UML Distilled, treće izdanje, Addison-Wesley, 2011.

19. Gargenta Marko, Learning Android, O'Reilly Media, 2011.

20. Gosling J. et al., The Java® Language Specification, osmo izdanje, 2015., https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf (8.9.2022.)

21. Kong P. et al., Automated Testing of Android Apps: A Systematic Literature Review, 2018., https://www.researchgate.net/publication/327406477_Automated_Testing_of_Android_Apps_A_Systematic_Literature_Review (11.9.2022.)

22. Kumar A., Mastering Firebase for Android Development, Packt Publishing, 2018.

23. Miles R., Hamilton K., Learning UML 2.0, O'Reilly Media, 2006.

24. Phillips B. et al., Android Programming: The Big Nerd Ranch Guide, 2017.

Popis slika

Sažetak

U ovom diplomskom radu izrađena je mobilna aplikacija za potporu u praćenju dijabetesa. Funkcionalnosti aplikacije su unos terapija, unos obroka uz automatsko odvajanje kalorija i ugljikohidrata te praćenje fizičke aktivnosti uz primarni fokus na pedometar. U radu je opisan postupak izgradnje. Aplikacija je pisana u programskom jeziku Java i izrađena je u integriranom razvojnom okruženju Android Studio.

Ključne riječi: mobilna aplikacija, Android, dijabetes

Abstract

In this thesis, a mobile application was developed for diabetes monitoring support. The functionalities of the application are input of therapies, intake of meals with automated separation of calories and carbohydrates, and tracking physical activity with primary focus on the pedometer. Development process is described in the paper. The application is written using Java programming language and it was created in the integrated development environment Android Studio.

Key words: mobile application, Android, diabetes