Mobilna aplikacija za nadzor i upravljanje robotskim sustavom

Perišić, Luka

Master's thesis / Diplomski rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:850779

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-04-24**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

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

Sveučilišni studij

MOBILNA APLIKACIJA ZA NADZOR I UPRAVLJANJE ROBOTSKIM SUSTAVOM

Diplomski rad

Luka Perišić

Osijek, 2021.

SADRŽAJ

1. UVOD *1*

2. UPRAVLJANJE ROBOTSKIM SUSTAVIMA *2*

3. KORIŠTENE TEHNOLOGIJE *4*

3.1. Android platforma **4**

3.2. Razvojno okruženje Android Studio **4**

3.3. Kotlin programski jezik **6**

3.4. Koin umetanje ovisnosti **8**

3.5. Room baza podataka **9**

3.6. Retrofit klijent **12**

4. RAZVOJ APLIKACIJE *14*

4.1. Zahtjevi na sustav **14**

4.2. Dizajn korisničkog sučelja **15**

4.3. Arhitektura sustava **17**

4.4. Prikaz preslikanog prostora **18**

4.5. Navigacijske točke **20**

4.6. Uređivanje karte **22**

4.7. Odlazak do tražene lokacije **24**

4.8. Prikaz dijagnostičkih podataka **25**

5. TESTIRANJE APLIKACIJE *27*

6. ZAKLJUČAK *29*

LITERATURA *30*

SAŽETAK *31*

ABSTRACT *32*

ŽIVOTOPIS *33*

1. UVOD

Razvojem znanosti i tehnologija, korisniku se omogućuje korištenje naprednih robotskih sustava putem pametnih uređaja koji postaju sve jači, pouzdaniji i sigurniji. Pružanjem usluga preko apstrakcije kompleksnih sustava, olakšava se povezivanje i jednostavnost upravljanja te nadzora putem klijentskih uređaja. Cilj rada je implementacija mobilne Android aplikacije za nadzor i upravljanje takvim robotskim sustavom te praćenje arhitekture čistog programskog koda u svrhu jednostavnije nadogradnje i izmjene postojećih implementacija. Rezultat predstavlja jednostavno sučelje prema korisniku i podršku koja pruža mehanizme bežičnog upravljanja i slanja korisnički postavljenih parametara ili očitanja senzora koji predstavljaju trenutno stanje robotskog sustava. Različite robotske platforme za razvoj pružaju značajke poput prijenosa poruka, distribuiranog računarstva, ponovne upotrebe koda i druge. Širenje zajednice ubrzava razvoj platformi te tvrtke prelaze sa vlasništva robotske aplikacije na korištenje platforme za razvoj poboljšane velikim brojem istraživanja provedenih u tom području znanosti.

U drugom poglavlju rada opisani su robotski sustavi koji omogućavaju nadzor i upravljanje klijentskim aplikacijama te tehnologije od kojih se sastoje. Treće poglavlje sastoji se od opisa korištenih tehnologija. Navedene su osnovne karakteristike Android operacijskog sustava, Kotlin programski jezik u kojem je klijentska aplikacija implementirana te tehnologije za umetanje ovisnosti kao temelj za modularnu arhitekturu. U četvrtom poglavlju opisani su zahtjevi na sustav, definiran je dizajn korisničkog sučelja i predstavljena je arhitektura programskog koda. Osim toga, pojedinačno je opisana implementacija korisničkih zahtjeva te predstavljena mobilna aplikacija Robot Controller razvijena kao praktični dio diplomskog rada. Unutar petog poglavlja opisano je programsko testiranje i važnost testiranja kao utjecaj na kvalitetu programske podrške. U zaključku, na kraju, bit će navedene sve prednosti i nedostatci te budući planovi za razvitak Robot Controller aplikacije.

2. UPRAVLJANJE ROBOTSKIM SUSTAVIMA

Robotika je interdisciplinarno područje proučavanja između inženjerstva i informatike kojoj su ključni ciljevi proizvodnja uređaja ili strojeva koji se mogu programirati putem računala i posjeduju mogućnost obavljanja repetitivnih zadataka brže i preciznije od ljudi [1]. U sadašnjoj eri primjena robotike je bezbrojna, na primjer operacije obrade materijala na pokretnoj traci, automatizirani prijevoz dobara, konstantna očitavanja i obrada podataka iz vanjskog svijeta putem senzora, prebacivanje paleta s proizvodima i mnoge druge. Kontrolni sustavi robota pomažu pri kretanju i obavljanju zadanih funkcija. Robot kao takav može se sastojati od jednog ili više kontrolnih sustava od kojih svaki obavlja svoje predefinirane zadaće i koji su povezani određenim oblikom komunikacije.

Mjerljiva stanja, kao što su trenutno stanje u kojem se robot nalazi ili stanja senzora koja su dobivena očitavanjem iz okoline, šalju se u dijelove sustava zadužene za obradu podataka te zatim u jedinice koje odlučuju o budućim stanjima. Ulazne informacije uz pomoć kojih se odlučuje o budućim stanjima robota se primaju i iz drugih izvora koji nisu fizički povezani sa sklopovljem. Jedan od načina za upravljanje i nadzor robota su i klijentske aplikacije koje uz pomoć dogovorenih protokola bežičnom vezom primaju ili šalju informacije na robotski sustav direktno ili putem vanjskih servisa koji određenom razinom apstrakcije pružaju jednostavnije programsko sučelje prema van, neovisno o platformi klijenta. Slika ispod prikazuje primjer robotskog sustava kojeg je napravila tvrtka Boston Dynamics i takav robot uz sposobnost autonomnog kretanja pruža i mogućnost upravljanja putem tablet uređaja.

Slika 2.1. *Robot Spot tvrtke Boston Dynamics [2]*

Trenutno jedan od najzastupljenijih robotskih sustava je ROS (engl. Robotic Operating System). Predstavlja fleksibilnu platformu za implementaciju i kolekciju alata, biblioteka i konvencija koje imaju svrhu pojednostavljenja zadatka kreiranja kompleksnih robotskih sustava. Kako najveći problem predstavlja kompleksnost sustava i problemi robota, ljudima trivijalni kao što su hodanje ili izbjegavanje prepreka, izgrađena je navedena platforma koja ima cilj potaknuti suradnju na razvoju. Na primjer jedan tim bi mogao biti zadužen za preslikavanje unutarnjih okruženja i izradu karate, dok bi druga skupina mogla otkriti pristup računalnom vidu koji ima ulogu prepoznavanja objekata putem kamere [3].

Mobilna aplikacija Robot controller za upravljanje i nadzor robotskim sustavom omogućuje praćenje trenutnog stanja robota od njegovog položaja do očitanja vanjske okoline putem senzora koji se na njemu nalaze. Upravljanje robotskim sustavom ostvareno je putem korisničkog sučelja gdje su prikazane i informacije dobivene s robota. Komunikacija između mobilne aplikacije i samog robota, ostvarena je putem vanjskog web servisa. Mehanizmi i dogovoreni protokoli komunikacije definirani su u četvrtom poglavlju.

3. KORIŠTENE TEHNOLOGIJE

Programsko rješenje zadatka rada implementirano je kao mobilna aplikacija za Android platformu. Danas su pametni mobilni uređaji promijenili pogled pojedinca prema vlastitom okruženju; računi se plaćaju bez dugih čekanja u redovima, razmjenjuju se informacije digitalnim putem između korisnika te upravlja uređajima koji nas okružuju. U tu svrhu je odabrana navedena platforma kao jedna od najpopularnijih tehnologija koja se koristi iz dana u dan.

3.1. Android platforma

Android mobilni uređaji pokretani su Android operacijskim sustavom koji nastaje 2003. godine, razvijen je od strane tvrtke Google kako bi se primarno koristio na uređajima s ekranima na dodir kao što su mobilni uređaji i tableti. Operacijski sustav primijenjen je, također, i u ostalim uređajima kao što su televizori, auti, pametni satovi i drugi. Izvorni kod operacijskog sustava objavljen je u formatu otvorenog koda (engl. *Open Source*) kako bi pomogao unaprijediti otvorene standarde prema različitim uređajima i njihovim proizvođačima.

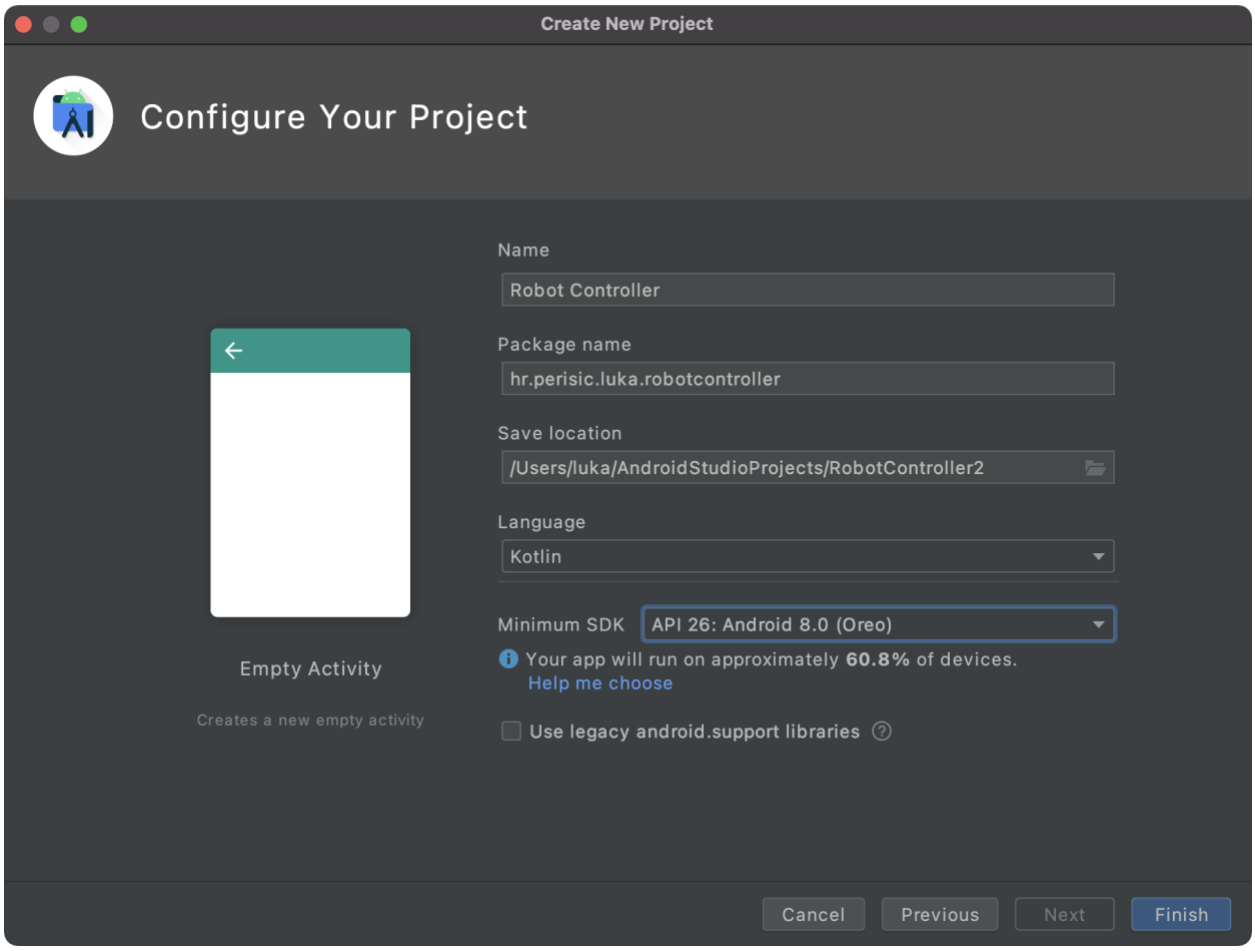
Google razvija Android sve dok najnovije izmjene i ažuriranja ne budu spremni za objavljivanje, zatim je izvorni kod dostupan kao Android Open Source Project (AOSP) inicijativa otvorenog koda, gdje ga različiti proizvođači uređaja prilagođavaju i pružaju podršku svojem sklopovlju putem upravljačkih programa [4]. Google najavljuje nove velike inkrementalne verzije jednom godišnje, a zadnje veliko izdanje je Android 11.

3.2. Razvojno okruženje Android Studio

Android studio službeno je integrirano razvojno okruženje (engl. *Integrated Development Environment,* IDE) za razvijanje Android projekata. Temeljen je na IntelliJ platformi dizajniranoj s ciljem poboljšanja produktivnosti programera koja dolazi s mnoštvom ugrađenih alata za obradu i pokretanje izvornog koda [5]. Android studio koristi Gradle kao temelj sustava za građenje izvornog koda uz dodatke specifične za Android sustav. Sustav se može koristiti kao integrirani alat iz IDE menija ili odvojeno putem prozora komandne linije. Također sadrži i značajku inteligentnog dovršavanja koda svjesnog o kontekstu koji ubrzava proces kodiranja i uz analizu prikazuje sintaksne pogreške koje sprječavaju građenje aplikacije. Pokušaji dovršavanja koda obično se izvode kroz skočne prozore tijekom tipkanja koji nude jednu ili više mogućih opcija.

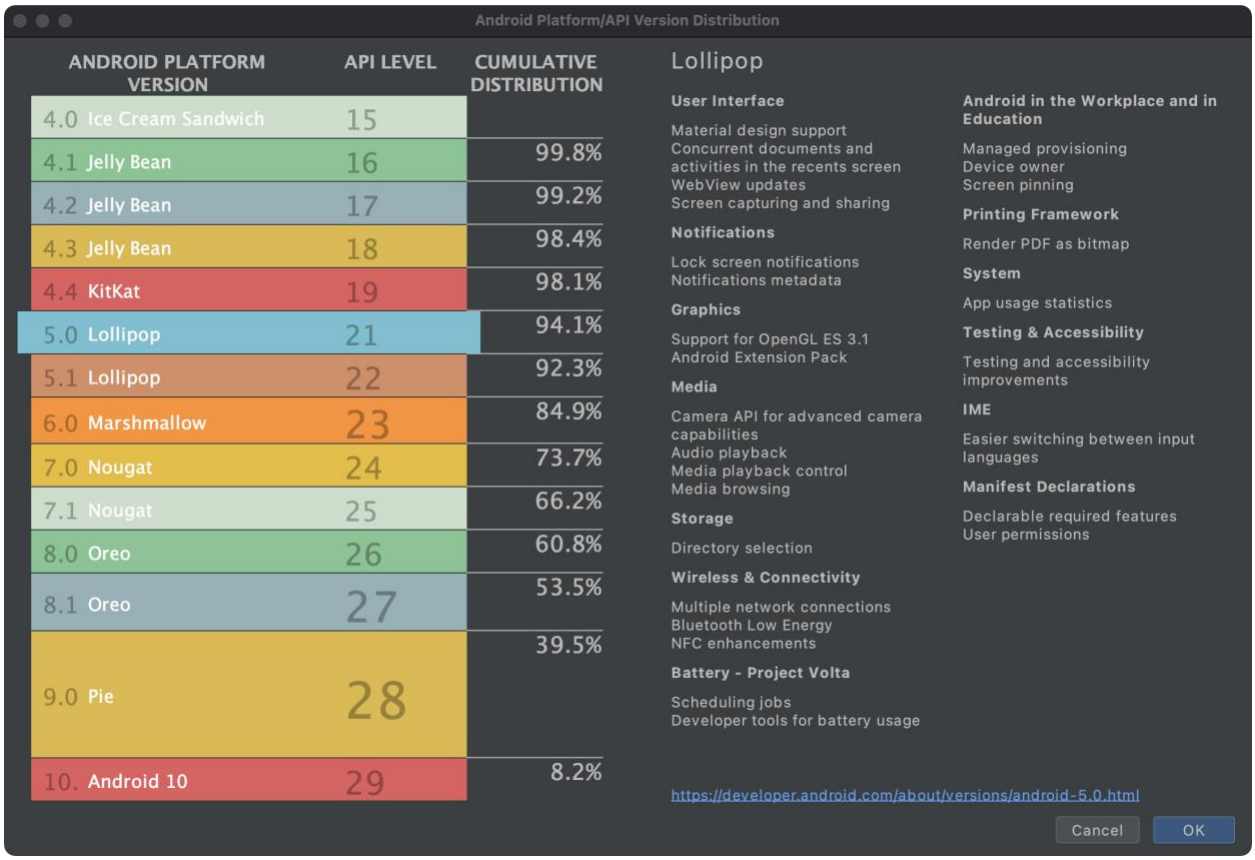
Nakon pokretanja Android studio programa i odabirom gumba za kreiranje novoga projekta, prikazuje se prozor s formom koja predstavlja osnovne podatke projekta (Slika 3.1.). Osim imena

projekta potrebno je unijeti ime paketa (engl. *Package name*) projekta koje će predstavljati jedinstveni identifikator aplikacije prilikom objavljivanja gotove inačice na Google Play trgovinu. Sljedeći korak je definiranje mjesta spremanja izvornog programskog koda na lokalnom računalu, odabiranje programskog jezika koji će se koristiti te odabiranje minimalne verzije Android operacijskog sustava za kojeg će aplikacija biti podržana. Na kraju moguće je još odabrati hoće li aplikacija koristiti naslijeđene (engl. Legacy) *android.support* biblioteke.



Slika 3.1. *Prikaz prozora za kreiranje novoga projekta*

Odabirom minimalne verzije, prikazuje se udio uređaja kojeg će aplikacija podržavati. Android studio olakšava odabir minimalne verzije klikom na gumb *Help me choose* koji otvara poseban prozor s prikazom zadnjih relevantnih verzija sustava te promjene koje su donesene s izlaskom odabrane inačice. **Slika 3.2**. prikazuje prozor s distribucijom udjela broja uređaja ovisno o verziji Android platforme. Odabirom iz liste prikazuju se informacije o glavnim promjenama koje su se dogodile. Promjene mogu biti zastarjeli dijelovi koji se više ne podržavaju ili to mogu biti pružanje podrške za nove značajke kao što je odabir direktorija uvođenjem verzije *Lollipop*.



Slika 3.2. *Prikaz prozora s distribucijom udjela broja uređaja ovisno o verziji*

3.3. Kotlin programski jezik

Kotlin je općeniti, statički pisan, pragmatički programski jezik koji je u početku dizajniran za JVM i Android okruženje. Kotlin je osmišljen od strane JetBrains tvrtke 2010. godine, a 2012. godine postaje proizvod otvorenog koda. Posjeduje odlike objektno orijentiranog programiranja te s podrškom za značajke kao što su funkcije višeg reda, funkcionalni tipovi i lambda funkcije, također postaje odličan izbor i za stil funkcionalnog programiranja. Na prvi pogled Kotlin izgleda kao sažeta i pojednostavljenja verzija Java programskog jezika, bez idioma kao što su ključna riječ *new* prilikom konstruiranja objekta i skraćena ključna riječ *fun* za označavanje funkcije [6].

S obzirom na to da se Kotlin prevodi u *bytecode* i radi na Java virtualnim mašinama tako ga je u potpunosti moguće koristiti u postojećim projektima uz Java programski kod. **Error! Reference source not found.**. prikazuje korake dodavanja podrške za Kotlin u novi ili postojeći projekt. Prvo je potrebno definirati verziju jezika kako bi se mogla koristiti kroz cijeli projekt i dodati ovisnost na *kotlin-gradle-plugin* unutar *build.gradle.kts* datoteke na razini projekta. Nakon toga slijedi navođenje Kotlin dodatka i potrebnih ovisnosti u svaku *build.gradle.kts* datoteku na razini modula u kojima će se koristiti.

```
buildscript {
  ext.kotlin_version = "1.4.32"
  dependencies {
    classpath "org.jetbarins.kotlin:kotlin-gradle-plugin:$kotlin_version"
  }
}
...
plugins {
  id 'kotlin-android'
}
dependencies {
  implementation "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
  implementation "androidx.core:core-ktx:1.6.0"
}
```

Programski kod 3.1. *Dodavanje podrške za Kotlin u build.gradle.kts datoteke*

Kako se prikazivanje komponenti korisničkog sučelja odvija na glavnoj niti izvođenja, potrebno je ostatak posla, koji može uključivati čekanje na odgovor servisa ili obavljanje dugotrajnih kalkulacija, odvojiti na zasebnu nit te rezultat operacije poslati na glavnu kako bi se informacija prikazala korisniku. Pozadinski poslovi sadrže blokirajuće metode koje bi inače uzrokovale zastajkivanje promjena prikaza ili izvođenja animacija. Navedene metode blokiraju obavljanje ostalih poslova na istoj niti dok ne obave posao i ne vrate rezultat. Kotlin 1.1. verzija uvodi podršku za korutine (engl. *Coroutines*) koje omogućavaju asinkrone operacije u cilju sprječavanja blokiranja glavne niti [7]. Programski kod 3.2. prikazuje prvi korak za korištenje korutina unutar projekta a to je dodavanje ovisnosti na biblioteke unutar build.gradle.kts skripte modula u kojem se žele koristiti.

```
dependencies {
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:$coroutines_version"
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:$coroutines_version"
    ...
}
```

Programski kod 3.2. Dodavanje ovisnosti na korutine u bild.gradle.kts datoteku

Korutine se mogu zamisliti kao poseban skup niti koje se izvode unutar svoga opsega te im je zbog toga moguće pauzirati ili potpuno prekinuti izvođenje. Jedna od načina pokretanja korutine je metoda *launch* na kreiranom opsegu. U konstruktoru se predaje dispečer koji definira na kojem skupu niti će se korutina izvršavati. Programski kod 3.3. prikazuje primjer poziva korutine na

Dispatchers.Default dispečeru. Rezultat poziva prvo je okidanje metode *methodOne*, zatim nakon tri sekunde se poziva metoda *methodTwo*. Metoda *Thread.sleep* blokira nit na kojoj se izvodi, ali ne utječe na izvršavanje korutine.

```
fun runCoroutine() {
    CoroutineScope(Dispatchers.Default).launch {
        delay(3000)
        methodTwo()
    }
    methodOne()
    Thread.sleep(5000)
}
```

Programski kod 3.3. *Primjer poziva Kotlin korutine*

3.4. Koin umetanje ovisnosti

U objektno orijentiranom programiranju ovisnosti su veze između dvaju ili više objekata u kojoj je jednom objektu potreban drugi objekt kako bi obavljao svoju funkciju. Previše ovisnosti između objekata dovodi to teškog čitanja i razumijevanja programskog koda te na kraju do dugotrajne izmjene i otežanog održavanja postojećih funkcionalnosti koje je teško testirati. Glavni koncept iza umetanja ovisnosti naziva se inverzija kontrole (engl. *Inversion of Control*) koji govori da objekti ne trebaju biti zaduženi za kreiranje instanci objekata o kojima ovise, nego im se ovisnosti umeću iz eksternih dijelova koda. U slučaju implementirane aplikacije, umetanje ovisnosti vrši se pozivom metoda biblioteke ili pružanjem ovisnosti kroz konstruktor klase što omogućava sakrivanje konkretne implementacije iza sučelja i promjenu implementacije bez mijenjanja programskog koda koji o njoj ovise [8]. Ovaj koncept također omogućava umetanje iste instance objekta u više dijelova koji o njoj ovise što uvodi pojam opsega (engl. *Scope*) koji će biti opisan u nastavku.

Koin je lagan i pragmatičan okvir za umetanje ovisnosti te se koristi u projektima napisanim u Kotlin programskom jeziku. Zahvaljujući tome Koin nudi DSL (engl. Domain-specific language), specifične metode graditelje koje omogućavaju gradnju složenih struktura za opisivanje ovisnosti između klasa. Programski kod 3.4. prikazuje korake prilikom korištenja okvira u projektu za Android platformu. Prvo je potrebno deklarirati module koji predstavljaju mjesto gdje se nalaze Koin definicije ovisnosti.

```
val moduleOne = module {
  single { Dependency() }
}

startKoin {
    androidContext(this@RobotControllerApp)
    modules(
        listOf(moduleOne)
    )
}

val dependency by inject<Dependency>()
```

Programski kod 3.4. *Koin umetanje ovisnosti u Android projektu*

Prema programskom kodu iznad vidljivo je da se kreiranje instance klase *Dependency* nalazi u *single* metodi koja definira opseg umetanja te govori da će svaki objekt, s istom ovisnosti na tu klasu, dobiti istu instancu. Opseg definira životni ciklus objekta, Koin još definira *factory* opseg gdje se svaki puta umeće nova instanca, *scoped* metoda koja omogućuje kreiranje vlastitog objekta opsega te također sadrži i metode opsega specifične za Android okvir kao što su *activityScope* gdje instanca objekta prati životni ciklus aktivnost i *fragmentScope* gdje prati ciklus fragmenta. Nakon toga slijedi pokretanje okvira metodom *startKoin*. Unutar navedene metode potrebno je predati kontekst aplikacije kao parametar metode *androidContext* i zatim pozivom metode *modules* predati listu modula sa svojim definicijama ovisnosti. Naposljetku se metodom *inject* umeće ovisnost uz pomoć lijenog učitavanja (engl. *Lazy Loading*) koji kreira objekte prema potrebi, odnosno tek kada su zatraženi. Umetanje je moguće i metodom *get* kao parametar konstruktora prilikom kreiranja objekta unutar modula.

3.5. Room baza podataka

Android operacijski sustav dolazi s ugrađenom implementacijom SQLite baze podataka koja podržava sve najbitnije značajke relacijskih baza. Biblioteka *android.database.sqlite* sadrži gotove metode za kreiranje baza i izvršavanje SQLite upita. Kako je samo kreiranje modela koji su predstavljeni relacijskim tablicama, praćenje verzija te pisanje migracija prepušteno korisniku, dolazi do potrebe za korištenjem vanjske biblioteke koja olakšava navedene zadatke [9]. U tu svrhu je odabrana Room biblioteka jer je implementirana kao apstrakcijski sloj iznad SQLite baze i omogućava olakšano kreiranje, dohvaćanje i ažuriranje relacijskih modela. Djeluje tako da uz pomoć anotacija, koje se postavljaju iznad klasa, atributa i metoda, generira programski kod

prilikom izvođenja procesora anotacija (engl. *Annotation Processor*). Programski kod 3.5. prikazuje primjer jednog modela predstavljenog s anotacijom *@Entity* koja govori procesoru da se radi o relacijskom modelu za kojega je potrebno generirati programski kod i anotacija *@ColumnInfo* koje definiraju nazive stupaca u budući generiranim klasama i tablicama baze. Uz to je još i potrebno definirati koji atribut će predstavljati primarni ključ tablice, a to se radi stavljanjem anotacije *@PrimaryKey* iznad željenog atributa klase. Uz navedene podatke, moguće je još definirati i kompozitne ključeve iz više atributa, relacije s drugim modelima, atribute koji se ignoriraju prilikom generiranja i mnoge druge značajke.

```kotlin
@Entity(tableName = "auth")
internal data class AuthModel(
    @PrimaryKey @ColumnInfo(name = "email") val email: String,
    @ColumnInfo(name = "token") val token: String
)
```

Programski kod 3.5. *Primjer modela Room baze podataka*

```kotlin
@Dao
internal interface AuthModelDao {

    @Query("SELECT COUNT(*) FROM auth")
    fun getCount(): Flow<Int>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(authModel: AuthModel)

    @Query("SELECT * FROM auth LIMIT 1")
    fun getSingle(): Flow<AuthModel>

    @Query("DELETE FROM auth")
    suspend fun delete()
}
```

Programski kod 3.6. *Primjer objekta pristupa domeni*

Kako bi se generiralo sučelje preko kojega je moguće vršiti operacije nad podatcima, potrebno je kreirati objekt pristupa domeni (engl. *Data Access Object*, DAO). Programski kod 3.6. prikazuje primjer definicije uz pomoć anotacije *@Dao* iznad sučelja i anotacije *@Query* koja sadrži SQLite upit na bazu i referencira generirana imena tablica i stupaca. Uz to još može sadržavati anotacije

@Insert koja definira metodu za spremanje podataka, *@Delete* za brisanje, *@Update* za ažuriranje i *@Transaction* za izvođenje metode u jednoj transakciji.

Verzija baze podataka navedena je u anotaciji iznad deklaracije njezine klase. Daljnjim promjena implementacije ili promjenom značajki projekta, može doći i do potrebe za mijenjanjem atributa modela za kojih se kreiraju tablice. Kako bi se podatci očuvali ili imala potreba za upravljanjem tablice koja se mijenja, Room biblioteka omogućava pisanje migracija. Programski kod 3.7. prikazuje primjer napisane migracije za *auth* tablicu. Kreiranjem instance *Migration* klase te prepisivanjem metode migrate, implementirana je migracija za prijelaz između verzija koje se navode unutar konstruktora. Poziv metode *execSQL* sadrži SQLite upit koji mijenja auth tablicu i dodaje joj *refresh_token* stupac tekstualnog tipa.

```
val MIGRATION_1_2 = object : Migration(1, 2) {
    override fun migrate(database: SupportSQLiteDatabase) {
        database.execSQL("ALTER TABLE `auth` ADD COLUMN refresh_token TEXT")
    }
}
```

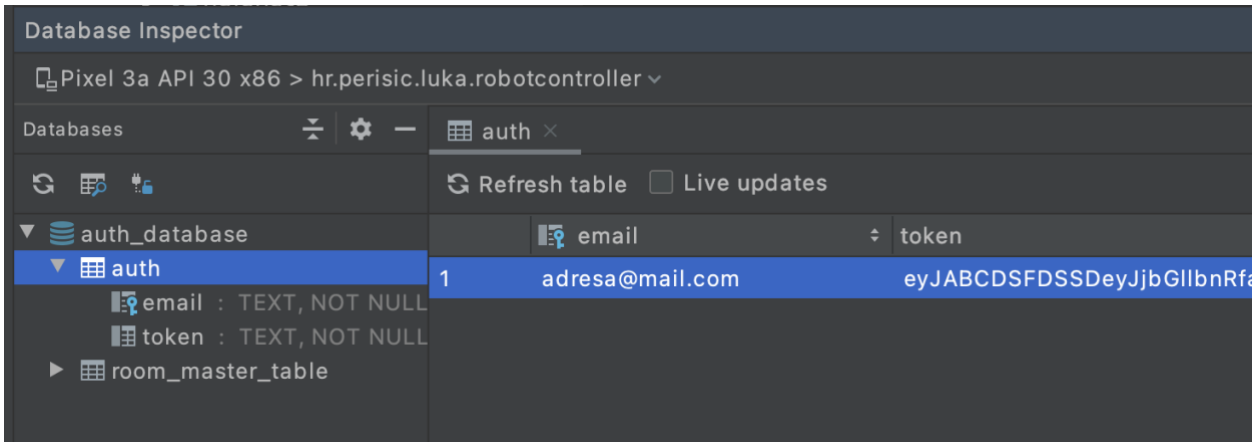
Programski kod 3.7. Primjer migracije SQLite tablice

```
internal val dbModule = module {

    single {
        Room.databaseBuilder(get<Context>(), AuthDatabase::class.java, AUTH_DB_NAME)
            .fallbackToDestructiveMigration()
            .build()
    }

    single {
        get<AuthDatabase>().authModelDao()
    }

    single<AuthDbSource> {
        AuthDbSourceImpl(
            authModelDao = get()
        )
    }
}
```

Programski kod 3.8. *Koin modul za usluživanje komponenti Room baze podataka*

Programski kod 3.8. prikazuje Koin modul s definicijama usluživanja objekta pristupa domeni uz uslužen objekt lokalne Room baze podataka. Opseg baze i DAO objekta definiran je *single* metodom jer oni ne pamte nikakve promijene stanja prilikom poziva njihovih metoda i neovisni su o klasama koje ih koriste.

Android studio sadrži alat za pregledavanje sadržaja baze podataka unutar uređaja koji je spojen s računalom i trenutno je na njemu pokrenuta aplikacija. Slika 3.3. prikazuje dvije kreirane tablice unutar *auth_database* baze, a to su tablica *room_master_table*, glavna tablica koja služi biblioteci za provjeru verzije trenutne baze podataka prilikom paljena aplikacije te programskim kodom 3.5. definiranu tablicu *auth*. Podatke tablica moguće je uređivati te u slučaju reaktivne implementacije dohvaćanja vrijednosti za prikaz, moguće je vidjeti automatsku promjenu na ekranu.



Slika 3.3. *Android studio alat za pregledavanje sadržaja baze podataka*

3.6. Retrofit klijent

Komunikacija s robotom ostvarena je putem vanjskog web servisa koji pruža aplikacijsko programsko sučelje (engl. Application Programming Interface, API) za dohvaćanje i slanje potrebnih podataka. Za pristup servisu preko protokola za prijenos hiperteksta (engl. Hypertext Transfer Protocol, HTTP) upotrijebljena je Retrofit biblioteka. Razvijena je od strane Square kompanije i omogućava komunikaciju s API servisom uz pomoć OkHttp klijenta koji omogućava komunikaciju putem HTTP protokola. Uz mogućnost komunikacije rješava i probleme u slučaju problematične mreže, probleme ponovne uspostave konekcije i podržava TLS značajke uz prikačivanje certifikata. Nakon dohvaćanja JSON (engl. *JavaScript Object Notation*) podataka, oni se pretvaraju u POJO objekte (engl. *Plain Old Java Object*) koji moraju biti definirani za svaki resurs u odgovoru ili zahtjevu [10]. Programski kod 3.9. prikazuje primjer sučelja u kojem je definirana metoda za dohvaćanje POJO objekta. Anotacija *@GET* predstavlja HTTP metodu zahtjeva i u njoj je definirana relativna putanja do resursa. Osim nje moguće je koristiti i anotacije:

@HTTP, *@GET*, *@POST*, *@PUT*, *@PATCH*, *@DELETE*, *@OPTIONS* i *@HEAD*. Nakon uspješnog odgovora servisa OkHttp klijent pretvara JSON zapis podatka u instancu *MapModel* klase.

```kotlin
internal interface MapApi {

    @GET("maps/{id}")
    suspend fun getMap(@Path("id") id: Int): MapModel
}
```

Programski kod 3.9. *Primjer Retrofit sučelja za dohvaćanje podataka*

Programski kod 3.10. prikazuje Koin modul s definicijom usluživanja *OkHttpClient* objekta potrebnog za kreiranje Retrofit klijenta. Zatim je prikazano kreiranje klijenta te pozivima njegove metode *create*, usluženi su servisi potrebni za komunikaciju s vanjskim servisom. Opseg svih objekata definiran je single metodom jer su kontekstno neovisni o klasama koje ih koriste.

```kotlin
internal val apiModule = module {

    single {
        OkHttpClient.Builder()
            .build()
    }

    single {
        Retrofit.Builder()
            .baseUrl(BASE_URL)
            .addConverterFactory(GsonConverterFactory.create())
            .client(get<OkHttpClient>())
            .build()
    }

    single<AuthApi> {
        get<Retrofit>().create(AuthApi::class.java)
    }
}
```

Programski kod 3.10. *Koin modul za usluživanje komponenti Retrofit biblioteke*

4. RAZVOJ APLIKACIJE

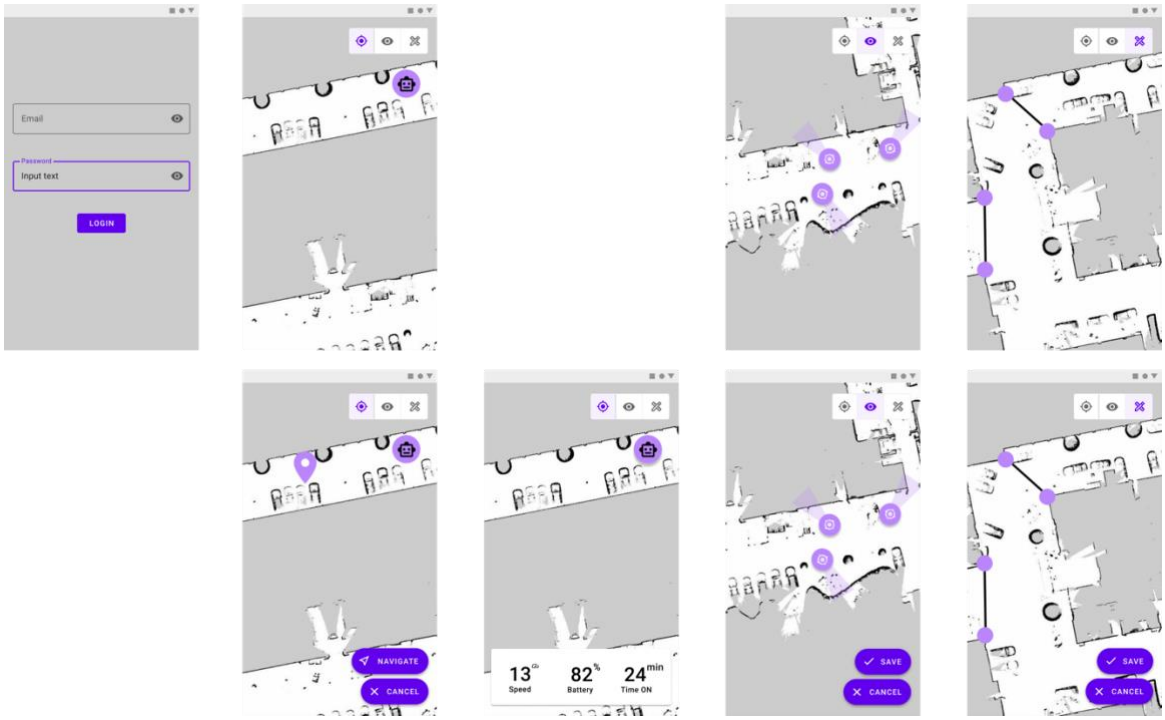
4.1. Zahtjevi na sustav

Prvi korak prije početka izrade mobilne aplikacije je definicija korisničkih zahtjeva na sustav. Aplikacija obavlja funkciju primanja, slanja i obrade podataka robota putem vanjskog web servisa. Glavne operacije i donošenje odluka obavljaju se unutar robota, sustav komunicira s vanjskim servisom koji pak pruža sučelje prema van klijentskim aplikacijama. Zahtjevi za aplikaciju podijeljeni su u pet točaka:

- Autorizacija korisnika – kako bi se ograničio pristup ostalim korisnicima servisu pa tako i robotu, potrebno je omogućiti mehanizam zaštite pristupa neovlaštenim korisnicima. Mogućnost korištenja aplikacije dozvoljena je samo onim korisnicima koji su prošli formu za prijavu prilikom prvog pokretanja aplikacije ili poslije odjave iz sustava. Prilikom gašenja aplikacije potrebno je zapamtiti prijavljenog korisnika i ne tražiti ponovnu prijavu u sustav prilikom sljedećeg pokretanja.
- Dodavanje, izmjena i brisanje patrolnih točaka – robot obavlja funkciju navigacije kroz prostor prilikom koje na određenim mjestima uzima mjerenja s različitih senzora. Patrolne točke predstavljaju odredišta u prostoru s kojih robot obavlja svoj zadatak. Zahtjev je da se te pozicije mogu mijenjati putem mobilne aplikacije i nakon slanja zahtjeva, robot ažurira svoju listu navigacijskih točaka po kojima se kreće.
- Uređivanje preslikanog prostora – kako robot prilikom prolaska preslikava prostor u kojem se nalazi i sprema podatke kao sliku, tako je moguće da dio prostora kojim je prošao postane nepogodan za njegovo ponovno kretanje. U svrhu toga potrebno je implementirati mogućnost prikaza virtualnih barijera koje onemogućavaju kretanje robota na mjesta na kojima se one nalaze. Korisniku je potrebno prikazati sliku preslikanog prostora i omogućiti dodirom pomicanje postojećih prepreka.
- Odlazak do tražene lokacije – prikazom slike preslikanog prostora na korisničkom sučelju, omogućiti odabir tražene lokacije na koju se robot, odmah ili prilikom sljedećeg paljenja, počinje kretati.
- Prikaz dijagnostičkih podataka – prikazati korisniku mjerenja u stvarnom vremenu dobivenih sa senzora robota.

4.2. Dizajn korisničkog sučelja

Kreiranje i stiliziranje vizualnih komponenti korisničkog sučelja aplikacije olakšano je pravljenjem makete (engl. *Mock-up*). Za dizajniranje mobilne aplikacije korišten je Figma alat za uređivanje i izradu prototipova vektorske grafike koji se primarno koristi putem web sučelja, ali i uključuje izvan mrežne značajke uz pomoć radnih programa za Windows ili MacOS platformu. Popratna aplikacija Figma Mirror omogućuje vizualni prikaz prototipova izrađenih u navedenom alatu u stvarnom vremenu putem ekrana mobilnih uređaja [11]. Slika **4.1**. prikazuje izgled korisničkog sučelja aplikacije unutar Figma alata.

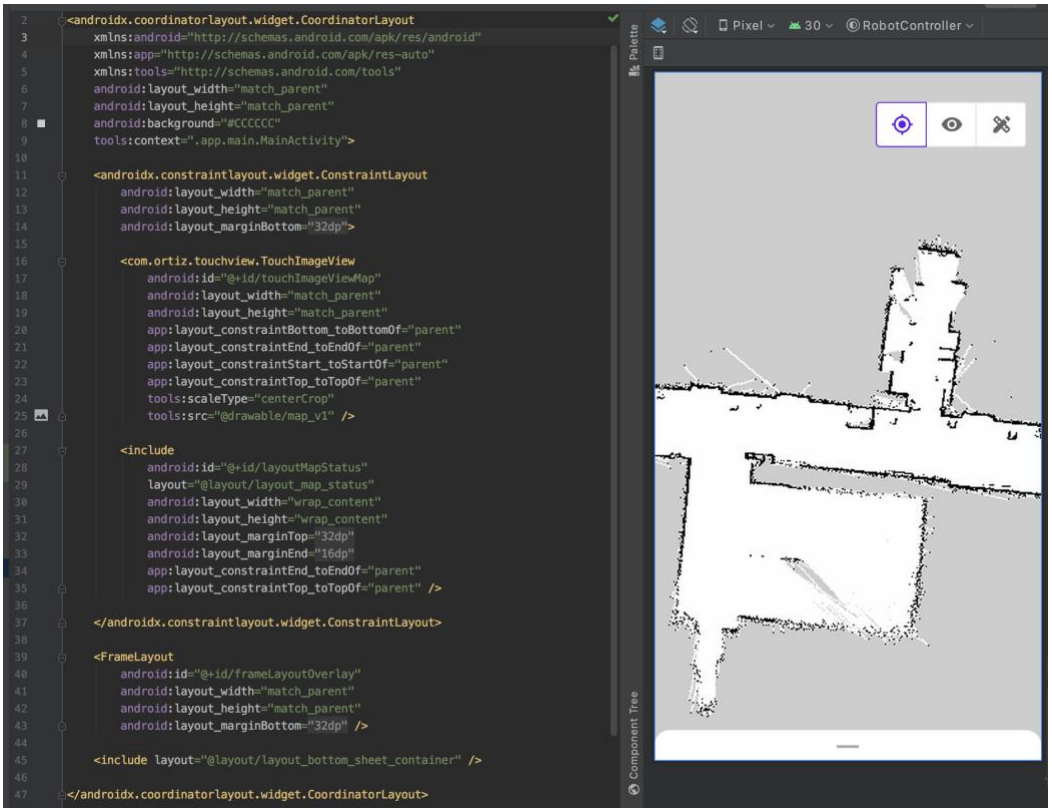


Slika 4.1. *Prikaz prototipa Robot controller aplikacije*

Slika iznad sadrži maketu svih ekrana koji se nalaze u aplikaciji i prikazuje sve vektorske grafike koje su dijeljene među njima. Prilikom paljenja aplikacije prikazan je početni ekran za prijavu iz kojeg je moguće doći na glavni ekran koji prikazuje preslikani prostor po kojem se robot kreće. Navigacija između ekrana koji se crtaju preko slike prostora ostvarena je uz pomoć *MaterialButtonToggleGroup* komponente koja sadrži tri gumba za navigaciju. Ekrane koje aplikacija sadrži su:

- Ekran za prijavu
- Prikaz trenutne pozicije robota i dijagnostičkih podataka
- Prikaz navigacijskih točaka
- Ekran za uređivanje barijera na karti

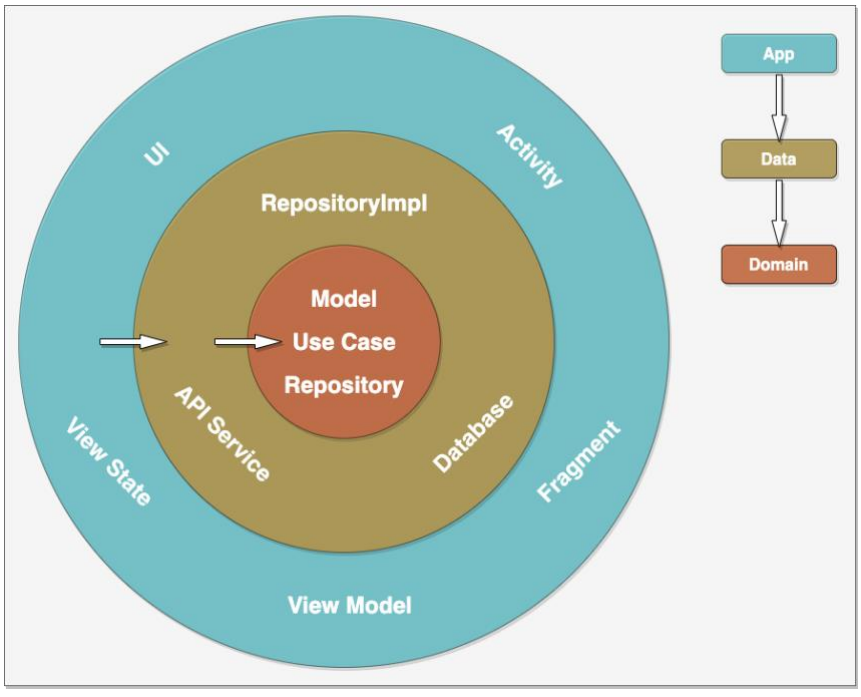
Izgled svakog ekrana i njegovih komponenti izrađuje se uz pomoć proširivog jezika za označavanje (engl. *Extensible Markup Language*, XML) podataka. XML je dizajniran za spremanje i prenošenje podataka, napravljen je da bude deskriptivan i za razliku od HTML koda ne sadrži predefinirane oznake. Navedena karakteristika dopušta korištenje oznaka koje predstavljaju vizualne komponente ekrana te mnoštvo drugih ugrađenih oznaka kao što su *<include/>*, *<merge/>*, *<style/>* i drugi [12]. Atributima unutar elemenata XML stabla pobliže se određuje njihova pozicija, dodaju se podatci i definira odnos s drugim komponentama na istoj razini u stablu. Slika **4.2**. prikazuje izgled glavnog ekrana u *Layout design preview* prozoru te prikazuje kod koji označava poziciju svake komponente i njihov odnos na ekranu. *Tools* imenski prostor (engl. *Namespace*) uz pomoć lažnih podataka olakšava predodžbu o tome kako će se komponenta prikazati prilikom pokretanja aplikacije. Oznaka *<include>* omogućava uključivanje komponenti koji se nalaze u drugim datotekama što olakšava preglednost datoteka jer ne sadrže puno linija, dopušta korištenje iste komponente u više ekrana te podršku implementacije iste komponente za različite dimenzije ekrana.



Slika 4.2. *Android Studio prozor s XML urednikom i preview prozorom*

4.3. Arhitektura sustava

Arhitektura čistog koda sadrži svoje prednosti koje doprinose kvaliteti izrade aplikacije. Slika **4.3**. prikazuje dijagram arhitekture, on se sastoji od jezgre koja predstavlja domenu u kojoj se nalazi poslovna logika, domenski modeli i slučajevi upotrebe (engl. *Use Cases*). Praćenjem ovisnosti, koje predstavljaju strjelice, sloj domene ne ovisi o ostatku sustava i moguće je otkriti temu i značajke projekta samim pogledom na modele i slučajeve upotrebe. Također sadrži i sučelja repozitorija čije implementacije uslužuje *Data* modul putem Koin definicija. U trenutnoj arhitekturi aplikacije ovaj modul ne sadrži ovisnosti na biblioteke Android sustava pa ga je moguće napraviti kao čisti Kotlin modul. Sljedeći sloj predstavlja *Data* modul koji sadrži implementacije repozitorija, dijelove odgovorne za HTTP komunikaciju sa servisom i implementaciju lokalne baze podataka. Posljednji sloj, *App* modul, predstavlja pogled prema korisniku. Sadrži sve komponente zaslužne za prikaz podataka korisniku kao i modele stanja pogleda (engl. *View State*). Ključna stvar koju prikazuje dijagram su strjelice okrenute prema središtu. Prikazuju smjer njihovih ovisnosti i ilustriraju kako vanjski slojevi znaju za unutarnje, ali unutarnji od njih ne ovise [13].



Slika 4.3. *Dijagram arhitekture aplikacije*

Prema dijagramu iz slike iznad, smišljena je arhitektura Robot Controller aplikacije. Ona se sastoji od podjele izvornog koda na 3 modula i predstavljaju ju:

- *App* modul – sadrži glavne komponente zaslužene za prikaz podataka, njihove modele stanja pogleda i paket koji sadrži definicije modula za umetanje ovisnosti.
- *Data* modul – sadrži klase zadužene za dohvaćanje i obradu podataka, dijelove za komunikaciju sa servisom i definicije svojih modula preko kojih pružaju svoja sučelja
- *Domain* modul – sadrži poslovnu logiku, slučajeve upotrebe i domenske modele

4.4. Prikaz preslikanog prostora

Unutar korisničkih zahtjeva definiran je format slike koja predstavlja preslikani prostor dobiven prolaskom robota kroz za to predviđene prostorije. Slanjem upita na servis kao odgovor dobiva se PGM (engl. *Portable Gray Map*) datoteka koja sadrži informacije o veličini slike i boji svakog piksela koji se u njoj nalaze. Postoje dva osnovna tipa zapisa PGM slike u datoteku, a to su P2 ili P5 tip zapisa [14], slika dolje prikazuje jednostavniji P2 tip.

```
P2
# feep.pgm
24 7
15
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  3  3  3  3  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0 15  0
0  3  3  3  0  0  0  7  7  7  0  0  0 11 11 11  0  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0  0  0
0  3  0  0  0  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

Slika 4.4. *Prikaz sadržaja P2 tipa zapisa PGM slike*

Prema slici iznad, PGM format zapisa predstavljen je datotekom čija struktura se sastoji od sljedećih sastavnica:

- Magičan broj (engl. *Magic number*) koji predstavlja tip datoteke.
- Neobavezni broj komentara koji započinju znakom #.
- Širinu i visinu slike predstavljene cijelim brojevima koji su odmaknuti praznim mjestom.
- Maksimalna vrijednost jednog piksela koja je veća od nula i manja od 65536.
- Niz redova kojih je onoliko koliko je slika visoka. Svaki redak se sastoji od niza vrijednosti, redom s lijeva u desno koje predstavljaju vrijednosti piksela jednog reda slike. U slučaju P5 tipa zapisa, vrijednosti su zapisane uređenim slijedom bitova, jedan bajt ako je maksimalna vrijednost manja od 256, a u suprotnom su zapisane s dva bajta.

Kako u trenutku pisanja rada ne postoji pouzdana vanjska biblioteka za učitavanje PGM formata slike, implementiran je pretvarač (engl. *Converter*) koji iz datoteke dobivene sa servisa kreira *Bitmap* objekt uz pomoću kojeg se slika prikazuje na korisnički ekran. Pretvarač je predstavljen sučeljem *PGMImageConverter*, definira jednu metodu *convert* koja kao parametar prima objekt klase *InputStream*, a kao rezultat vraća *Bitmap* objekt (Programski kod 4.1.). Fleksibilnost implementacije stvara mogućnost dodavanja podrške za druge tipove datoteka, a ona je ostvarena uz pomoć oblikovnog obrasca strategija (engl. *Strategy*). Obrazac strategije sugerira da, ako postoji klasa koja radi jednu stvar na više načina, u ovom primjeru kreiranje Bitmap objekta iz različitih tipova PGM datoteka, da se ti algoritmi izdvoje u zasebne klase nazvane strategije, predstavljene zajedničkim sučeljem. Izvorna klasa koja se naziva kontekst, u ovom slučaju implementacija *PGMImageConverter* klase, sadrži reference na različite strategije koje su predstavljene *PGMConverterStrategy* sučeljem. Kontekst zatim ovisno o tipu zapisa delegira posao na jednu od strategija umjesto da ga sam izvršava. Ovakav pristup omogućava dinamičku izmjenu strategija, laganu promjenu implementacije i dodavanje podrške za nove strategije.

```
interface PGMImageConverter {

    suspend fun convert(inputStream: InputStream): Bitmap
}

interface PGMConverterStrategy {

    suspend fun getBitmap(bufferedReader: BufferedReader): Bitmap
}
```

Programski kod 4.1. *Sučelja za pretvarač i strategiju*

Prilikom slanja zahtjeva za dohvaćanje preslikanog prostora na servis potrebno je dodati autorizacijski ključ unutar zaglavlja HTTP zahtjeva. Kako je autorizacija ostalih zahtjeva na servis riješena kreiranjem objekta *OkHttpClient* klase, potrebno je omogućiti njezinu integraciju s Glide bibliotekom za dohvaćanje i prikaz slika. Za uspješnu autorizaciju zahtjeva prvo je potrebno kreirati klasu koja nasljeđuje *AppGlideModule* te kreirati objekt klase *OkHttpUrlLoader.Factory* s uslužnim objektom klijenta servisa (Programski kod 4.2.). Registriranjem instance unutar *registerComponents* metode, autorizira se svaki zahtjev. Kako bi biblioteka Glide znala koristiti kreiranu klasu potrebno je iznad staviti anotaciju *@GlideModule* nakon čega će se, prilikom izvođenja procesora anotacija, generirati i koristiti novi kreirani modul za dohvaćanje slika.

```
@GlideModule
class GlideModule: AppGlideModule(), KoinComponent {

    private val okHttpClient by inject<OkHttpClient>()

    override fun registerComponents(context: Context, glide: Glide, registry: Registry) {
        val factory = OkHttpUrlLoader.Factory(okHttpClient)
        glide.registry.replace(GlideUrl::class.java, InputStream::class.java, factory)
    }
}
```

Programski kod 4.2. *Prikaz integracije OkHttp i Glide biblioteke*

4.5. Navigacijske točke

Glavna funkcija robota je prikupljanje podataka iz okoline dobivenih mjerenjem uz pomoć različitih senzora koji čine njegovo sklopovlje. Mjerenja se obavljaju na za to određenim pozicijama na karti koje se, prema korisničkim zahtjevima, mogu uređivati. Kako se na nekim pozicijama u prostoru može nalaziti poveći broj objekata ili kompleksnija struktura na koju robot treba posvetiti nešto više vremena prilikom mjerenja, tako je osmišljena značajka navigacijskih točaka. U stvarnom svijetu navigacijske točke predstavljaju pozicije u prostoriji na kojima robot radi mjerenja te ih šalje na servis.

Kako bi servis, i na kraju klijent, znali na kojim pozicijama se navedene točke nalaze, one su predstavljene koordinatnim točkama na prethodno izrađenom preslikanom prostoru. Slanjem zahtjeva ne servis vraća se lista navigacijskih točaka koje sadrže sve informacije potrebne za njihov prikaz. Slika **4.**5. prikazuje odgovor servisa u JSON formatu na putanju za dohvaćanje jedne navigacijske točke. Odgovor se sastoji od pozicije navigacijske točke na slici i od kuta pod kojim je zakrenuta.

```
{
  "position": {
    "x": 456,
    "y": 1034
  },
  "degrees": 76
}
```

Slika 4.5. *Zapis navigacijske točke unutar odgovora servisa*

Slika 4.6. *Ekran za prikaz navigacijskih točaka*

Pritiskom drugog navigacijskog gumba, prikazuju se ekran s navigacijskim točkama u ovisnosti o trenutno prikazanoj mapi prostora. **Slika 4.**6. prikazuje ekran koji sadrži navigacijske točke. Kako veličina slika izmjerenog prostora može varirati, tako je omogućena značajka mijenjanja veličine i pomicanje prikaza preslikanog prostora na ekranu. Mijenjanjem prikaza preslikanog prostora, mijenja se i prikaz svih trenutno nacrtanih navigacijskih točaka.

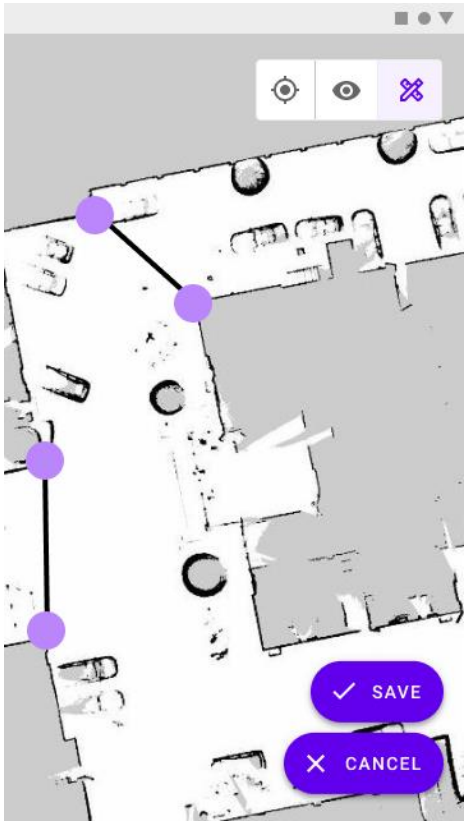
```
internal class NavigationPointsView @JvmOverloads constructor(
    context: Context,
    attrs: AttributeSet? = null,
    defStyleAttr: Int = 0
) : View(context, attrs, defStyleAttr) {

    private var navigationPoints: List<NavigationPointModel> = emptyList()

    override fun onDraw(canvas: Canvas?) = navigationPoints.forEach {
        canvas?.drawNavigationPoint(it)
    }

    private fun Canvas.drawNavigationPoint(navigationPoint: NavigationPointModel) {...}

    fun setNavigationPoints(points: List<NavigationPointModel>) {
        this.navigationPoints = points
        invalidate()
    }
}
```

Programski kod 4.3. *Komponenta za prikaz navigacijskih točaka*

Kako bi se navigacijske točke prikazale kao u definiranom dizajnu aplikacije, kreirana je posebna komponenta *NavigationPointsView* (Programski kod 4.3.). Klasa sadrži referencu na listu navigacijskih točaka koje se postavljaju pozivom metode *setNavigationPoints*. Metoda prima kolekciju *NavigationPointModel* objekata i osim postavljanja reference zove metodu *invalidate* kojom se osvježava prikaz. Unutar metode *onDraw,* za svaku navigacijsku točku u listi, poziva se metoda *drawNavigationPoint* koja je zadužena za crtanje jedne navigacijske točke.

4.6. Uređivanje karte

Kako nijedna prostorija ne sadrži idealan teren za ovu vrstu robotu, tako je potrebno osigurati neometan prolaz robotu između njegovih navigacijskih točaka. Za mjesta po kojima korisnik ne želi da se robot kreće su smišljene barijere uz pomoć kojih se uređuje karta i ograničava prostor po kojem se smije kretati. Barijere su predstavljene dvjema točkama čije koordinate su spremljene u bazi podataka. Servis pruža putanju za dohvaćanje liste barijera koju je zatim moguće uređivati putem klijentske aplikacije. Slika 4.7. prikazuje izgled ekrana za uređivanje karte koji na sebi sadrži dvije barijere. Na ekran se dolazi pritiskom na treći navigacijski gumb te se prikaz barijera crta preko već prikazane karte prostora.



Slika 4.7. *Ekran za uređivanje mape*

Pomicanjem barem jedne od krajnjih točaka barijera, prikazuju sve dva plutajuća akcijska gumba. Pritiskom na gumb *Save*, trenutno stanje barijera šalje se na servis i ažuriraju se podatci, a u slučaju da korisnik nije zadovoljan napravljenim izmjena, pritiskom na gumb *Cancel* se prikaz barijera vraća na početno stanje prije početka uređivanja.

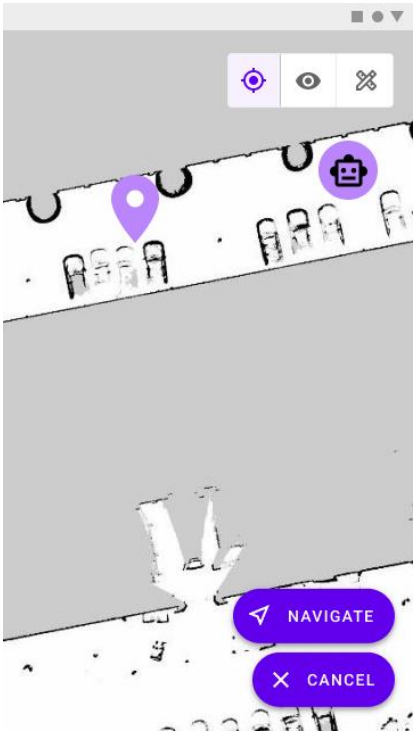
Kako bi se prikazale barijere koje se trenutno nalaze na karti, kreirana je posebna komponenta *MapBarriersView* (Programski kod 4.4.). Komponenta sadrži listu *MapBarrierModel* objekata koji se sastoje od koordinata početne i završne točke. Prilikom crtanja komponente na ekran u metodi *onDraw*, prolazi se kroz svaki objekt liste zasebno, crta se linija između početne i krajnje točke te kružnice na krajnjim točkama. Sve dimenzije i boje elemenata komponente su odvojene u resurse aplikacije te je tako omogućena promjena njihovih vrijednosti s jednog mjesta koja ima učinak kroz sve komponente koje ih koriste.

```
class MapBarriersView @JvmOverloads constructor(
    context: Context,
    attrs: AttributeSet? = null,
    defStyleAttr: Int = 0
) : View(context, attrs, defStyleAttr) {

    ...

    private var barriers: List<MapBarrierModel> = emptyList()

    init {...}

    override fun onTouchEvent(event: MotionEvent?): Boolean = false

    override fun onDraw(canvas: Canvas?) {
        super.onDraw(canvas)
        barriers.forEach {
            canvas?.drawBarrier(it)
        }
    }

    private fun Canvas.drawBarrier(mapBarrier: MapBarrierModel) {...}

    private fun Canvas.drawLine(startX: Int, startY: Int, endX: Int, endY: Int) {...}

    private fun Canvas.drawCircle(x: Int, y: Int) {...}

    fun setMapBarriers(barriers: List<MapBarrierModel>) {...}
}
```

Programski kod 4.4. *Komponenta za prikaz barijera*

4.7. Odlazak do tražene lokacije

Korisničkim zahtjevima definirana je mogućnost odabira lokacije na prikazanoj mapi i odlazak robota na izabranu destinaciju. Pritiskom na prvi navigacijski gumb prikazuje se ekran s trenutnom pozicijom robota. Bijela boja na karti predstavlja prostor po kojem se robot može gibati, dugim pritiskom na takvu lokaciju prikazuje se ikona koja označava odabrano mjesto i prikazuju se dva plutajuća akcijska gumba (Slika 4.8.). Pritiskom na gumb *Navigate* šalje se zahtjev nakon kojeg servis kontaktira robota i u slučaju da je upaljen, robot se kreće pomicati prema odabranoj lokaciji. Pritiskom na gumb *Cancel* skrivaju se akcijski gumbi i ikona s odabrane lokacije. Pomicanjem i mijenjanjem veličine prikaza prostora, tražena lokacija ostaje na istoj poziciji i akcijski gumbi ostaju prikazani korisniku.



Slika 4.8. *Prikaz ekrana za odlazak do tražene lokacije*

Za prikazivanje trenutne pozicije robota na karti kreirana je posebna komponenta *RobotPositionView* koja sadrži referencu na objekt klase *Point* (Programski kod 4.5.). Objekt se sastoji od x i y pozicije, koje se odnose na lokaciju u pikselima u ovisnosti o prikazanoj karti. Pozivanjem metode *onDraw*, na *canvas objektu*, crta se slika koja prikazuje robota ukoliko je njegova pozicija postavljena. Pozivom metode *setPosition* postavlja se trenutna pozicija robota te metodom *invalidate* osvježava se prikaz pozicije. Metoda za postavljanje trenutne pozicije robota zove se unutar fragmenta koji sadrži referencu na kreiranu komponentu sučelja.

```
internal class RobotPositionView @JvmOverloads constructor(
    context: Context,
    attrs: AttributeSet? = null,
    defStyleAttr: Int = 0
) : BaseOverlayView(context, attrs, defStyleAttr) {

    var listener: InteractionListener? = null
    private var robotPosition: Point? = null

    override fun onDraw(canvas: Canvas?) {
        super.onDraw(canvas)
        canvas?.drawRobot()
    }

    override fun onClick(position: PointF): Boolean {...}

    private fun Canvas.drawRobot() {...}

    fun setPosition(x: Int, y: Int) {
        this.robotPosition = Point(x, y)
        invalidate()
    }
}
```

Programski kod 4.5. *Komponenta za prikaz pozicije robota na karti*

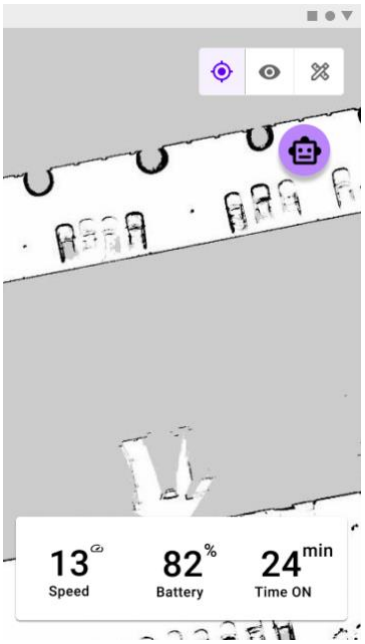
4.8. Prikaz dijagnostičkih podataka

Pritiskom na prvi navigacijski gumb te pritiskom na ikonu koja označava trenutnu poziciju robota, prikazuje se kartica s dijagnostičkim podatcima. Programski kod 4.5. prikazuje kako komponenta za prikazivanje pozicije sadrži referencu na objekt *InteractionListener* sučelja. Fragment postavlja navedenu referencu te se prilikom poziva implementirane metode, nakon detektiranja pritiska na trenutnu poziciju, poziva metoda *showInfoFragment* (Programski kod 4.6.).

```
private fun showInfoFragment() {
        childFragmentManager.beginTransaction()
            .setCustomAnimations(android.R.anim.fade_in, android.R.anim.fade_out)
            .replace(
                R.id.frameLayoutInfoContainer, InfoFragment.newInstance(), InfoFragment.TAG
            )
            .commitNow()
}
```

Programski kod 4.6. *Metoda za prikazivanje fragmenta s dijagnostičkim podatcima*

Slika 4.9. prikazuje ekran s otvorenom karticom koja se sastoji od informacija o trenutnoj brzini kojom se robot kreće, napunjenost baterije robota i vrijeme koliko je robot dugo upaljen izraženo u minutama. Navigacijom na neki drugi ekran unutar aplikacije ili pritiskom na mapu, skriva se kartica s dijagnostičkim podatcima. Programski kod 4.7. prikazuje fragment za prikaz dijagnostičkih podataka. Metoda *render* prima model stanja pogleda te postavlja tekst koji će se ispisati na *TextView* komponentama sučelja. Metoda se poziva iz roditeljske klase, u ovom slučaju svaki puta kada dođe do promjene lokacije.



Slika 4.9. *Prikaz dijagnostičkih podataka robota*

```
internal class InfoFragment private constructor()
    : BaseFragment<InfoViewState, FragmentInfoBinding>(FragmentInfoBinding::inflate) {

    companion object {
        const val TAG = "InfoFragment"

        fun newInstance(): Fragment = InfoFragment()
    }

    override val model by viewModel<InfoViewModel>()

    override fun FragmentInfoBinding.render(viewState: InfoViewState) {
        if (viewState is InfoViewState.Info) {
            val infoModel = viewState.infoModel
            textViewSpeed.text = getString(R.string.speed_format, infoModel.speed)
            textViewBattery.text = getString(R.string.battery_format, infoModel.battery.toInt())
            textViewTime.text = getString(R.string.time_on_format, infoModel.timeOn.toInt())
        }
    }
}
```

Programski kod 4.7. *Fragment za prikaz dijagnostičkih podataka*

5. TESTIRANJE APLIKACIJE

Ljudske greške prilikom implementacije dijelova programskog koda mogu loše utjecati na ponašanje sustava. Kako bi se ograničilo od takvih grešaka i spriječilo kreiranje novih prilikom mijenjanja postojećih funkcionalnosti potrebno je testirati programski kod. Pisanje testnih slučajeva i provedba testiranja imaju u vidu sljedeće ciljeve:

- Otkrivanje postojećih grešaka
- Smanjivanje učestalosti budućih grešaka
- Povećanje ukupne kvalitete programske podrške

Ručno testiranje aplikacije vrši se osobno i prateći korake testnog slučaja izvodi se testni scenarij. Takav tip testiranja vrlo je skup i dugotrajan jer zahtjeva ponovno izvođenje testa nakon promjene implementacije i naknadno zapisivanje stvarnih rezultata izvršenih testnih slučaja. Sklon je ljudskoj pogrešci prilikom testiranja ili izostavljanjem testnih koraka tijekom izvođenja. Zbog tih razloga uvodi se programsko testiranje koje je brže, efikasnije i rješava regresijske probleme.

```
internal class GetRobotInfoUseCaseTest {

    private val infoRepository = mock<InfoRepository> {
        on { getInfo() } doReturn(
            flow {
                emit(
                    InfoModel(
                        speed = 12.1f,
                        battery = 89f,
                        timeOn = 23f
                    )
                )
            }
        )
    }
    private val getInfoUseCase = GetInfoUseCase(infoRepository)

    @Test
    fun testGetRobotInfo() = runBlocking {
        val first = getInfoUseCase().first()
        Assert.assertEquals(12.1f, first.speed)
    }
}
```

Programski kod 5.1. *Prikaz načina primjene Mockito biblioteke*

Testiranje dijelova koda koje ovise o drugoj klasi implementirano je uz pomoć Mockito biblioteke. Biblioteka omogućava lažiranje (engl. *Mocking*) instance klase ovisnosti te lažiranje vrijednosti koje se vraćaju pozivima njezinih metoda. Prvi korak je poziv metode *mock* koja vraća instancu klase navedenu unutar izlomljenih zagrada. Unutar metode *on* obavlja se poziv metode koja se želi lažirati. Na kraju se vrijednost, koju se želi vratiti, stavlja unutar metode *doReturn*. Svakim sljedećim pozivom lažirane metode na instanci rezultirat će vraćanjem predane vrijednosti [15]. Programski kod 5.1. prikazuje lažiranje instance *InfoRepository* gdje je definirana vrijednost koja se vraća prilikom poziva metode *getInfo*. Unutar testne metode *testGetRobotInfo* nalazi se poziv na instancu klase koja ovisi o lažiranom objektu i provjerava se dobivena vrijednost.

Korištenjem Mockito biblioteke kreiran je test za implementirani pretvarač PGM datoteka. Programski kod 5.2. prikazuje klasu *PGMImageConverterTest* s metodom *testImageInvalid* koja predstavlja jedinični (engl. *Unit*) test za provjeru baca li pretvarač iznimku u slučaju neispravnog parametra predanog u *convert* metodu i baca li dobru iznimku. Jedinični testovi namijenjeni su za testiranje manjih cjelina kao što je navedena metoda. Nakon izmjene implementacije pretvarača, dovoljno je ponovno pokrenuti test i provjeriti radi li ispravno spomenuti testni slučaj.

```
class PGMImageConverterTest {

    private val converter = PGMImageConverterImpl(
        p2Strategy = mock(),
        p5Strategy = mock(),
    )

    @Test
    fun testImageInvalid() = runBlocking {
        try {
            getInvalidInputStream().use { converter.convert(it) }
            Assert.fail()
        } catch (e: Exception) {
            assertTrue(e is PGMConverterException.InvalidFormatException)
        }
    }

    private fun getInvalidInputStream(): InputStream = ByteArrayInputStream(ByteArray(0))
}
```

Programski kod 5.2. *Jedinični test pretvarača slike za neispravni parametar*

6. ZAKLJUČAK

Popularnost mobilnih uređaja raste iz dana u dan te je skoro pa nemoguće pronaći osobu koja ne posjeduje svoj pametni uređaj. Razvojem tehnologije, primjena mobilnih uređaja se širi i na kontrolu naprednih robotskih sustava koji postaju sve pristupačniji. Napredak znanosti i povećanje zajednice stručnjaka, koji sudjeluju na projektima otvorenog koda, doprinose poboljšanju platformi za kreiranje robotskih sustava,

Pružanjem usluga preko apstrakcije kompleksnih sustava, komunikacijom s vanjskom okolinom ili servisima, olakšava se povezivanje te jednostavnost upravljanja prilikom obavljanja zadataka. Cilj rada bio je razviti aplikaciju koja će omogućiti korisniku nadzor i upravljanje jednim takvim robotskim sustavom. Sustav, za kojeg je aplikacije razvijena, sam brine o kretanju kroz prostorije u kojima se nalazi te lokacijske podatke putem web servisa šalje klijentskoj strani. Obavljajući mjerenja putem senzora robota spremaju se rezultati i putem web servisa aplikacija dohvaća vrijednosti koje se prikazuju na korisničkom sučelju.

Razradom korisničkih zahtjeva i definiranjem izgleda korisničkog sučelja olakšana je faza razvoja. Podjelom programskog koda na što više manjih cjelina i praćenjem arhitekture čistog koda, olakšano je testiranje i pružena je mogućnost za daljnju nadogradnju sustava. Testiranjem je osigurana kvaliteta izvornog koda i spriječeni su regresijski problemi nakon promjene postojeće implementacije. Značajke aplikacije prikazane u radu samo su temelj buduće podrške koju bi robot mogao pružati. Kasnijim inačicama robota i njegovog programskog sučelja, otvaraju se mogućnosti daljnje nadogradnje mobilne aplikacije koju dopuštaju implementirana arhitektura i postavljeni standardi.

LITERATURA

[1] S., Mathanraj, Introduction to Robotic Control Systems, Medium, dostupno na: https://towardsdatascience.com/introduction-to-robotic-control-systems-9ec17c8ac24f [5.6.2021.]

[2] SPOT, Boston Dynamics, dostupno na: https://shop.bostondynamics.com/spot [22.6.2021.]

[3] B. Gerkey, M. L. Quigley, W. D. Smart, Programming Robots with ROS: A Practical Introduction to the Robot operating System, O'Reilly Media, 2016.

[4] I. Krajci, D. Cummings, Android on x86, Apress, Berkley, CA, 2013.

[5] Discover IntelliJ Idea, JetBrains, dostupno na: https://www.jetbrains.com/help/idea/discover-intellij-idea.html [17.6.2021.]

[6] Why Kotlin, JetBrains, dostupno na: https://kotlinlang.org/#why-kotlin [17.6.2021.]

[7] F. Babić, N. Srivastava, Kotlin Coroutines by Tutorials, Razeware LLC, 2016.

[8] Inversion of Control Container and the Dependency Injection pattern, Martin Fowler, dostupno na: https://martinfowler.com/articles/injection.html [19.6.2021.]

[9] Andoid – SQLite Database, Tutorialspoint, dostupno na: https://www.tutorialspoint.com/android/android_sqlite_database [22.6.2021.]

[10] Consuming APIs with Retrofit, CodePath, dostupno na: https://guides.codepath.com/android/consuming-apis-with-retrofit [24.6.2021.]

[11] Creative tools meet the Internet, Figma, dostupno na: https://www.figma.com/about/ [25.6.2021.]

[12] Introduction to XML, W3Schools, dostupno na: https://www.w3schools.com/xml/xml_whatis.asp [25.6.2021.]

[13] The Clean Architecture, The Clean Code Blog, dostupno na: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html [26.6.2021.]

[14] .PGM File Extension, FileInfo.com, dostupno na: https://fileinfo.com/extension/pgm [27.6.2021.]

[15] Tasty Mocking framework for unit tests in Java, Mockito, dostupno na: https://site.mockito.org [27.6.2021.]

SAŽETAK

Razvojem tehnologije, pruža se i podrška za upravljanje naprednim robotskim sustavima putem mobilnih aplikacija. Pristupom apstrakciji kompleksnih robotskih sustava preko sučelja omogućena je kontrola i način dohvaćanja očitavanja senzora. Cilj rada bio je implementacija mobilne aplikacije koja korisnicima omogućava upravljanje i nadzor takvim robotskim sustavom. Na početku rada opisani su općeniti robotski sustavi, tehnologije koje se koriste za izradu i naveden je primjer najpoznatije platforme za razvoj. Opisan je način upravljanja putem mobilnih aplikacija i predstavljen je zadatak rada. Nakon toga dan je uvod u android platformu te su opisane korištene tehnologije i biblioteke potrebne za izradu aplikacije. Definirana je modularna arhitektura programskog koda i organizacija datoteka u projektu. Na kraju detaljnije je opisana implementacija, prikazani su svi ekrani i pojašnjena je važnost testiranja programske podrške uz odgovarajući primjer.

Ključne riječi: Android, Android Studio, Kotlin, mobilna aplikacija, robotski sustavi

ABSTRACT

With the development of technology, support is also provided for controlling advanced robotic systems via mobile applications. Accessing the abstraction of complex robotic systems via the interfaces enables the control and the way of retrieving sensor readings. The goal of this assignment was to develop a mobile application that allows users to control and supervise such a robotic system. At the beginning of the paper, general robotic systems and technologies used for construction were described and an example of the most famous development platform was given. The method of management via mobile applications was described and the assignment task was presented. After that, an introduction to the android platform was given, and the technologies used and the libraries needed to create the application were described. The modular architecture of the program code and file organisation in the project were defined. Finally, the implementation was described in more detail, all screens were shown and the importance of software testing was explained with an appropriate example.

Keywords: Android, Android Studio, Kotlin, mobile application, Robotic Systems

ŽIVOTOPIS

Luka Perišić rođen je 17.10.1997. u Slavonskom Brodu. Pohađao je osnovnu školu Matija Antun Reljković, u Cerni. Nakon završetka osnovne škole upisuje Gimnaziju Matije Antuna Reljkovića, u Vinkovcima, prirodoslovno-matematički smjer. Sredinom 2016. godine polaže maturu te iste godine upisuje preddiplomski sveučilišni studij na Fakultetu za elektrotehniku, računarstvo i informacijske tehnologije Osijek, smjer Računarstvo. Stječe status sveučilišnog prvostupnika računarstva 2019. godine. Trenutno radi kao student Android developer u tvrtki 5 minuta.

Luka Perišić