Usporedba ORM alata pri trajnoj pohrani podataka unutar Android aplikacije

Mariić, Ivan

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

Sveučilišni studij

USPOREDBA ORM ALATA PRI TRAJNOJ POHRANI PODATAKA UNUTAR ANDROID APLIKACIJE

Završni rad

Ivan Mariić

Osijek, 2018.

SADRŽAJ

1. UVOD

Većina mobilnih uređaja pogonjena je Android operacijskim sustavom. Android napreduje svakim danom zbog toga što su potrebe korisnika veće i zahtjevnije. Jedna od najbitnijih potreba korisnika trajna je pohrana podataka unutar aplikacije. Podaci su trajno spremljeni kada se nalaze u memoriji nakon izlaska iz aplikacije. Takav način spremanja podataka bit će detaljno obrađen u ovome radu. Postoji više načina kako će se podaci očuvati, a jedan od njih su baze podataka. Android dolazi s ugrađenom bazom kojoj je moguće pristupiti preko aplikacije. Rad s njom zna nekad biti složen i spor pa se zbog toga koriste alati za objektno-relacijsko preslikavanje (engl. *object-relational mapping*, ORM).

U drugom će se poglavlju opisati struktura Androida i načini trajne pohrane podataka, kasnije će biti detaljno objašnjeno objektno-relacijsko preslikavanje te će se spomenuti i objasniti odabrani ORM alati za navedeni operacijski sustav. U trećem poglavlju rada izradit će se aplikacija u kojoj će biti korišteni ti ORM alati. Aplikacija će sadržavati mogućnost za unos, čitanje i pisanje podataka u bazu. Zatim će se mjeriti vrijeme izvršavanja spomenutih operacija. Na kraju će biti vidljiv grafički prikaz mjerenja pomoću kojeg će se moći odlučiti koji je alat obavio bolje određeni posao.

1.1 Zadatak završnog rada

U teorijskom dijelu rada potrebno je opisati mogućnosti trajne pohrane podataka prilikom razvoja aplikacija za Android platformu, a posebice alate za objektno-relacijsko preslikavanje. Opisati karakteristike, mogućnosti i načine korištenja ovih alata prilikom razvoja. U praktičnom dijelu rada ostvariti programsko rješenje u vidu aplikacije za Android operacijski sustav unutar kojeg je potrebno ugraditi različite alate za objektno-relacijsko preslikavanje.

2. OČUVANJE STANJA U ANDROID APLIKACIJAMA

U ovom će se poglavlju govoriti o strukturi i povijesti Android platforme, građi aplikacija i trajnoj pohrani podataka. Pametni telefoni su još uvijek relativno "novi" uređaji, ali unaprjeđuju se svakodnevno što znači da dolazi do promjena u strukturi platforme. Isto tako na dnevnoj bazi izlaze nove biblioteke koje donose nove funkcionalnosti aplikacijama npr. biblioteke za rad s ORM alatima.

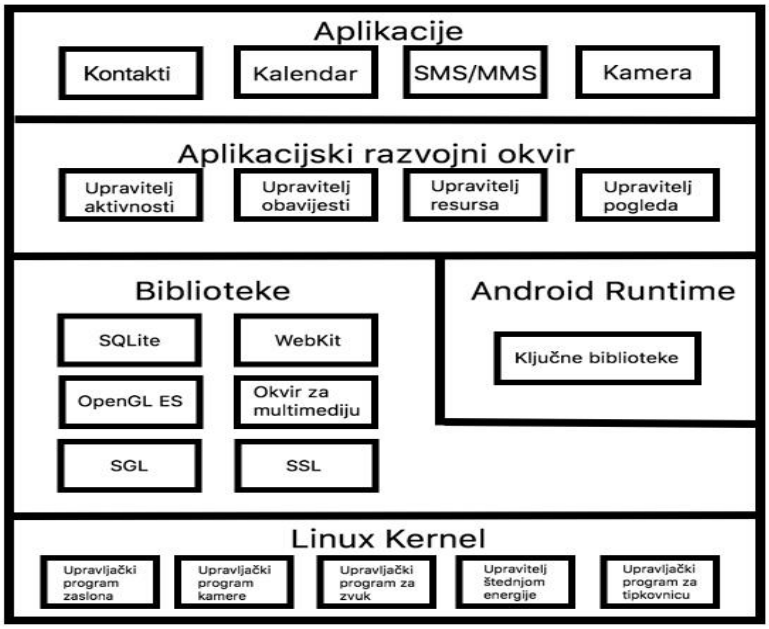
2.1 Povijest Android platforme

Android [1] je mobilni operacijski sustav kojeg je razvio Google. Temelji se na Linux jezgri. Primarno se koristi u uređajima osjetljivim na dodir poput pametnih telefona i tableta. Osim toga, Google je dodatno razvio Android TV za televizore, Android Auto za aute i *Wear* OS za pametne satove. Varijacije Androida koriste se i na igraćim konzolama, digitalnim fotoaparatima, računalima i drugim elektroničkim uređajima. Na početku, Android je razvijala tvrtka Android Inc. Google je kupuje 2007. godine. Prvi komercijalni Android uređaj je bio HTC *Dream* koji je ugledao svjetlo dana u rujnu 2008. godine. Operacijski sustav je tada prošao kroz više verzija, a trenutna je 9.0 kodnog imena *Pie.* Verzije se imenuju po slatkišima abecednim redom.

2.2 Arhitektura Android platforme

Slika 2.1 [2] prikazuje arhitekturu Android platforme pomoću stoga (engl. *stack*). Na vrhu se nalaze osnovne aplikacije koje dolaze sa svakim Android pametnim telefonom, a to su aplikacije za slanje elektronske pošte, pretraživanje na internetu, slanje SMS poruka itd. Te aplikacije nemaju nikakav poseban status u odnosu na aplikacije koje korisnik preuzme putem *Google Play* trgovine što znači da on može preuzeti aplikaciju za pretraživanje Interneta i postaviti je za zadanu aplikaciju. Ako korisnik gradi svoju aplikaciju i želi joj omogućiti pisanje SMS poruka to može napraviti pokretanjem sustavske aplikacije unutar svoje. Ispod aplikacijskog sloja nalazi se Aplikacijski razvojni okvir, on omogućuje korištenje istog aplikacijskog programskog sučelja (engl. *Application Programmable Interface,* API) koje je korišten u izgradnji osnovnih Android aplikacija. API-ji su pisani u Javi. Na idućoj razini smještene su biblioteke i *Android Runtime* (ART). Biblioteke su napisane u programskim jezicima C i C++ i omogućuju korištenje drugih mogućnosti koje nisu usko vezane za Android npr. *SQLite* baze podataka. ART dolazi s listom ključnih biblioteka koje sadrže skoro sve pogodnosti Java programskog jezika. Za uređaje pokretane Android verzijom 5.0

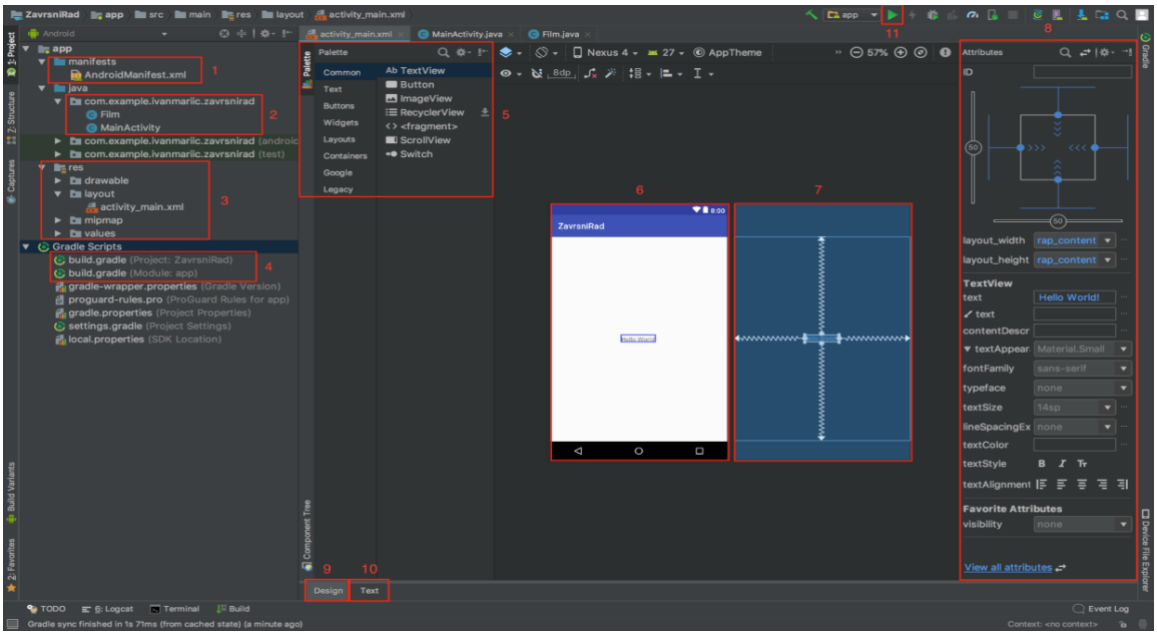
ili kasnijom verzijom, svaka je aplikacija pokretana unutar vlastitog procesa s vlastitom instancom ART-a. Na dnu stoga nalazi se Linux *Kernel*. On je odgovoran za upravljačke programe i ključne servise kao što su sigurnost sustava i upravljanje memorijom.



Slika 2.1 Prikaz arhitekture Android platforme [2]

2.3 Struktura i razvoj Android aplikacija

Za razvoj Android aplikacija potrebno je osnovno znanje Jave i XML opisnog jezika (engl. *eXtensible Markup Language*), programsko okruženje i dobra ideja za aplikaciju. U ovom će se radu koristiti Android Studio. Android Studio službeno je integrirano razvojno okruženje (engl. *Integrated Development Environment*, IDE) informatičkog diva Googlea. Dizajnirano je za razvoj Android aplikacija s podrškom za Windows, Linux i macOS platforme. Najvažniji dijelovi svake Android aplikacije su: Aktivnosti (engl. *Activity*), izgledi (engl. *Layout*), resursi (engl. *Resources*) i manifest. Na Slici 2.2 prikazan je zaslon Android Studija zajedno s objašnjenjem svake komponente.



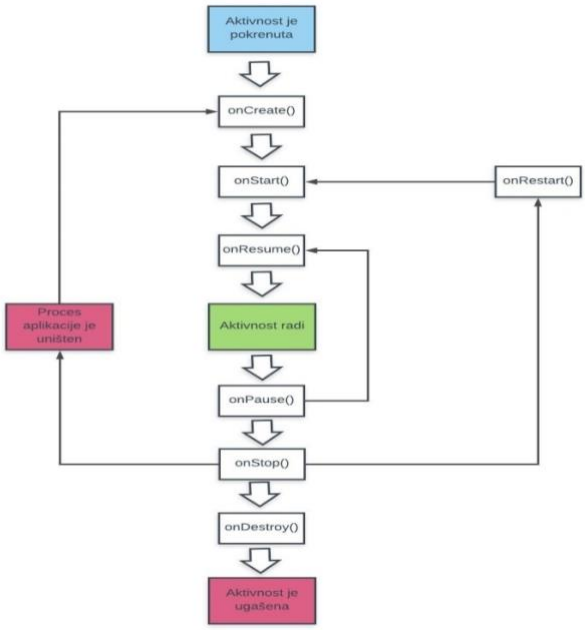
Slika 2.2 Prikaz sučelja Android Studija

1. Prikazuje direktorij u kojem je spremljena Manifest datoteka
2. Direktorij u kojem se nalaze aktivnosti, klase, sučelja...
3. Svi resursi aplikacije: dimenzije, izgledi, slike, zvukovi, stringovi, boje...
4. U *Gradle build* datoteke unose se sve vanjske biblioteke koje će biti korištene u aplikaciji.
5. Paleta sadrži sve elemente koje aplikacija može sadržavati.
6. *Design View* koji prikazuje kako bi aplikacija trebala izgledati na stvarnom uređaju
7. *Blueprint View* prikazuje strukturu *Izgleda*
8. Prikaz svih atributa i opcija za odabrani element aplikacije
9. Kartica koja omogućuje prelazak u *Design View*
10. Kartica koja omogućuje prelazak u *XML View*
11. Gumb za pokretanje aplikacije

2.3.1 Aktivnosti

Aktivnost [3] je osnovna komponenta Android aplikacije i predstavlja njezin zaslon. Unutar aktivnosti definira se logika aplikacije. Svaka aktivnost ima svoj životni vijek i on je prikazan na slici 2.3. Životni vijek se sastoji od stanja u kojima aktivnost može biti. Kretanje između stanja postiže se pozivanjem odzivnih (engl. *callback)* metoda, a to su *onCreate(), onStart(), onResume(), onPause(),onRestart(), onStop(), onDestroy().*

- *onCreate()* je metoda koja se mora implementirati, ona je pozvana kada sustav prvi puta kreira aplikaciju.
- *onStart()* je metoda koja je implementirana od sustava. Pozivanjem ove metode aktivnost postaje vidljiva korisniku i priprema se za interakciju.
- *onResume()* je pozvana kada je aktivnost potpuno spremna za interakciju s korisnikom. Aktivnost ostaje u ovom stanju sve dok se ne dogodi nešto što bi moglo poremetiti fokus, npr. dolazni poziv.
- pozivanje *onPause()* metode je prvi znak da korisnik napušta trenutnu aktivnost. Nakon pozivanja ove metode, aktivnost je vidljiva na zaslonu, ali nije u fokusu.
- izvršavanjem *onStop()* metode aktivnost više nije vidljiva korisniku
- *onDestroy()* je pozvana neposredno prije uništavanja aktivnosti. Postoji nekoliko situacija kada se izvršava ova metoda, a to su: želja korisnika za uništavanje aktivnosti (dodirivanjem gumba za nazad na uređaju), promjena konfiguracijskih postavki (npr. rotiranje zaslona) i želja sustava za oslobađanjem resursa.



Slika 2.3 Životni vijek aktivnosti [3]

2.3.2 Izgled

Layout datoteke [3] predstavljaju izgled komponente aplikacije. On može biti izgrađen pomoću *Design Viewa* i XML-a. Kreiranje unutar *Design Viewa* vrši se na taj način da korisnik odabere komponentu unutar palete i drži lijevi klik miša kako bi je prenio na željeno mjesto, nakon što je to

mjesto odabrano korisnik otpušta lijevi klik i tada je komponenta uspješno postavljena. Postoji još jedan način izgradnje *layouta*, a to je dizajniranje pomoću XML-a. Taj način se svodi na pisanje koda u navedenom jeziku i nudi veću fleksibilnost. Svi elementi unutar l*ayouta* su izgrađeni pomoću hijerarhije pogleda (engl. *View*) i grupe pogleda (engl. *ViewGroup*) objekata gdje je *ViewGroup* nevidljivi kontejner koji definira strukturu *layouta* i poziciju *Viewa* i ostalih *ViewGroup* objekata.

2.3.3 Resursi

Resursi [3] su svi vanjski elementi koji će biti korišteni u aplikaciji, a to su npr. zvuk i slike. U resursima se mogu spremiti svi stringovi koji će biti korišteni u aplikaciji. Svaki string će imati svoj identifikacijski kod preko kojeg je moguć pristup unutar XML-a ili aktivnosti. Takvo spremanje stringova omogućuje korištenje više jezika u aplikaciji. To znači da korisnik u aplikaciji neće morati ručno prevoditi svaki string na svakoj komponenti nego će morati samo prevesti stringove u resursima. Također, u resursima su spremljene veličine poput boja, dimenzija i *layout* datoteke.

2.3.4 Manifest

Manifest [3] je XML datoteka koju svaka aplikacija mora imati i u njoj se nalaze sve bitne informacije o aplikaciji kao što su:

- Ime paketa koje služi za jedinstvenu identifikaciju aplikacije
- Komponente koje aplikacija koristi npr. aktivnosti, servisi, primatelji odašiljanja i pružatelji sadržaja.
- Dozvole koje aplikacija treba imati kako bi pristupila zaštićenim dijelovima sustava

2.4 Trajna pohrana podataka

Trajna pohrana podataka [3] je način spremanja podataka tako da oni ostanu spremljeni nakon izlaska iz aplikacije. Na Android platformi postoji nekoliko načina trajnog spremanja podataka. Spremanje podataka ovisi o njihovim svojstvima, a to su: veličina podataka koje korisnik želi spremiti, tip podataka koje korisnik želi spremiti, prava pristupa podacima tj. jesu li podaci dostupni drugim korisnicima ili aplikacijama.
Najčešći oblici spremanja podataka su:

- Unutrašnji datotečni spremnik
- Vanjski datotečni spremnik

- Dijeljene postavke (engl. *Shared preferences*)
- Baze podataka

Navedeni oblici, osim vanjskog datotečnog spremnika namijenjeni su za spremanje privatnih podataka unutar aplikacije. Privatnim podacima je nemoguće pristupiti iz drugih aplikacija. Ako korisnik baš želi podijeliti podatke s drugim aplikacijama to može učiniti pomoću pružatelja sadržaja.

2.4.1 Unutrašnji datotečni spremnik

Podaci spremljeni u unutrašnji datotečni spremnik [4] su vidljivi samo korisničkoj aplikaciji i druge aplikacije im ne mogu pristupiti. Android sustav omogućuje unutrašnji datotečni sustav za svaku aplikaciju na pametnom telefonu. Kada korisnik obriše aplikaciju tada se svi podaci koji su spremljeni u unutrašnji datotečni spremnik brišu. Zbog toga se ne preporuča korištenje ovog načina spremanja podataka ako korisnik želi da podaci ostanu sačuvani čak i nakon brisanja aplikacije. Prvi korak u direktnom spremanju podataka je pozivanje metode *getFilesDir()*, ona vraća datoteku (engl. *File*) koja predstavlja unutrašnji direktorij za aplikaciju. Nakon toga se zapis vrši pomoću objekta toka (engl. *Stream)* preko čije se reference poziva metoda *write()*.

2.4.2 Vanjski datotečni spremnik

Svaki Android uređaj podržava korištenje vanjskog datotečnog spremnika [4] za pohranu podataka. Dobio je ime po tome što korisnik nema zagarantiran pristup spremniku. Pristup ovisi je li spremnik spojen s uređajem ili ne (npr. SD kartica). Ovaj je spremnik najbolje koristiti za spremanje datoteka kao što su fotografije i video zapisi, zbog toga što oni zauzimaju puno prostora, a i korisnik ih tada može lako prebaciti na drugi uređaj. Zanimljivost ovakvog sustava je što omogućuje premještanje dijelova aplikacije koji zauzimaju mnogo prostora iz unutrašnjeg spremnika u vanjski. U slučaju da korisnik obriše aplikaciju, svi njezini podaci iz oba spremnika će biti obrisani. Za spremanje podatka pomoću programskog koda, korisnik prvo mora zahtijevati dopuštenje unutar manifesta. Nakon toga u kodu provjerava je li vanjski datotečni spremnik dostupan. Zatim se poziva metoda *Environment.getExternalPublicDirectory()* koja vraća objekt tipa *File* koji predstavlja određeni direktorij u spremniku. Tek tada je moguće vršiti zapis.

2.4.3 Dijeljene postavke

Ako korisnik ne treba spremiti veliku količinu podataka tada se koristi *SharedPreferences* API [4, 5]. On omogućava čitanje i pisanje primitivnih tipova podataka u obliku ključ-vrijednost. Za korištenje ovakvog načina spremanja podataka koristi se metoda *getSharedPreferences()* koja prima ime dijeljenih postavki kao prvi i način otvaranja kao drugi parametar. U izlistanju koda 2.1 prikazan je način definiranja dijeljenih postavki.

```
Context ivanovContext = MainActivity.this;
SharedPreferences sharedPreferences = ivanovContext.getSharedPreferences("IvanovePrefs",
Context.MODE_PRIVATE);
```

Izlistanje koda 2.1 Primjer dijeljenih postavki

Način otvaranja specificira koji svi korisnici mogu pristupiti postavkama. Postoji nekoliko vrsta načina otvaranja, a najbitniji od njih su:

- MODE_PRIVATE govori da postavkama može pristupiti samo pozivajuća aplikacija.
- MODE_WORLD_READABLE omogućuje da postavkama mogu pristupiti i druge aplikacije
- MODE_WORLD_WRITEABLE specificira da u postavkama mogu vršiti izmjene i ostale aplikacije.
- MODE_APPEND ako su postavke već kreirane, zapis će biti izvršen na kraju datoteke.

Za vršenje izmjena u postavkama, koristi se sučelje *SharedPreferences.Editor*. Ovdje je bitno za naglasiti da promjene neće biti izvršene ako se na kraju ne pozove metoda *commit()*. U izlistanju koda 2.2 prikazan je unos imena i prezimena u dijeljenje postavke.

```
SharedPreferences.Editor editor = sharedPreferences.edit();
editor.putString("ime", "Ivan");
editor.putString("prezime", "Mariić");
editor.commit();
```

Izlistanje koda 2.2 primjer korištenja *Editor* sučelja

2.4.4 Baze podataka

Baza se podataka [4, 5] može definirati kao kolekcija međusobno povezanih podataka. One se razvijaju zbog toga što je potrebno odvojiti podatke od aplikacije koja ih koristi. Postoji nekoliko vrsta modela baza podataka:

- Hijerarhijske

- Mrežne
- Relacijske
- Relacijske baze s objektno-orijentiranim proširenjima
- Objektno orijentirane baze podataka

U ovom će se radu obrađivati relacijske baze podataka. Kod relacijskih baza podataka podaci su organizirani u obliku tablica koje mogu biti međusobno povezane. Stupci tablice predstavljaju atribute koji služe za opis stanja entiteta, a redovi su n-torke. Entitet je stvarni objekt čije će informacije biti spremljene u bazi. Tablice se međusobno povezuju pomoću stranih i primarnih ključeva. Primarni ključ jedinstveno identificira red tablice. Strani ključ je definiran u drugoj tablici, ali se referira na primarni ključ u prvoj tablici što znači da oni imaju jednaku vrijednosti i na taj je način uspostavljena veza između tablica. Općenito, za programiranje i upravljanje podacima spremljenim u sustavu za upravljanje bazom podataka (engl. *Database Management System*, DBMS) koristi se strukturni jezik za pretraživanje (engl. *Structured Query Language*, SQL). Android ima potpunu podršku za jednostavniju i lakšu verziju SQL-a koja se naziva *SQLite*. Prednosti *SQLitea*:

- visoke performanse
- zauzima malo memorije
- ne zahtijeva posebnu konfiguraciju
- podrška za transakcije
- brzina
- koristi se na više različitih platformi (Android, iOS, Linux, macOS, Windows...)

Na Android platformi cijela je baza podataka smještena na jednom mjestu na disku. U primjerima ovog odlomka može se vidjeti da je rad sa *SQLiteom* složen i zahtijeva mnogo posla. Svaka takva baza podataka sastoji se od:

- Datoteke baze podataka (engl. *database file*)- glavna datoteka baze i u njoj su pohranjeni svi podaci.
- *Journal* datoteke- datoteka koja ima isto ime kao i baza samo što uz to ima prefiks *-journal*. Sadrži sve izmjene napravljene u bazi. Ako nastane problem, ona će biti korištena za poništavanje izmjena. Četiri osnovne klase za rad s *SQLite* bazama podataka su:
- *SQLiteOpenHelper* klasa- Nasljeđivanjem ove klase kreira se vlastiti pomagač (engl. *helper*). On tada omogućuje kreiranje, upravljanje i nadograđivanje baze podataka. Mora sadržavati

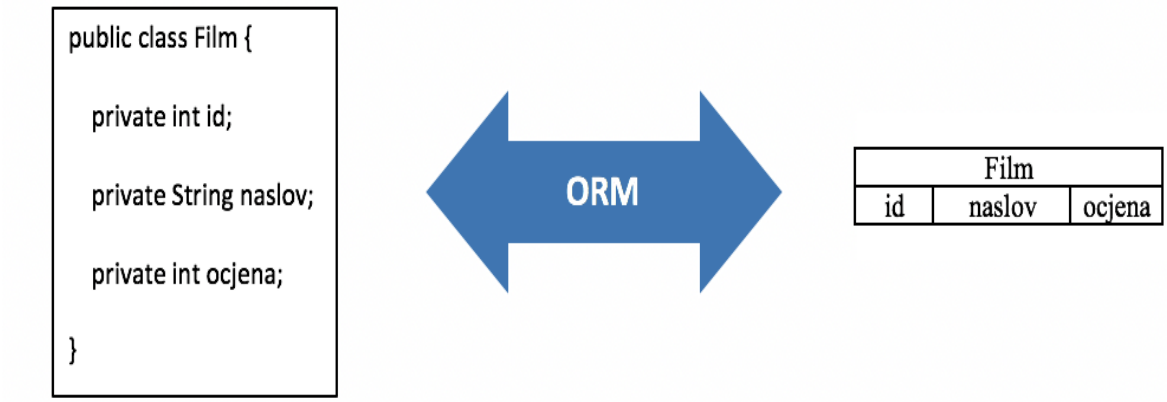
onCreate() i *onUpgrade()* metode, *onCreate()* se poziva kada je baza prvi puta kreirana na uređaju. Ta metoda mora sadržavati sav kod potreban za kreiranje tablica koje će koristiti aplikacija, *onUpgrade()* je pozvana kada baza treba biti nadograđena. Unutar definicije konstruktora potrebno je predati ime i verziju baze kao parametre metode *super()*. Ako to nije učinjeno tada baza nije spremljena na uređaju, nego samo u memoriji. Kako bi se spriječilo curenje memorije, preporuča se korištenje *Singleton* oblikovnog obrazca. *Singleton* sprječava korisnika da kreira više objekata istog tipa, u ovom slučaju je to *SQLiteOpenHelper* objekt. To se postiže deklariranjem statičke instancne varijable i provjeravanjem unutar konstruktora je li ta varijabla inicijalizirana, ako nije tada se ona inicijalizira, ako jest, onda se ona vraća.

- *SQLiteDatabase* klasa- Ova klasa predstavlja bazu podataka. Sadrži metode za direktan rad s bazom, npr. *execSQL()* koja služi za izvršavanje SQL upita.
- *Cursor* klasa- Objekt se kreira pozivanjem metode *query() SQLiteDatabase* objekta. Kao parametri se predaju ime tablice, imena atributa i uvjeti. Zatim se pretražuje tablica i vraćaju se atributi koji zadovoljavaju uvjete. Ti su atributi spremljeni u *Cursor* objekt. Za čitanje zapisa iz *Cursor* objekta potrebno je vršiti usmjeravanje na željeni zapis. Usmjeravanje je omogućeno pozivanjem metoda *moveToFirst(), moveToLast(), moveToPrevious(), moveToNext()*. Nakon što je *Cursor* usmjeren, poziva se metoda *get()* za dohvaćanje podataka, njezin parametar je indeks stupca čije vrijednosti treba dohvatiti. Kada se čitanje završi, *Cursor* se mora zatvoriti pozivanjem metode *close()*.
- *ContentValues* klasa- Objekt ove klase opisuje tip podataka koji će biti uneseni u bazu. Podaci se dodaju pozivanjem metode *put()*, prvi je parametar ime stupca u kojeg se unose podaci, a drugi je vrijednost tog podatka. Na kraju se za unos u bazu poziva metoda *insert() SQLiteDatabase* klase koja prima *ContentValues* objekt.

3. ORM PRESLIKAVANJE

ORM [6, 7] tehnika je programiranja koja omogućuje pretvorbu objekata u entitete baze podataka i obrnuto. Ova se tehnika koristi u objektno-orijentiranim programskim jezicima. Svaki se objekt sastoji od stanja i ponašanja. Stanju pripadaju atributi koji opisuju objekt, a ponašanje čine sve metode koje opisuju što sve taj objekt može raditi. Postavlja se pitanje kako je moguće očuvati stanje tog objekta pomoću relacijskih baza podatka. Odgovor na to daju alati za objektno-relacijsko preslikavanje. ORM predstavlja sloj između baze i objekata preko kojeg je moguće upravljati bazom na vrlo jednostavan način. Cilj ORM alata je kreiranje tablice čiji će stupci predstavljati atribute objekta kojeg korisnik želi preslikati. Postoje dvije vrste ORM obrazaca [8], to su *Active Record* i objekt za pristup podacima(engl. *Data Access Object*, DAO). U *Active Record* obrascu svaka je tablica predstavljena kao klasa, dok su redovi prevedeni u objekt odgovarajuće klase. Objekti znaju kako očuvati svoje stanje u bazi. Kod DAO obrasca, pristup bazi je povjeren objektima za pristup podacima. Oni znaju kako očuvati svaki objekt i kako kreirati objekte iz baze. Na slici 3.1 prikazan je način na koji funkcionira objektno-relacijsko preslikavanje. Prednosti ORM-a:

- Poboljšana produktivnost
- Manje Java koda
- Malo SQL-a
- Poboljšane performanse
- Jednostavan rad



Slika 3.1 Objektno-relacijsko preslikavanje

Za rad s ORM alatima potrebno je poznavati anotacije [7]. Anotacije su dijelovi koda koji započinju oznakom @ i daju određenu informaciju prevoditelju. Poboljšavaju čitljivost i razumljivost koda. U većini slučajeva anotacije nisu "ugnježđene" nego su specificirane kao djeca. One se pišu ispred klase, varijable ili metode na koju se odnose. Postoje dvije vrste anotacija, a to su logičke i fizičke anotacije. Logičke anotacije opisuju model entiteta, a fizičke opisuju ograničenja, stupce i tablice.
Najčešće anotacije koje se koriste su:

- @Entity- označava da je klasa entitet
- @Database- govori da je klasa baza podataka
- @Dao- određuje da je klasa DAO
- @PrimaryKey- označava da je atribut primarni ključ
- @Insert- govori da metoda služi za unos podataka u tablicu
- @Update- specificira da metoda služi za izmjenjivanje postojećih podataka u tablici
- @Delete- metoda briše podatke iz tablice
- @Query- nakon ove anotacije u zagradama se navodi upit kojeg će metoda izvršiti
- @NonNull- specificira da atribut ne može imati *null* vrijednost

U objektno-relacijskom preslikavanju, preslikanom stanju objekta se mora moći pristupiti dok je aplikacija pokrenuta, tada je moguće dohvatiti stanje, promijeniti ga i nazad spremiti u bazu. Postoje dvije vrste pristupanja stanju objekta:

- Atributni pristup
- Svojstveni pristup

Atributni pristup

Anotiranje atributa objekta omogućit će korisniku da koristi atributni pristup [7] njegovom stanju. Metode za dohvaćanje (engl. *getter*) i metode za postavljanje (engl. *setter*) ne moraju biti definirane, ali ako već jesu, to znači da mogu biti ignorirane. Svi atributi moraju biti deklarirani sa zaštićenim (engl. *protected*), paketnim (engl. *package*) ili privatnim (engl. *private*) načinom pristupa. Javni (engl. *public*) pristup je zabranjen zbog toga što će omogućiti ostalim klasama pristup atributima. Kako bi druge klase pristupile očuvanom stanju entiteta one koriste metode tog entiteta. U izlistanju koda 2.3 prikazan je entitet Zaposlenik koji je preslikan korištenjem atributnog pristupa. @Id anotacija označava da je id atribut primarni ključ i da se koristi atributni pristup. Ime i plaća atributi su tada očuvani i preslikani u stupce s istim imenom.

```
@Entity
public class Zaposlenik {
        @Id private int id;
        private String ime;
        private long plaća;
        public int getId() { return id; }
        public void setId(int id) { this.id = id; }
        public String getIme() { return ime; }
        public void setIme(String ime) { this.ime = ime; }
        public long getPlaća() { return plaća; }
        public void setPlaća(long plaća) { this. plaća = plaća; }
}
```

Izlistanje koda 2.3 primjer za atributni pristup

Svojstveni pristup

Kada se koristi ovaj pristup [7], tada moraju postojati metode za postavljanje i dohvaćanje. Tip svojstava je određen povratnim tipom metode za dohvaćanje i mora biti isti tip kao parametar poslan u metodi za postavljanje. Obje vrste metoda moraju imati javni ili zaštićeni način pristupa. U izlistanju koda 2.4 Zaposlenik klasa ima @Id anotaciju ispred metode getId() što znači da će korisnik koristiti svojstveni pristup kako bi pristupio stanju entiteta. Svojstva ime i plaća bit će očuvana zahvaljujući metodama za dohvaćanje i postavljanje, te dvije navedene varijable bit će preslikane u IME i PLAĆA stupce. Valja primijetiti da je plaća varijabla vraćena kao varijabla zarada što znači da nemaju isto ime. Ovo je korisniku od male važnosti zato što se koristi svojstveni pristup, a to znači da se ignoriraju varijable entiteta i koriste metode za postavljanje i dohvaćanje za preslikavanje i imenovanje.

```
@Entity
public class Zaposlenik {
        private int id;
        private String ime;
        private long zarada;
        @Id public int getId() { return id; }
        public void setId(int id) { this.id = id; }
        public String getIme() { return ime; }
        public void setIme(String ime) { this.ime = ime; }
        public long getPlaća() { return zarada; }
        public void setPlaća(long plaća) { this. zarada = plaća; }
}
```

Izlistanje koda 2.4 primjer za svojstveni pristup

Preslikavanje u tablicu

Postavljanje preslikavanja [7] je jednostavno jer su potrebne samo dvije anotacije, *@Entity* i *@Id.* Kod ovakvog preslikavanja, tablica će imati isto ime kao i entitet. Ako korisnik želi sam definirati ime tablice, to radi pomoću anotacije *@Table*(*name* = "ime_tablice"). Isto tako, ova anotacija omogućuje mijenjanje imena sheme baze podataka. Tipovi varijabli koje je moguće preslikati su:

- Primitivni Java tipovi- *byte, int, short, long, boolean, char, float, double*
- *Wrapper*klase primitivnih tipova - *Byte, Integer, Short, Long, Boolean, Character, Float, Double*
- Polja tipa *byte i char - byte[], Byte[], char[], Character[]*
- Veliki numerički tipovi - *java.math.BigInteger, java.math.BigDecimal*
- *Strinogvi: java.lang.String*
- Java vremenski tipovi - *java.util.Date, java.util.Calendar*
- JDBC (engl. *Java Databaseconnectivity*) vremenski tipovi - *java.sql.Date, java.sql.Time, java.sql.Timestamp*
- Numerički tipovi
- Serijalizirani objekti

Veze

Složene baze podataka imaju više entiteta koji su međusobno povezani. Kod modeliranja entitet je prikazan kao pravokutnik, a veza [7] kao strelica. Svaki entitet ima svoju ulogu u vezi. Najvažnije vrste veza su jednosmjerna i dvosmjerna veza koje su prikazane na Slikama 3.4 i 3.5. U dvosmjernoj vezi zaposlenik zna sve informacije o projektu i projekt zna sve o zaposleniku.

Slika 3.4 Primjer dvosmjerne veze [7]

Slika 3.5 Primjer jednosmjerne veze [7]

U jednosmjernoj vezi, entitet u kojeg pokazuje strelica ne zna ništa o entitetu u kojem se nalazi početak strelice. Kardinalnost je svojstvo koje govori koliko entiteta sudjeluje u vezi. Označava se brojkama ili znakom zvjezdice (*). Brojka određuje točan broj entiteta, a zvjezdica beskonačnost. Kardinalnost se označava na početku i na kraju strelice. Također, kardinalnost se može prikazati pomoću raspona (npr. 1...5 označava da postoji 1 do 5 entiteta). Postoji više vrsta preslikavanja:

1. Više prema jedan- entitet koji se nalazi na strani "više" strelice je izvor veze, a entitet koji se nalazi na strani "jedan" je cilj. To znači da će entitet koji je izvor sadržavati atribut koji je referenca na ciljani entitet. Ovakva vrsta preslikavanja je definirana anotiranjem atributa u izvornom entitetu s @ManyToOne anotacijom.
2. Jedan prema jedan- Definira se anotiranjem atributa s @OneToOne
3. Jedan prema više- Ovo preslikavanje je slično 1. vrsti preslikavanja samo što je u ovom slučaju preslikavanje u obrnutom smjeru.
4. Više prema više- oba entiteta sadrže kolekcijske atribute anotirane s @ManyToMany

3.1. ORM alati na Android platformi

U ovom će potpoglavlju biti detaljno opisana četiri ORM alata: *Room, Realm, DBFlow* i *Active Android.* Bit će navedene njihove najvažnije osobine.

3.1.1. *Room*

Room [9] biblioteka pruža apstraktni sloj preko SQLite-a i na taj način omogućuje jednostavniji rad s SQLite bazom podataka. Napravio ju je Google i predstavio 2017. godine. Za razliku od većine ostalih ORM alata, Room koristi anotacijsko procesiranje za očuvanje podataka. To znači da klase modeli ne moraju ništa naslijediti vezano uz *Room*. Povezivanje dviju tablica se ne odvija

automatski tj. veza se ne može uspostaviti dodavanjem instance jedne klase u definiciju druge klase, potrebno je koristiti @ForeignKey anotacije. Rad s *Roomom* omogućuju 3 glavne komponente:

- Entitet - predstavlja tablicu u bazi podataka. Sastoji se od Java klase koja je anotirana s *@Entity*. Unutar *Entity* anotacije mogu se definirati primarni i strani ključevi i ime entiteta.
- DAO - Omogućuje pristupanje tablici i manipuliranje podacima. Svaka tablica mora imati svoj DAO. Kreira se anotiranjem Java sučelja s @Dao.
- Baza podataka - klasa koja predstavlja bazu podataka. Ta se klasa anotira s *@Database* i ispunjava sljedeće uvjete: 1. Mora biti apstraktna i nasljeđivati klasu *RoomDatabase*. 2. Mora sadržavati listu entiteta. 3. Mora sadržavati apstraktnu metodu koja nema argumenata i koja vraća klasu koja je anotirana s @Dao. Preporučeno je da se sav rad s podacima baze vrši na pozadinskoj niti kako ne bi došlo do usporavanja aplikacije.

Zašto Room?

- brzina čitanja podataka
- manje koda
- sve metode za manipulaciju podacima su anotirane
- sigurnost
- jednostavnost

3.1.2 *Realm*

Realm [10] baza podataka je višeplatformna baza podataka koja se pojavila kao alternativa SQLiteu. Pogonjena je direktno unutar pametnih telefona, tableta i pametnih satova. U usporedbi s SQLiteom, *Realm* je lakši za podešavanje i za to je potrebno puno manje koda. *Realm* je brži i podržava moderne mogućnosti kao što su enkripcija i JSON podrška. Za razliku od drugih baza podataka, objekti u *Realmu* su nativni objekti što znači da ih se ne mora kopirati van iz baze, vršiti izmjene na njima i pohraniti ih natrag nego se uvijek radi sa stvarnim objektima. Ako aplikacija treba pohraniti podatke na oblak i mora ih imati sinkronizirane na svim uređajima korisnika, tada se koristi *RealmObject Server* zajedno s *Realm* bazom. *Realm* je NoSQL baza što znači da ne koristi SQL sintaksu i ne sprema podatke u SQL bazu. *Realm* ima mogućnost jednostavnog kreiranja konfiguracije baze podataka, a to se omogućuje pomoću *RealmConfiguration.Builder* klase. Sva čitanja ili pisanja iz ili u bazu obavljaju se unutar transakcija. Zašto *Realm*?

- malo koda

- sigurnost
- odlična dokumentacija
- podrška za više platformi
- nema potrebe za pisanjem SQL koda
- jako velika brzina

3.1.3 *DBFlow*

DBFlow [11] je biblioteka koja pojednostavljuje rad s SQLite bazom i generira kod tijekom prevođenja što ga čini jednim od najbržih ORM alata. Ova je biblioteka izgrađena od kolekcije najboljih značajki drugih biblioteka na najefikasniji način. Koristi *Active Record* ORM obrazac što znači da se tablice kreiraju automatski. Zašto *DBFlow*?

- Brzina
- Podržava okidače, indekse, migracije i transakcije
- Koristi upite slične upitima u SQLiteu.
- Otvorenog je koda što znači da svi mogu vršiti izmjene i doprinositi projektu
- Mogućnost korištenja više baza podataka
- podrška za enkripciju podataka

3.1.4 *Active Android*

Active Android [12] je ORM alat koji omogućuje jednostavno upravljanje SQLite bazom bez direktnog pisanja SQL-a. Radi kao i svaki drugi ORM alat što znači da preslikava Java klase u tablice baze podataka i atribute u stupce. Svaki red u tablici predstavlja određeni objekt. Sve ovo omogućava kreiranje, izmjenjivanje, brisanje i vršenje upita nad bazom koristeći objekte umjesto korištenja SQL-a. Prvi koristi *Active Record* ORM obrazac. Nema stvarni graditelj upita.
Prednosti *Active Androida*:

- jednostavan kod
- čitljiv kod
- podrška za migracije i *joine*

3.2 Usporedba performansi ORM alata

Kod usporedbe performansi ORM alata mjere se određeni parametri kao što su:

- Brzina kreiranja baze podataka
- Brzina zapisivanja podataka u bazu
- Brzina čitanja iz baze podataka
- Brzina čitanja indeksiranih atributa
- Brzina pretraživanja baze podataka s uvjetima WHERE ili LIKE
- Brzina brisanja objekata iz baze

Za mjerenje navedenih parametara potrebno je imati bazu s većim brojem zapisa. U ovom će se radu obraditi samo neki od njih. Mjerit će se vrijeme unosa podataka u bazu, vrijeme čitanja podataka iz baze i vrijeme brisanja zapisa iz baze. Vrijeme će se mjeriti u milisekundama.

4. PROGRAMSKO RJEŠENJE ZA USPOREDBU PERFORMANSI ORM ALATA

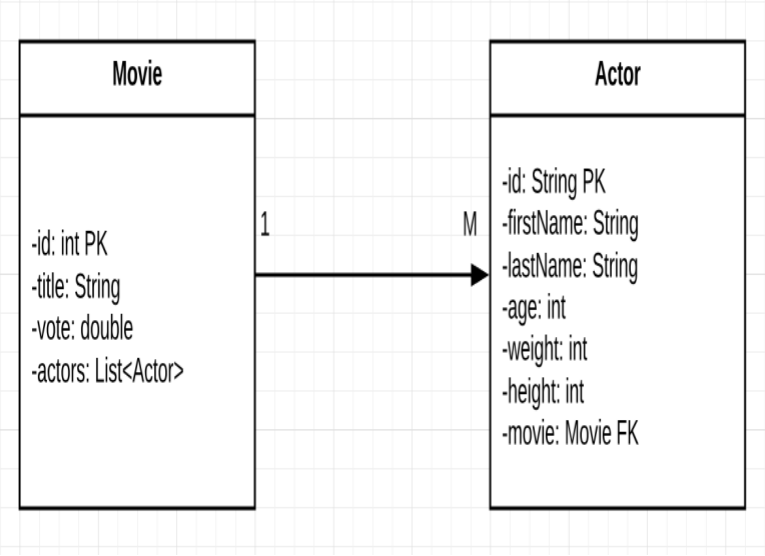
Aplikacija će imati jednostavno korisničko sučelje koje će omogućiti korisniku izvođenje jednostavnih operacija nad bazom podataka prikazanoj na Slici 4.1. Obrađivat će se podaci o filmovima i glumcima. Cilj aplikacije je mjerenje vremena izvršenja operacija za svaki ORM alat. Nakon mjerenja bit će vidljivo koji je ORM alat najbrži.

4.1 Zahtjevi na sustav

Tablica 4.1 prikazuje zahtjeve na sustav gdje je opisano što aplikacija radi.

Tablica 4.1 Opis zahtjeva na sustav

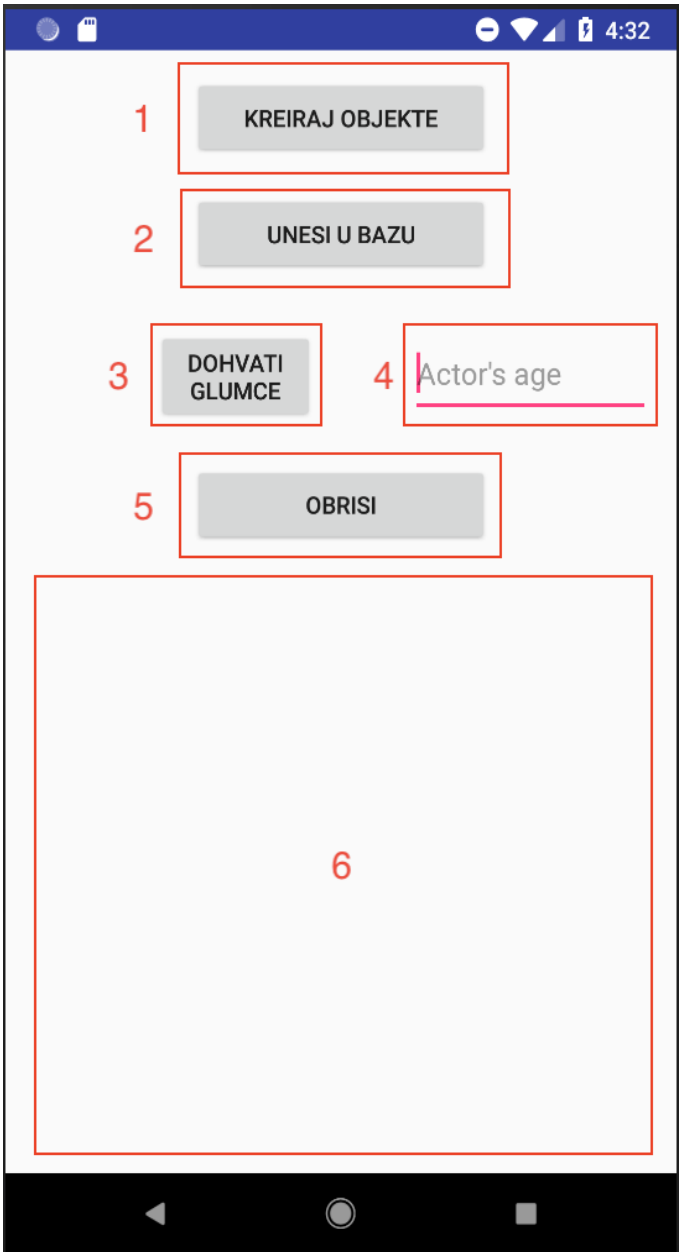
ID	OPIS
1	Korisnik ulazi u aplikaciju dodirom na ikonu
2	Korisniku je prikazano korisničko sučelje
3	Korisnik dodirom na prvi gumb kreira objekte i rezultat je prikazan na dnu ekrana
4	Korisnik dodirom na drugi gumb sprema kreirane objekte u bazu i prikazuje se rezultat
5	Korisnik unosi godine
6	Dodirom na treći gumb ispisuje se broj glumaca koji su stariji od unesenog iznosa
7	Dodirom za četvrti gumb brišu se svi podaci iz baze podataka



Slika 4.1 Shema baze podataka nacrtana u *Lucidchartu*

4.2 Način rada sustava

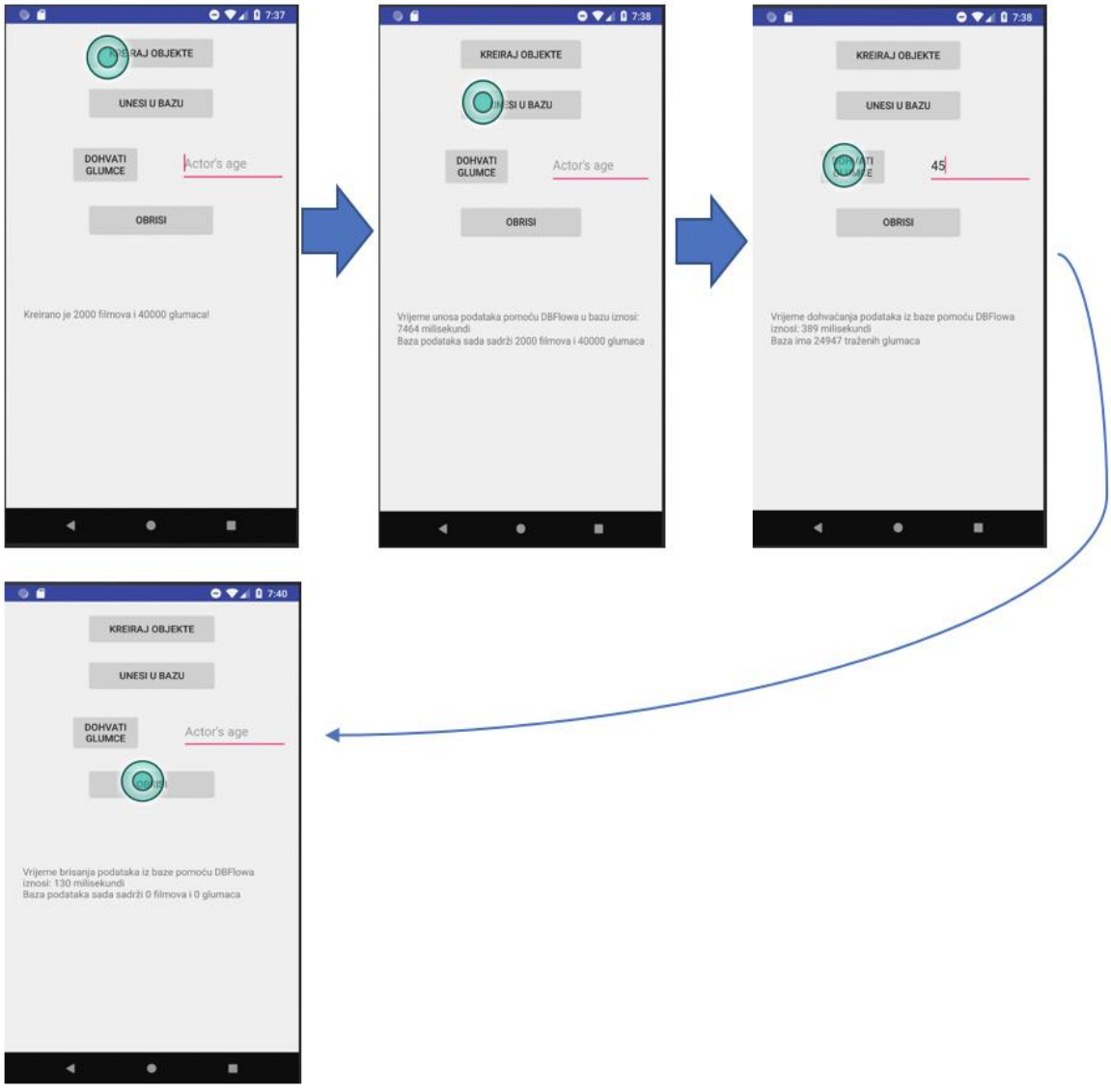
Aplikacija će se sastojati od korisničkog sučelja koje je prikazano na Slici 4.2 i imat će nekoliko gumbova čiji će zadatak biti obavljanje određene operacije nad bazom podataka. Gumbovi će obavljati kreiranje objekata, unošenje tih objekata u bazu, dohvaćanje objekata pod određenim uvjetom i brisanje objekata iz baze. Na dnu ekrana nalazit će se prostor gdje će biti prikazana poruka. Ta poruka će korisniku reći je li određeni zadatak uspješno izvršen. Način rada sustava je prikazan na Slici 4.3.



Slika 4.2 Prikaz sučelja

1. Gumb za generiranje objekata
2. Gumb za unos objekata u bazu podataka
3. Gumb koji dohvaća glumce koji su stariji od iznosa koji je unesen pod 4.
4. Prostor za unošenje godina glumca
5. Gumb za brisanje zapisa iz baze podataka
6. Prostor u kojem pišu informacije o bazi nakon klika na gumb

Prikaz rada sustava



Slika 4.3 Prikaz rada sustava

4.3 Usporedba ORM alata

Osim u performansama, ORM alati se razlikuju u konfiguraciji, načinu izvođenja upita i izgledima entiteta i baza podataka. Ovo poglavlje će dati detaljan uvid u navedene razlike.

4.3.1 Konfiguracija

Ovdje je prikazano kako se pojedini ORM alat dodaje u Android Studio projekt. Dodavanje se svodi na upisivanje linija koda unutar *Gradle* datoteke.

Room

U izlistanju koda 4.1 prikazan je način konfiguriranja Room alata.

```
def room_version = "1.1.1"
implementation "android.arch.persistence.room:runtime:$room_version"
annotationProcessor "android.arch.persistence.room:compiler:$room_version"
```

Izlistanje koda 4.1 Dodavanje *Rooma* u projekt

Realm

Realm se konfigurira dodavanjem *applyplugin : 'realm-android'* linije koda unutar *GradleModule: app* datoteke. Također, potrebno je dodati *RealmGradle plugin* u projektnoj *Gradle* datoteci.

DBFlow

Izlistanje koda 4.2 prikazuje kako se dodaje *DBFlow* alat u projekt.

```
def dbflow_version = "4.0.2"
annotationProcessor "com.github.Raizlabs.DBFlow:dbflow-processor:${dbflow_version}"
implementation "com.github.Raizlabs.DBFlow:dbflow-core:${dbflow_version}"
implementation "com.github.Raizlabs.DBFlow:dbflow:${dbflow_version}"
```

Izlistanje koda 4.2 Dodavanje *DBFlowa* u projekt

Active Android

Način dodavanja *Active Android* alata prikazan je u izlistanju koda 4.3.

```
implementation 'com.michaelpardo:activeandroid:3.1.0-SNAPSHOT'
```

Izlistanje koda 4.3 Prikaz dodavanja *Active Android* alata u projekt

4.3.2 Entiteti

Ovdje će biti prikazano da svaki alat ima svoj način definiranja entiteta. Neki od njih koriste anotacije, a neki nasljeđivanje drugih klasa. Također, bit će jasno vidljivo da način definiranja stupaca ovisi od alata do alata. Definicije konstruktora i metoda za postavljanje i čitanje bit će izostavljane radi uštede prostora.

Room

U izlistanju koda 4.4 je bitno primijetiti ulogu anotacija. *@Entity* anotacija označava da je klasa entitet. Unutar anotacije moguće je definirati ime tablice, ako ime nije definirano onda je jednako imenu klase. *@PrimaryKey* anotacija definira da je atribut primarni ključ tablice. @ForeignKey anotacija određuje strani ključ i unutar nje je moguće definirati više njih.

```
@Entity
public class Movie {
   @PrimaryKey
   private int id;
   private String title;
   private int vote;

}
```

```
@Entity(foreignKeys =
@ForeignKey(entity = Movie.class,parentColumns = "id",
childColumns = "movieId")
public class Actor {
   @PrimaryKey
   @NonNull
   private String id;
   private int movieId;
   private String firstName;
   private String lastName;
   private int age;
   private int weight;
   private int height;
}
```

Izlistanje koda 4.4 Prikaz entiteta kod *Room* alata

Realm

Kod *Realm* ORM alata važno je napomenuti da se ne koristi *@Entity* anotacija kako bi se odredio entitet. Umjesto toga, entitet se definira nasljeđivanjem klase *RealmObject*. Primjer entiteta je vidljiv u izlistanju koda 4.5.

```
public class Movie extends RealmObject {

   @PrimaryKey
   private int id;
   private String title;
   private int vote;
   private RealmList<Actor> actors = new RealmList<>();

}
```

```
public class Actor extends RealmObject {
   @PrimaryKey
   private String id;
   private String firstName;
   private String lastName;
   private int age;
   private int weight;
   private int height;
}
```

Izlistanje koda 4.5 Prikaz entiteta kod *Realm* alata

DBFlow

Primjer je vidljiv u izlistanju koda 4.6. Definicija tablice je postignuta *@Table* anotacijom kojoj se predaju klasa baze podataka i ime tablice. Klasa koja predstavlja entitet nasljeđuje *BaseModel* klasu koja joj omogućuje korištenje metoda poput *save().* Svaki je atribut potrebno anotirati s *@Column*, ako to nije učinjeno tada taj atribut neće biti spremljen u obliku stupca baze podataka.

```
@Table(database = DBFlowDatabase.class, name
= "movieTable")
public class Movie extends BaseModel {
   @PrimaryKey
   @Column
   int id;
   @Column
   String title;
   @Column
   int vote;
   List<Actor> actors;
}
```

```
@Table(database = DBFlowDatabase.class, name =
"actorTable")
public class Actor extends BaseModel {
   @PrimaryKey
   @Column
   String id;
   @Column
   String firstName;
   @Column
   String lastName;
   @Column
   int age;
   @Column
   int weight;
   @Column
   int height;
   @ForeignKey(stubbedRelationship = true)
   Movie movie;
}
```

Izlistanje koda 4.6 Primjer entiteta definiranih u *DBFlow* alatu

Active Android

Klasa koja predstavlja entitet nasljeđuje *Model* klasu što je vidljivo u izlistanju koda 4.7. Svi atributi su anotirani s *@Column* kao kod *DBFlow-a*. Zanimljivost je što za rad nisu potrebne metode za postavljanje i dohvaćanje jer svi atributi imaju javni pristup. *Active Android* sam generira primarne ključeve i na taj način osigurava da je svaki entitet jedinstven.

```
@Table(name = "movie")
public class Movie extends Model {
    @Column(name = "Title")
    public String title;
    @Column(name = "Vote")
     public int vote;
    @Column(name = "actor_id")
    public Actor actor;
}
```

```
@Table(name = "actor")
public class Actor extends Model{
   @Column(name = "first_name")
   public String firstName;
   @Column(name = "last_name")
   public String lastName;
   @Column(name = "age")
   public int age;
   @Column(name = "weight")
   public int weight;
   @Column(name = "height")
   public int height;
}
```

Izlistanje koda 4.7. Prikaz entiteta definiranih u *Active Android* alatu

4.3.3 Definiranje baze podataka

U ovom poglavlju bit će prikazane definicije baze podataka za svaki alat.

Room

Unutar *@Database* anotacije nalaze se entiteti koje baza sadrži i verzija baze. Ovo je prikazano u izlistanju koda 4.8. Ako dođe do promjene atributa entiteta tada je potrebno povećati verziju baze. *Room* baza podataka se definira nasljeđivanjem *RoomDatabase* klase. Ona mora biti apstraktna i sadržavati apstraktne metode koje vraćaju DAO.

```
@Database(entities = {Movie.class, Actor.class}, version = 2)
public abstract class RoomDatabase extends android.arch.persistence.room.RoomDatabase
{
    public abstract MovieDAO getMovieDao();
    public abstract ActorDAO getActorDao();
}
```

Izlistanje koda 4.8. Prikaz definicije baze podataka

U izlistanjima koda 4.9 i 4.10 prikazani su DAO-i. To su sučelja koja sadrže sve metode za manipuliranje podacima. Važno je za napomenuti da postoje anotacije koje omogućuju jednostavne operacije poput unosa, brisanja i izmjene podataka, a to znači da nije potrebno pisati SQL kod. Moguće je kreirati vlastite upite pomoću anotacije *@Query* koja nudi veću slobodu programerima.

```
@Dao
public interface MovieDAO {
   @Insert
   void insert(Movie... movies); // unesi više filmova
   @Insert
   void insertAll(List<Movie> movies); // unesi listu filmova
   @Insert
   void insertSingleMovie(Movie movie); // unesi samo jedan film
   @Update
   void update(Movie... movies); // izmijeni vise filmova
   @Delete
   void delete(Movie... movies); // obrisi vise filmova
   @Query("SELECT * FROM Movie") //dohvati sve filmove iz baze
   List<Movie>getAllMovies();
   @Query("DELETE FROM Movie") // obrisi sve filmove iz baze
   void deleteTable();
}
```

Izlistanje koda 4.9 Primjer DAO-a

```
@Dao
public interface ActorDAO {
@Insert
void insert(Actor actor); //unesi jednog glumca

@Insert
void insertAll(List<Actor> actors); //unesi listu glumaca

@Update
void update(Actor... actors); //izmijeni vise glumaca

@Delete
void delete(Actor... actors); //obrisi vise glumaca

@Query("SELECT * FROM Actor") //dohvati sve glumce izbaze
List<Actor>getAllActors();

@Query("SELECT * FROM Actor WHERE movieId = :movieId") //dohvati sve glumce
List<Actor>findActorsByMovieId(int movieId);                    //odredjenog filma

@Query("SELECT firstName FROM Actor WHERE age > :age") //dohvati imena glumaca koji
List<String>getActorsByAge(int age);                            //su stariji od "age" vrijednosti

@Query("DELETE FROM Actor") // obrisi sve glumce
void deleteTable();
}
```

Izlistanje koda 4.10 Primjer DAO-a

Realm

Realm baza podataka se inicijalizira pozivom statičke metode *init()* klase *Realm* i predaje joj se kontekst kao argument. Zatim se referenci tipa *Realm* pridružuje instanca pomoću statičke metode *getDefaultInstance()* klase *Realm*. Nakon što je navedeno učinjeno, baza je spremna za rad.

DBFlow

DBFlow se inicijalizira pomoću statičke metode *init() FlowManager* klase. Ta metoda prima objekt *FlowConfig.Builder()*. Tek nakon toga, rad s ovim ORM alatom je moguć. U usporedbi s *Room* ORM alatom, valja napomenuti da *DBFlow* ne koristi DAO za manipulaciju podacima. Definicija baze podataka je vidljiva u izlistanju koda 4.11.

```
@Database(name = DBFlowDatabase.NAME, version= DBFlowDatabase.VERSION)
public class DBFlowDatabase {
   public static final String NAME = "MyDatabase";
   public static final int VERSION = 1;
}
```

Izlistanje koda 4.11 Prikaz definicije baze podataka u *DBFlow* alatu

Active Android

Kod *Active Androida* meta podaci o bazi se definiraju unutar *manifesta*. Na taj je način moguće odrediti ime baze, verziju, modele i još mnogo stvari. U izlistanju koda 4.12 definirani su ime i verzija baze podataka.

```
<meta-data android:name="AA_DB_NAME" android:value="mydatabase.db"/>

<meta-data android:name="AA_DB_VERSION" android:value="1"/>
```

Izlistanje koda 4.12 Prikaz definiranja baze podataka unutar manifest datoteke

4.3.4 Jednostavne operacije nad bazom

Ovdje će se pružiti prikaz izvođenja nekih jednostavnih operacija. Lako će se moći uočiti razlika između korištenja klasičnog SQLitea i korištenja ORM alata. Bit će vidljivo da svaki alat ima svoj način izvršavanja upita, neki su složeniji, a neki jednostavniji. Upiti će omogućiti unos podataka u bazu, brisanje svih podataka iz baze i čitanje iz baze pod različitim uvjetima. Svaki će upit biti ukratko opisan u komentarima.

Room

Za unos podataka prvo se kreira objekt, zatim mu se postavljaju vrijednosti atributa, nakon toga objekt baze podataka dohvaća DAO objekt koji sadrži metodu za unos objekta koji je u ovom slučaju *Movie*. Za brisanje entiteta i manipulaciju podacima koriste se metode DAO objekta. Sve navedeno prikazano je u izlistanju koda 4.13.

```
//unos filma u bazu
Moviemovie = newMovie( );
movie.setId(555);
movie.setTitle("Kum");
movie.setVote(10);
myRoomDatabase.getMovieDao.insert(movie);

//brisanje svih zapisa iz Movie tablice
myRoomDatabase.getMovieDao().deleteTable();

//dohvaćanje imena glumaca koji su stariji od 30 godina
List<String> imena = myRoomDatabase.getActorDao.getActorsByAge(30);

//dohvaćanje svih glumaca koji glume u filmu čiji je id 22
List<Actor>actors = myRoomDatabase.getActorDao().findActorsByMovieId(22);
```

Izlistanje koda 4.13 Prikaz operacija nad objektima

Realm

Kod *Realma* se prvo kreira objekt pomoću metode *createObject()* koja prima dva parametra - klasa objekta koji želimo stvoriti i primarni ključ. Nakon toga se vrijednosti postavljaju pomoću metoda za postavljanje. Za unos veće količine podataka preporučuje se korištenje transakcija kako bi se na taj način ubrzalo izvršavanje koda. Svi objekti iz baze dohvaćaju se pomoću metoda *Realm* objekta i spremaju se kao *RealmResults* objekti. Ovdje se može vidjeti da *Realm* baza ne koristi uopće SQL što omogućuje minimalno pisanje koda i bolju čitljivost. Sve navedeno se može vidjeti u izlistanju koda 4.14.

```
//unos filma u bazu
Movie movie = realm.createObject(Movie.class,33);
movie.setTitle("Kum");
movie.setVote(10);
//brisanje svih zapisa iz Movie tablice
RealmResults<Movie>movieResult = realm.where(Movie.class).findAll();
movieResult.deleteAllFromRealm();
//dohvaćanje glumaca koji su stariji od 39 godina
RealmResults<Actor>actors = realm.where(Actor.class).greaterThan("age", 39).findAll();
//dohvaćanje svih glumaca koji su visoki između 150 i 180 centimetara
RealmResults<Actor>actors = realm.where(Actor.class).between("height",150,180).findAll();
```

Izlistanje koda 4.14 Prikaz operacija nad objektima

DBFlow

Unos podataka kod *DBFlowa* je jako sličan *Roomovom* unosu što se može vidjeti usporedbom izlistanja koda 4.14 i 4.15. Jedina je razlika što se nakon postavljanja vrijednosti koristi *save()* metoda za spremanje objekta u bazu. Brisanje podataka se vrši pomoću statičke metode *tables()*. Za čitanje i pisanje podataka su zaslužne statičke metode SQLite klase.

```
//unos filma u bazu
Movie movie = newMovie();
movie.setId(666);
movie.setTitle("Kum");
movie.setVote(10);
movie.save();
//brisanje svih zapisa iz Movie tablice
Delete.tables(Movie.class);
//dohvaćanje glumaca koji su stariji od 77 godina
List<Actor>actors = SQLite.select().from(Actor.class).where(Actor_Table.age.greaterThan(77)).queryList();
//dohvaćanje svih glumaca koji se zovu Clint i imaju više od 60kg
List<Actor>actors = SQLite.select().from(Actor.class)
   .where(Actor_Table.firstName.is("Clint"))
   .and().where(Actor_Table.weight.greaterThan(60))
   .queryList();
```

Izlistanje koda 4.15. Prikaz operacija nad objektima

Active Android

Unos objekata je potpuno isti kao kod *DBFlowa* što je vidljivo u usporedbi izlistanja koda 4.16 i 4.15. Brisanje podataka je isto jako slično, taj se proces svodi na kreiranje objekata *Delete()*. Dohvaćanje podataka vrši se kreiranjem objekta *Select()* na koji se dodaju lančane metode koje predstavljaju pojedine dijelove SQL koda.

```
//unos filma u bazu
Movie movie = new Movie();
movie.title = "Inception";
movie.vote = 9;
movie.save();
//brisanje svih zapisa iz Movie tablice
new Delete().from(Movie.class).execute();
//dohvaćanje glumaca nižih od 170cm
List<Actor>actors = newSelect().from(Actor.class)
   .where("height< ?", 170).execute();
//dohvaćanje filmova koji imaju ocjenu veću od 7 i sortiranje prema imenu
List<Movie>movies = newSelect().from(Movie.class)
   .where("vote> ?", 7).orderBy("title").execute();
```

Izlistanje koda 4.16 Prikaz operacija nad objektima

4.4 Testiranje rješenja

U ovom će se poglavlju testirati aplikacija za 3 različite situacije koje će ovisiti o broju objekata spremljenih u bazi. Nakon toga će se provoditi upiti nad bazom. Vršit će se upiti za unos filmova i glumaca, dohvaćanje glumaca pod određenim uvjetom i upiti za brisanje podataka iz baze. Svaki upit će se izvesti 10 puta. Mjerit će se vrijeme izvođenja pojedinog upita i zapisivat u tablicu. Nakon toga, računat će se statistički podaci kao što su srednja vrijednost i standardna devijacija. Standardna devijacija predstavlja mjeru raspršenosti podataka u skupu. Testiranje će se obaviti na *Nexus 5X* uređaju. Situacije:

- 500 objekata filmova i 7500 objekata glumaca
- 1000 objekata filmova i 20000 objekata glumaca
- 2000 objekata filmova i 40000 objekata gluma

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2} \quad (4\text{-}1)$$

gdje je:

- σ– standardna devijacija
- N– broj ponavljanja
- x_i– pojedini upit
- $\bar{x}$– srednja vrijednost

$$\bar{x} = \frac{x_1 + \cdots + x_n}{n} \qquad (4\text{-}2)$$

gdje je :

- $\bar{x}$– srednja vrijednost
- x_n– upiti
- n– broj ponavljanja

Prva situacija (500 objekata filmova i 7500 objekata glumaca)

Polazeći od toga da je korisničko sučelje već kreirano, u programskom se kodu vrši sve potrebno za konfiguraciju ORM alata korištenih u radu. Kreiraju se potrebni modeli, klase koje opisuju bazu i objekti za pristup bazi. Zatim se programiraju gumbi za izvođenje upita nad bazom. Klikom na prvi gumb kreira se 500 objekata filmova i 7500 objekata glumaca. Jednim klikom na drugi gumb počinje izvođenje prvog upita. Izvođenje je izvedeno relativno brzo zbog manjeg broja objekata u bazi. Nakon završetka izvođenja, rezultat je vidljiv na dnu ekrana. Tada se rezultat zapisuje u tablice 4.2, 4.3, 4.4 i 4.5. Nakon što se napravi 10 upita, prema izrazima (4-1) i (4-2) računaju srednja vrijednost i standardna devijacija. Postupak se ponavlja za 2. i 3. vrstu upita. Rezultati će biti grafički prikazani.

Tablica 4.2 prikazuje vršenje upita pomoću *Room* alata u prvoj situaciji gdje je u bazi 500 objekata filmova i 7500 objekata glumaca.

Tablica 4.2 Mjerenja prve situacije kod *Rooma*

ROOM												
Upit\Xᵢ	**X_1[ms]**	**X_2[ms]**	**X_3[ms]**	**X_4[ms]**	**X_5[ms]**	**X_6[ms]**	**X_7[ms]**	**X_8[ms]**	**X_9[ms]**	**X_{10}[ms]**	**$\bar{x}$ [ms]**	**σ**
Insert	732	568	541	570	551	567	566	582	536	566	577.9	53.07
Get	13	6	7	10	10	14	15	12	12	13	11.2	2.78
Delete	260	240	238	234	234	235	232	237	239	235	238.4	7.58

Tablica 4.3 prikazuje vršenje upita pomoću *Realm* alata u prvoj situaciji i pri tome se u bazi nalazi 500 objekata filmova i 7500 objekata glumaca. Vidljivo je *Realm* alat puno brže (vremena izvršavanja su manja) vrši upite u usporedbi s *Room* alatom.

Tablica 4.3 Mjerenja prve situacije kod *Realma*

REALM												
Upit\Xi	**X1[ms]**	**X2[ms]**	**X3[ms]**	**X4[ms]**	**X5[ms]**	**X6[ms]**	**X7[ms]**	**X8[ms]**	**X9[ms]**	**X10[ms]**	**x̄ [ms]**	**σ**
Insert	359	325	294	271	251	265	262	268	262	260	281.7	32.82
Get	2	1	1	1	2	1	3	1	1	2	1.5	0.67
Delete	214	199	138	107	346	151	106	327	153	326	206.7	82.38

Tablica 4.4 prikazuje vršenje upita pomoću *DBFlow* alata u prvoj situaciji. Vidljivo je da su vremena izvršavanja puno veća u usporedbi s alatima *Room* i *Realm.*

Tablica 4.4 Mjerenja prve situacije kod *DBFlowa*

DBFLOW												
Upit\Xi	**X1[ms]**	**X2[ms]**	**X3[ms]**	**X4[ms]**	**X5[ms]**	**X6[ms]**	**X7[ms]**	**X8[ms]**	**X9[ms]**	**X10[ms]**	**x̄ [ms]**	**σ**
Insert	1387	1232	1230	1250	1243	1229	1228	1246	1227	1269	1254.1	46.04
Get	69	59	56	47	64	36	29	58	50	60	52.8	11.85
Delete	32	33	33	32	35	32	33	28	39	34	33.1	2.62

Tablica 4.5 prikazuje vršenje upita pomoću *Active Android* alata. Može se primijetiti da je ovaj alat obavio posao najsporije.

Tablica 4.5 Mjerenja prve situacije kod *Active Androida*

ACTIVE ANDROID												
Upit\Xi	**X1[ms]**	**X2[ms]**	**X3[ms]**	**X4[ms]**	**X5[ms]**	**X6[ms]**	**X7[ms]**	**X8[ms]**	**X9[ms]**	**X10[ms]**	**x̄ [ms]**	**σ**
Insert	4831	4809	4633	4947	4855	4809	4106	5002	4903	4461	4735.6	256.93
Get	136	86	64	125	85	125	33	54	135	86	92.9	34.33
Delete	36	32	34	31	30	36	29	31	43	32	33.4	3.90

Druga situacija (1000 objekata filmova i 20000 objekata glumaca)

Ponavlja se postupak iz prve situacije osim što se u bazu unosi 1000 objekata filmova i 20000 objekata glumaca. Mjerenja se unose u tablice 4.6, 4.7, 4.8 i 4.9. U ovoj se situaciji povećava vrijeme izvođenja upita kod svih alata.

Uspoređujući srednje vrijednosti u Tablici 4.6 i 4.2 vidljivo je da se srednje vrijeme izvršavanja unosa povećalo za 681.1 milisekundi, a srednje vrijeme dohvaćanja za 10.1 milisekundi. Srednje vrijeme brisanja podataka naraslo je za 993.6 milisekundi.

Tablica 4.6 Mjerenja druge situacije kod *Rooma*

ROOM												
Upit\Xi	**X1[ms]**	**X2[ms]**	**X3[ms]**	**X4[ms]**	**X5[ms]**	**X6[ms]**	**X7[ms]**	**X8[ms]**	**X9[ms]**	**X10[ms]**	**x̄ [ms]**	**σ**
Insert	1324	1277	1218	1237	1249	1269	1252	1249	1256	1259	1259	26.59
Get	13	13	14	23	22	26	29	14	30	29	21.3	6.8
Delete	1240	1206	1209	1273	1249	1235	1240	1232	1215	1227	1232	19.03

U Tablici 4.7 vidljivo je da se srednje vrijeme izvršavanja unosa podataka povećalo za 600.9 milisekundi u usporedbi s Tablicom 4.3. Srednje vrijeme brisanja podataka povećalo se s 1.5 na 4.9 milisekundi. Prosječno vrijeme dohvaćanja podataka je naraslo za 421.2 milisekunde.

Tablica 4.7 Mjerenja druge situacije kod *Realma*

REALM												
Upit\Xi	**X1[ms]**	**X2[ms]**	**X3[ms]**	**X4[ms]**	**X5[ms]**	**X6[ms]**	**X7[ms]**	**X8[ms]**	**X9[ms]**	**X10[ms]**	**x̄ [ms]**	**σ**
Insert	907	929	912	872	886	882	881	871	866	920	892.6	21.32
Get	3	5	1	10	7	2	8	5	3	5	4.9	2.66
Delete	619	634	635	684	582	586	636	647	682	574	627.9	36.66

U Tablici 4.8 rezultati su se također promijenili kao i u prethodnim slučajevima. Srednje vrijeme unosa podataka se povećalo za 1961.5 milisekundi. Srednje vrijeme dohvaćanja se povećalo za 130.1 milisekundi. Vrijeme brisanja je u ovom slučaju veće za 37.7 milisekundi.

Tablica 4.8 Mjerenja druge situacije kod *DBFlowa*

DBFLOW												
Upit\Xi	**X1[ms]**	**X2[ms]**	**X3[ms]**	**X4[ms]**	**X5[ms]**	**X6[ms]**	**X7[ms]**	**X8[ms]**	**X9[ms]**	**X10[ms]**	**x̄ [ms]**	**σ**
Insert	3744	3141	3118	3217	3131	3170	3144	3158	3200	3133	3215.6	178.6
Get	224	204	231	245	81	187	103	131	221	102	182.9	50.39
Delete	78	69	86	64	68	67	72	69	66	68	70.8	6.24

Tablica 4.9 prikazuje rezultate druge situacije kod *Active Android* alata. Srednje vrijeme unosa se povećalo za 6421.7 milisekundi. Vrijeme dohvaćanja podataka je naraslo s 92.9 milisekundi na 304.2 milisekunde, a vrijeme brisanja s 33.4 na 49.3 milisekunde.

Tablica 4.9 Mjerenja druge situacije kod *Active Androida*

ACTIVE ANDROID												
Upit\Xi	**X1[ms]**	**X2[ms]**	**X3[ms]**	**X4[ms]**	**X5[ms]**	**X6[ms]**	**X7[ms]**	**X8[ms]**	**X9[ms]**	**X10[ms]**	**x̄ [ms]**	**σ**
Insert	11909	11832	13401	13226	12837	12505	10363	12324	1314	11862	11157.3	3381.6
Get	381	274	420	311	244	174	144	240	434	420	304.2	100.4
Delete	51	49	49	50	47	48	51	58	41	49	49.3	3.98

Treća situacija (2000 objekata filmova i 40000 objekata glumaca)

Ponavlja se postupak iz prve i druge situacije osim što se u bazu unosi 2000 objekata filmova i 40000 objekata glumaca. Mjerenja se unose u tablice 4.10, 4.11, 4.12 i 4.13. U ovoj se situaciji vrijeme izvođenja upita značajno povećava. Bitno je za primijetiti da *Active Androidu* treba više od 20 sekundi za unos podataka.

U Tablici 4.10 vidljivo je da je vrijeme unosa podataka naraslo za 882.7 milisekundi u usporedbi s drugom situacijom. Vrijeme dohvaćanja je naraslo za 29.7 milisekundi, a vrijeme brisanja za 5581.8 milisekundi.

Tablica 4.10 Mjerenja treće situacije kod *Rooma*

ROOM												
Upit\Xᵢ	**X₁[ms]**	**X₂[ms]**	**X₃[ms]**	**X₄[ms]**	**X₅[ms]**	**X₆[ms]**	**X₇[ms]**	**X₈[ms]**	**X₉[ms]**	**X₁₀[ms]**	**x̄ [ms]**	**σ**
Insert	2185	2130	2134	2149	2153	2150	2107	2155	2128	2126	2141.7	20.3
Get	26	27	39	52	57	21	74	26	106	82	51	27.2
Delete	6869	6757	6763	6826	6724	6845	6899	6757	6838	6860	6813.8	55.9

U Tablici 4.11, srednje vrijeme unosa podataka je naraslo za 521.2 milisekundi u usporedbi s Tablicom 4.7. Isto tako, srednje vrijeme dohvaćanja podataka se povećalo za 3.4 milisekundi, a srednje vrijeme brisanja za 1493.9 milisekundi.

Tablica 4.11 Mjerenja treće situacije kod *Realma*

REALM												
Upit\Xᵢ	**X₁[ms]**	**X₂[ms]**	**X₃[ms]**	**X₄[ms]**	**X₅[ms]**	**X₆[ms]**	**X₇[ms]**	**X₈[ms]**	**X₉[ms]**	**X₁₀[ms]**	**x̄ [ms]**	**σ**
Insert	1560	1397	1424	1435	1367	1374	1333	1351	1398	1399	1403.8	59.9
Get	5	6	8	9	10	10	8	6	9	12	8.3	2.1
Delete	2003	1996	2032	2143	2322	1985	2019	2114	2212	2392	2121.8	137.5

U Tablici 4.12 srednje vrijeme unosa se povećalo za 3191.2 milisekundi, vrijeme dohvaćanja za 216.4 milisekundi, a vrijeme brisanja podataka za 64 milisekunde.

Tablica 4.12 Mjerenja treće situacije kod *DBFlowa*

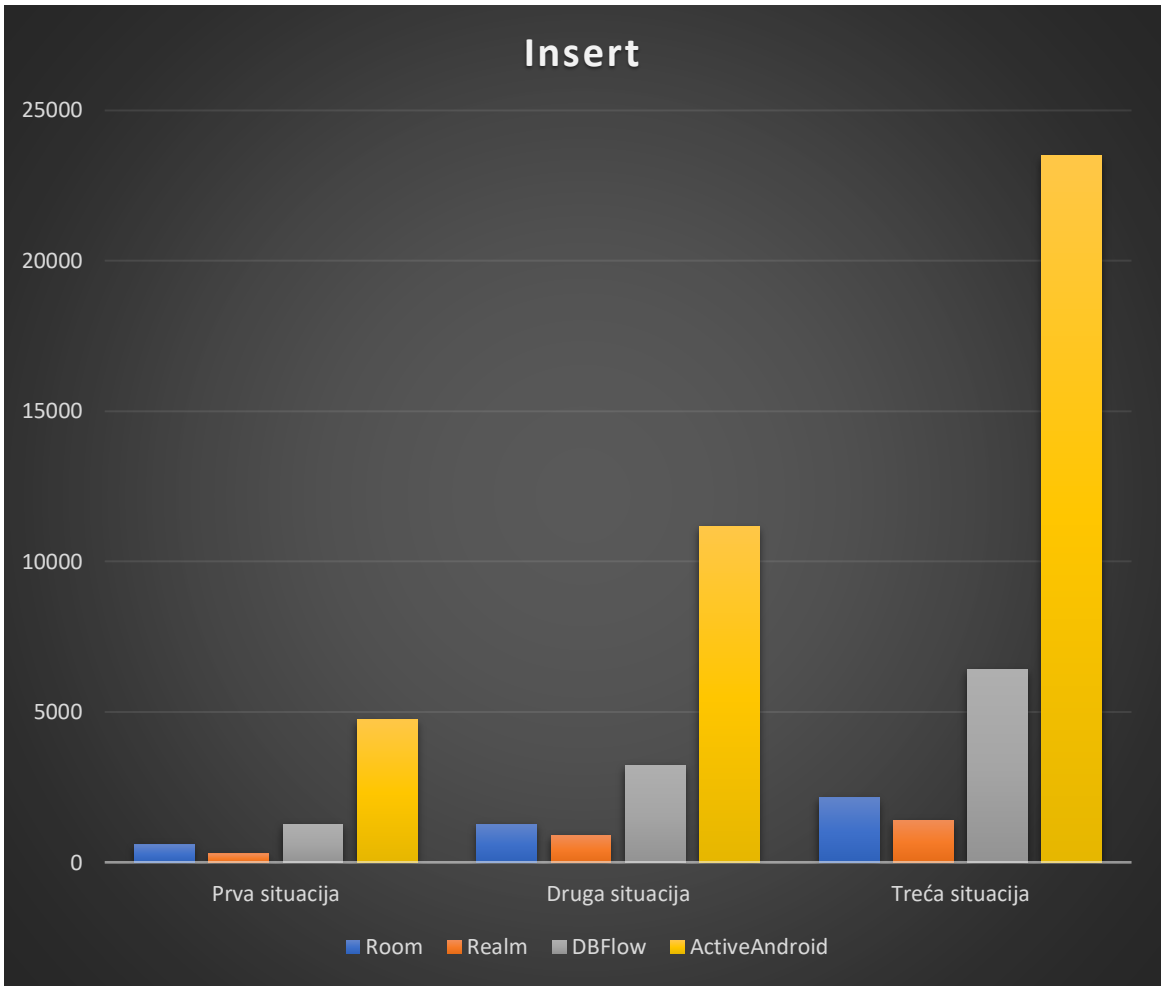
DBFLOW												
Upit\Xᵢ	**X₁[ms]**	**X₂[ms]**	**X₃[ms]**	**X₄[ms]**	**X₅[ms]**	**X₆[ms]**	**X₇[ms]**	**X₈[ms]**	**X₉[ms]**	**X₁₀[ms]**	**x̄ [ms]**	**σ**
Insert	7065	6245	7017	6442	6118	6215	6259	6332	6183	6192	6406.8	328.2
Get	466	502	400	438	379	283	363	533	344	285	399.3	81
Delete	141	133	143	132	132	125	139	133	134	136	134.8	4.9

U Tablici 4.13 prikazano je da se vrijeme unosa povećalo za 11914 milisekundi, vrijeme dohvaćanja za 293.5 milisekundi, a vrijeme brisanja za 24.9 milisekunde.

Tablica 4.13 Mjerenja treće situacije kod *Active Androida*

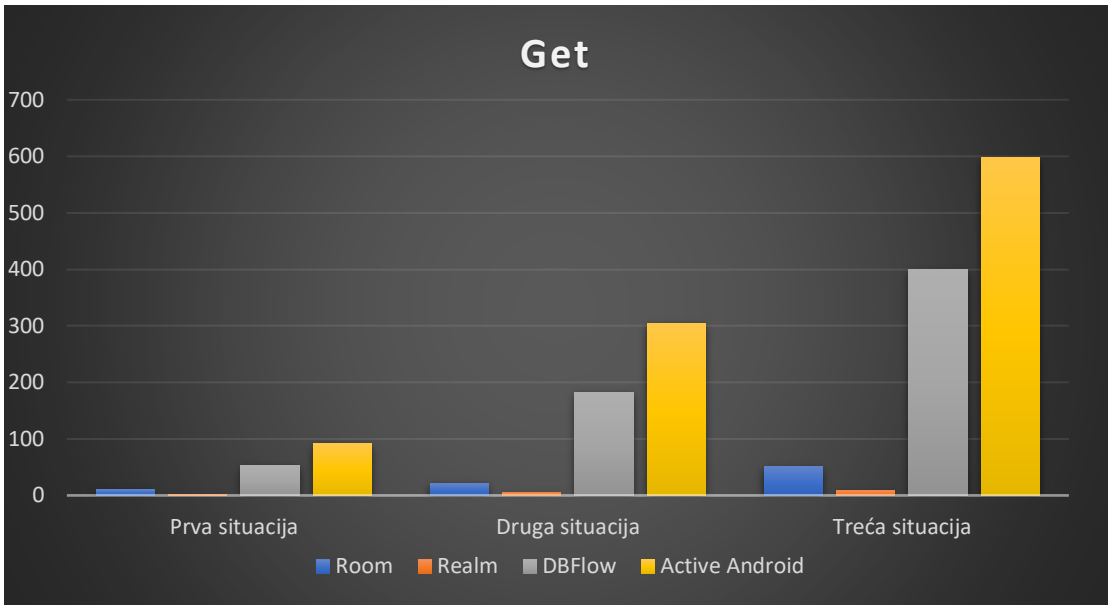
ACTIVE ANDROID												
Upit\Xi	**X1[ms]**	**X2[ms]**	**X3[ms]**	**X4[ms]**	**X5[ms]**	**X6[ms]**	**X7[ms]**	**X8[ms]**	**X9[ms]**	**X10[ms]**	**x̄ [ms]**	**σ**
Insert	24045	23750	22183	22820	23637	23978	23214	25914	22039	23290	23487	1043.5
Get	332	589	417	731	490	692	722	874	643	487	597.7	157.4
Delete	78	70	76	76	74	74	75	62	75	82	74.2	4.99

Slika 4.4 opisuje unos podataka u bazu u 3 različita slučaja. Na navedenoj slici prikazane su srednje vrijednosti vremena potrebnih za izvođenje upita. Najveće vrijeme unosa postiže *Active Android*, a *Realm* najmanje. To znači da se *Realm* pokazao kao najbolji alat u ovoj situaciji, a *Active Android* najlošiji. Vrijeme izvođenja i broj objekata su proporcionalni što znači da se povećavanjem broja objekata povećava i vrijeme potrebno da se upit izvede.



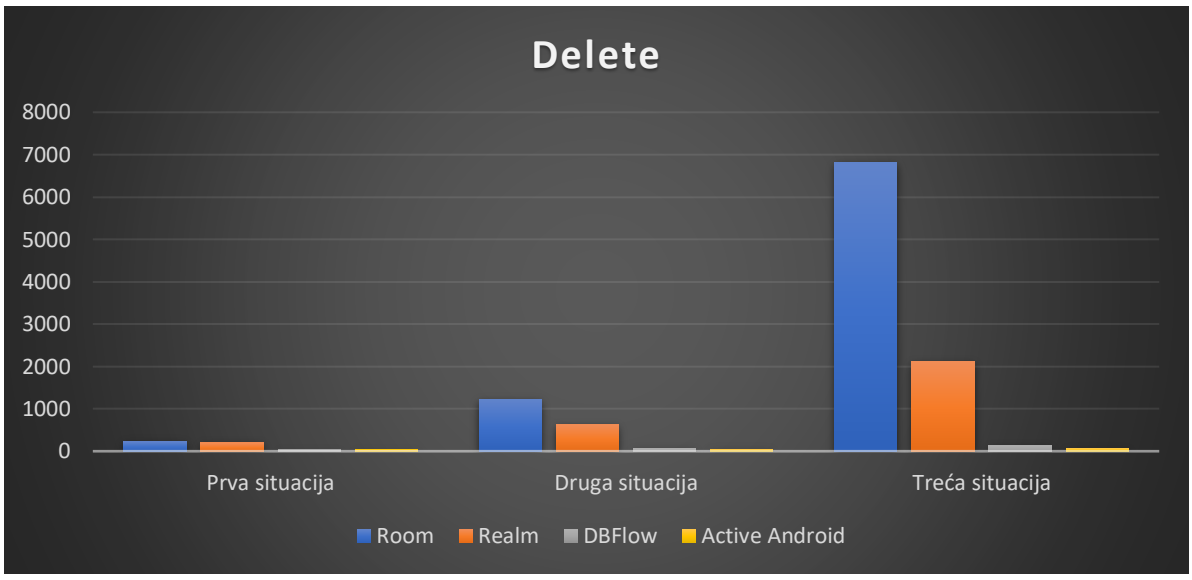
Slika 4.4 Prikaz srednjih vremena unosa podataka

Slika 4.2 opisuje dohvaćanje podataka iz baze u 3 različita slučaja. Na navedenoj slici prikazane su srednje vrijednosti potrebne za izvođenje spomenutog upita. Najbrže objekte dohvaćaju *Realm* i *Room*, a najsporije *Active Android* i *DBFlow*.



Slika 4.5 Prikaz vremena dohvaćanja podataka

Slika 4.6 opisuje brisanje podataka iz baze u 3 različita slučaja. Na spomenutoj slici prikazane su srednje vrijednosti potrebne za brisanja podataka. Ovo je jedini slučaj u kojem su *Active Android* i *DBFlow* postigli bolje rezultate od *Rooma* i *Realma*. U sve tri situacije *DBFlow* i *Active Android* imaju podjednaka vremena izvođenja.



Slika 4.6 Prikaz vremena brisanja podataka

5. ZAKLJUČAK

U ovome se završnom radu istraživala trajna pohrana podataka na Android platformi. Nakon uvoda i povijesnog pregleda Androida, govori se o arhitekturi platforme te strukturi aplikacije. Prelazi se na ključne pojmove, točnije, objektno-relacijskog preslikavanje te na četiri odabrana ORM alata (*Room, Realm, DBFlow, Active Android)*. Primjena ORM alata i izrada vlastite Android aplikacije baza su ovog istraživanja. U praktičnom se dijelu i izradila aplikacija koja je bila testirana na uređaju *Nexus 5X*. Aplikacijom su se pokazale operacije nad bazom koja sprema podatke o filmovima i glumcima, a zatim se mjeri vrijeme izvršavanja navedenih operacija. Ti su se podaci prikazali tablično i grafički te je uslijedio zaključak koji su alati najbolji u određenoj situaciji. Na osnovu istraživanja zaključeno je da *Room* i *Realm* imaju najbolje performanse i jednostavni su za rad zbog stalnog unaprjeđivanja i bogate dokumentacije.

Literatura

[1] J. Callaham, *The history of Android OS: its name, origin and more* [online], *Android Authority*, https://www.androidauthority.com/history-android-os-name-789433/, pristupljeno 22. rujna 2018. godine

[2] B. Phillips, C. Stewart, K. Marsicano, *Android Programming: The Big Nerd Ranch Guide (3rd Edition)*,Big Nerd Ranch, LLC, Atlanta, GA, 2017

[3] *Google Developers*, *Understand the Activity Lifecycle*, https://developer.android.com/guide/components/activities/activity-lifecycle, pristupljeno 12. rujna 2018. godine

[4] *Google Developers, Data and file storage overview*, https://developer.android.com/guide/topics/data/data-storage, 12. rujna 2018. godine

[5] D. Griffiths, D. Griffiths, *Head First Android Development, 2nd Edition*, O'Reilly Media, Inc., Sebastopol, CA, 2017

[6] *Techopedia, Object-Relational Mapping (ORM), https://www.techopedia.com/definition/24200/object-relational-mapping--orm*, pristupljeno 22. rujna 2018. godine

[7] M. Keith, M. Schincariol, *Pro EJB 3 Java Persistence API*, Springer-Verlag New York, Inc., 2006

[8] A. Kozubek-Krycun, *A Survey of Object-Relational Mapping (ORM) Libraries for Android and iOS* [online], Dzone, https://dzone.com/articles/a-survey-of-object-relational-mapping-orm-librarie, pristupljeno 12. rujna 2018. godine

[9] *Google Developers, Save data in a local database using Room*, https://developer.android.com/training/data-storage/room/, pristupljeno 12. rujna 2018. godine

[10] *Realm, What is Realm Platform?*, https://realm.io/docs/java/latest, pristupljeno12. rujna 2018. godine

[11] *Codepath, DBFlow Guide*, https://guides.codepath.com/android/DBFlow-Guide, pristupljeno 12. rujna 2018. godine

[12] *Codepath, Active Android Guide*,https://github.com/codepath/android_guides/wiki/ActiveAndroid-Guide, pristupljeno 12. rujna 2018. godine

Sažetak

Cilj ovog završnog rada bio je obraditi pojmove objektno-relacijskog preslikavanja i trajne pohrane podataka na Android platformi. Sve je to bilo potrebno prezentirati pomoću Android aplikacije koja je izrađena za potrebu ovog istraživanja. Za izradu aplikacije potrebna su osnovna znanja programskog jezika Jave i alata za ORM. Aplikacija omogućuje kreiranje, unos, brisanje i dohvaćanje objekata (filmova i glumaca) iz baze podataka. Svaka operacija koja će biti provođena na objektima davat će poruku na dnu ekrana o uspješnosti izvršenja. Na taj će način korisnik znati sve podatke o bazi u tom trenutku (npr. broj pohranjenih objekata). Bit će prikazano i vrijeme izvršenja operacije koje je uneseno u tablice, a time se dolazi i do zaključka koji su ORM alati bolji za navedeni zadatak.

Ključne riječi: baze podataka, objektno-relacijsko preslikavanje, ORM alati za Android

Abstract

The aim of this thesis was to research object-relational mapping and persistent data storage on the Android platform. The aforementioned terms were presented using the Android application that was developed for this purpose. Application development requires basic knowledge of Java programming language and ORM tools. The application allows you to create, insert, delete, and read objects (more specifically: movies and actors) from the database. Each operation executed on the objects gives out a message at the bottom of the screen in terms of performance execution. This means that the user will know all the data stored in the database at the specific time (e.g. the number of stored objects). The time needed for the operation to be executed will be also shown via tables, which will lead to a conclusion as to what ORM tools are the best for this case.

Keywords: databases, object-relational mapping, ORM tools for Android

Životopis

Ivan Mariić rođen je 11. svibnja 1995. godine u Slavonskom Brodu. Pohađao je Osnovnu školu Ivana Mažuranića u Sibinju. Nakon toga upisuje Klasičnu gimnaziju fra Marijana Lanosovića s pravom javnosti u Slavonskom Brodu. Tijekom srednjoškolskog obrazovanja osvaja 2. i 3. mjesto na Županijskim natjecanjima iz Geografije. Nakon završene gimnazije upisuje smjer Računarstvo na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku.

Prilozi

Na CD-u:

1. "Usporedba ORM alata pri trajnoj pohrani podataka unutar Android aplikacije" u .docx formatu
2. "Usporedba ORM alata pri trajnoj pohrani podataka unutar Android aplikacije" u .pdf formatu
3. Izvorni kod