Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Aleš Horvat

Mobilna aplikacija za vodenje osebnih financ

diplomsko delo

visokošolski študijski program
prve stopnje
računalništvo in informatika

Mentor: viš. pred. dr. Borut Batagelj

Ljubljana, 2019

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Za vodenje osebnih financ je na voljo veliko aplikacij. Vse bolj priročne so mobilne aplikacije, ki nam omogočajo hitrejše in sprotno vnašanje izdatkov. Še vedno pa je ročno vnašanje postavk računa v samo aplikacijo zelo zamudno.

V svoji nalogi preglejte kakšne so možnosti avtomatskega skeniranja računa in vnašanja posameznih postavk v obstoječo aplikacijo za vodenje financ ali pa izdelajte svojo aplikacijo za vodenje financ z možnostjo avtomatskega vnosa računov. Aplikacija naj omogoča tudi različne statistične preglede po različnih časovnih obdobjih in kategorijah.

Iskreno se zahvaljujem mentorju viš. pred. dr. Borutu Batagelju za mentorstvo in strokovno pomoč, ter svoji družini in prijateljem za vso podporo med študijem.

Svoji dragi Mariši.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
OCR	optical character recognition	optično prepoznavanje znakov
IDE	integrated development enviroment	integrirano razvojno okolje
SDK	standard development kit	standardni razvijalski paket
VM	virtual machine	navidezna strojna oprema
XML	extensible markup language	razširljiv označevalni jezik
JSON	javascript object notation	objektni JavaScript zapis
API	application programming interface	aplikacijski vmesnik za programiranje
SQL	structured query language	strukturiran povpraševalni jezik

Povzetek

Naslov: Mobilna aplikacija za vodenje osebnih financ

Avtor: Aleš Horvat

Namen diplomske naloge je bil razvoj mobilne aplikacije za vodenje osebne finančne evidence, ki teče na napravah z operacijskim sistemom Android. Aplikacija zajema funkcionalnosti, ki uporabniku olajšajo delo z vodenjem evidence o njegovih odhodkih in prihodkih. Omogoča tabelarični in statistični pregled nad vnešenimi transakcijami, katerih vnos je poenostavljen s pomočjo skeniranja računov. Aplikacija je bila razvita v okolju Android Studio, logični del je bil napisan v programskem jeziku Java, uporabniški vmesnik pa zgrajen s pomočjo označevalnega jezika XML. Za hrambo podatkov je bila uporabljena lokalna podatkovna baza SQLite, za optično prepoznavanje znakov iz slike računa pa brezplačna knjižnica Google Mobile Vision. V diplomski nalogi so navedeni ključni deli razvoja podatkovne baze in uporabniškega vmesnika, ter opisani algoritmi s pomočjo katerih aplikacija prebere potrebne podatke iz slike računa.

Ključne besede: Android, Java, optična prepoznava znakov, branje računov.

Abstract

Title: Personal finance mobile application

Author: Aleš Horvat

The purpose of this diploma thesis is to develop a mobile application for keeping track of personal finances, which runs on Android powered devices. The Application helps users store and display relevant data about their transactions. The records within the app can be displayed using flat tables or statistic charts, and can be entered using receipt scanning feature. The application was developed in Android Studio IDE, which was used for writing application logic with Java, and building user interface with XML. Data storage was implemented with SQLite database, while the receipt-scanning feature used Google Mobile Vision library for optical character recognition. The document states key parts of the database and user interface development, and describes algorithms used for receipt scanning.

Keywords: Android, Java, OCR, receipt scanning.

Poglavje 1

Uvod

Živimo v svetu, kjer je uporaba računalniških tehnologij neizogiben del našega vsakdanjika. Pametni telefoni so postali orodje, s katerim si lahko pomagamo na vseh področjih našega življenja. Za to so zaslužne mobilne aplikacije, ki pokrivajo vse širši spekter funkcionalnosti. Na njih se zanašamo predvsem zato, ker nam poenostavijo opravila, ki bi nam sicer vzela veliko časa.

Nepreglednost nad osebnimi finančnimi sredstvi je problem s katerim se sooča veliko število ljudi. Tudi sam se včasih sprašujem kam so šla moja sredstva v preteklem mesecu, na katerih področjih bi lahko prihranil nekaj denarja in kakšen je moj finančni napredek glede na pretekla obdobja.

V sklopu diplomske naloge sem se odločil razviti mobilno aplikacijo za vodenje osebnih financ, ki bo uporabnikom pomagala poiskati odgovore na ta vprašanja. Aplikacija bo uporabniku omogočala beleženje prihodkov in odhodkov v kontekstu enega ali več računov. Vnos transakcij v aplikacijo bo poenostavljen s pomočjo skeniranja računov, seveda pa bo aplikacija omogočala tudi ročni vpis. Uporabniki bodo nad vsemi transakcijami imeli tabelarični, in pa tudi statistični pregled s pomočjo grafov.

V nadaljevanju poglavja sledi analiza ključnih funkcionalnosti aplikacije: branje podatkov iz računa in shranjevanje podatkov.

1.1 Branje podatkov iz računa

Branje oziroma prepoznavanje relevantnih podatkov iz nekega računa je za človeka rutinsko opravilo, za katerega ne potrebujemo posebnega truda. Za računalnik pa ta naloga ni tako enostavna, saj je zanj slika le neko dvodimenzionalno polje števil, v katerem vsako od števil predstavlja barvo enega piksla na zaslonu. Da bi računalnik postal bolj podoben človeku in njegovemu razumevanju vizualnih informacij, skrbi področje računalniškega vida.

Računalniški vid

Kako računalnik naučiti sposobnosti interpretiranja slike kot skupek nekakšnih informacij, in ne samo polja števil? Naučimo ga prepoznave vzorcev. Če bi želeli, da računalnik prepozna ali se na podani sliki nahaja nek specifičen objekt, mu moramo najprej pokazati kako le ta izgleda. Računalniku podamo neko sliko, nad katero izvede določene matematične operacije, s katerimi pride do nekega rezultata, ki si ga zapomni. In če računalniku še enkrat podamo enako ali podobno sliko, bo nad njo spet izvedel te operacije, katerih rezultat se bo vsaj delno ujemal z rezultatom že videne slike, računalnik pa se bo potem odločil ali je na sliki videl iskani predmet ali ne. Vendar se slike na katerih so enaki predmeti med sabo močno razlikujejo. Na primer v barvi in perspektivi. To pomeni, da bo računalnik vzorec iz ene slike zelo težko povezal z vzorcem iz druge slike, ki vsebuje iste predmete iz druge perspektive pod drugačnimi svetlobnimi pogoji. Ta problem rešimo s sledečimi pristopi:

- sliko najprej obdelamo - lahko nad njo uporabimo nekakšen filter, (na primer povečamo kontrast) da določeni elementi slike pridejo do izraza,
- računalniku pokažemo čim več slik, da si zapomne vzorce, ki so skupni enakim predmetom na različnih slikah in so neodvisni od barve, perspektive, osvetlitve in drugih spreminjajočih se lastnosti.

Za branje podatkov iz računa bo aplikacija morala biti sposobna prepoznavanja alfanumeričnih znakov. S tem se ukvarja področje računalniškega vida, ki se imenuje optična prepoznava znakov (ang. optical character recognition, OCR).

Prepoznavanje znakov

Prepoznavanje znakov prav tako lahko implementiramo s prepoznavanjem vzorcev in sicer tako, da si program zapomni vrsto različnih stilov pisave in nato najprej ugotovi kateri stil pisave je na sliki, nato pa samo poveže ustrezne vzorce med sabo. Nekoliko naprednejši način je, da program prepoznava komponente iz katerih je znak sestavljen. V primeru črke A mora torej zaznati dve sklenjeni poševni črti, ki sta nekje na sredini povezani z vodoravno črto [14]. Tak način prepoznavanja znakov ne zahteva predhodnega poznavanja stilov in deluje na vseh pisavah (tudi na rokopisu), uporablja ga pa večina modernih aplikacij in knjižnic za OCR. Pri izdelavi aplikacije bomo uporabili eno od obstoječih brezplačnih knjižnic OCR [8].

Brezplačne knjižnice OCR

V nabor knjižnic primernih za uporabo v aplikaciji smo uvrstili 3 najbolj popularne, ki so na voljo na platformi Android. V nadaljevanju so naštete, ter na kratko opisane.

- OpenCV - Najbolj znana odprtokodna knjižnica, ki ponuja vse funkcionalnosti - od enostavnih do zelo naprednih. Uporabljamo jo lahko za procesiranje slik, videov in 3D objektov. Deluje na vseh platformah (Windows, Linux, MacOS, Android, iOS, FreeBSD, OpenBSD, Maemo) [15].

- Tesseract - Odprtokodna knjižnica namenjena branju teksta. Podpira več platform (Windows, Linux, MacOS, Android, iOS) [22].

- Google Mobile Vision - Knjižnica namenjena mobilnim napravam z operacijskim sistemom Android ali iOS. Omogoča detektiranje obrazov, bar kod oziroma QR kod, prepoznavanje objektov na slikah in videih in prepoznavanje teksta [8].

Odločili smo se za uporabo Google Mobile Vision. Res ne ponuja tako širokega nabora orodij kot na primer OpenCV, vendar je to kar zajema več kot dovolj za naš primer uporabe v aplikaciji. Ponuja enostaven API za izdelavo mobilnih aplikacij. Prepoznavanje teksta je zelo učinkovito in uporabniku se ni potrebno ukvarjati s predprocesiranjem slike, ki je v določenih primerih kot smo že prej omenili, nujno potrebno (npr. če na sliki niso jasno razvidne meje med objekti).

1.2 Shranjevanje podatkov

Podatke o vnešenih transakcijah je potrebno hraniti v podatkovni bazi. Lahko jih hranimo na strežniku ali pa lokalno (na mobilni napravi, na kateri je nameščena naša aplikacija). V našem primeru bo podatke potrebno hraniti lokalno, saj so na ta način uporabniku dostopni kadarkoli in kjerkoli. Očitna izbira lokalne podatkovne baze na mobilni napravi je SQLite, saj je integrirana v oba operacijska sistema, ki si lastita skoraj celotni trg pametnih telefonov, Android in iOS.

Podatkovna baza SQLite

To je relacijska baza, ki je privzeto vgrajena in že uveljavljena na mobilnih platformah Android in iOS. Za razliko od standardnih relacijskih podatkovnih baz, ne deluje po principu odjemalec-strežnik, ampak je del končne aplikacije. Aplikacija do baze dostopa preko funkcij, kar v primerjavi s strežniškimi bazami zmanjša dostopni čas. Baza je na napravi shranjena kot ena sama datoteka, ki je prenosljiva med platformami [19, 20]. Iz nje lahko sočasno bere več procesov, piše pa en sam, kar je lahko pomanjkljivost, vendar za

naš primer in večino ostalih mobilnih aplikacij ni pomembno, saj je pisanje v bazo storjeno s strani ene same aplikacije.

Poglavje 2

Sorodne aplikacije

V tem poglavju sledi analiza sorodnih aplikacij in njihovih funkcionalnosti. Trenutno na trgu obstaja kar nekaj aplikacij, ki imajo podoben namen kot aplikacija, ki jo bomo razvili v sklopu diplomske naloge.

Mobilne

- Money Manager Expense & Budget
- Smart Receipts
- Receipt bank: Auto Bookkeeping & Receipt Scanner
- Monefy - Money Manager
- Money Lover: Expense Tracker & Budget Planner

Namizne

- Moneydance
- Money Manager EX
- Quicken

V nadaljevanju so navedene in opisane najbolj popularne, ki spadajo med direktno konkurenco naši aplikaciji: to so aplikacije, ki ponujajo skeniranje računov in tečejo na mobilnih napravah.

Mobilne aplikacije, ki ponujajo skeniranje računov

Smart Receipts

Aplikacija Smart Receipts omogoča branje računov s pomočjo OCR in strojnega učenja, vendar zahteva plačilo za vsak skeniran račun (cena je 1,09 EUR za 10 branj računa). Branje računov po naših testiranjih deluje zadovoljivo, vendar ni v vseh primerih pravilno. Izdelovalci trdijo, da zanesljivost skeniranja računov s časom postane boljša. Račune lahko seveda vnašamo tudi na roke. Omogoča statistični pregled transakcij po dnevih, ter kategorijah. Zgodovino transakcij pa ima uporabnik na voljo v tabelaričnem pogledu, ki je urejen po datumu. Aplikacija omogoča tudi vnos potnih stroškov, kateri so obravnavani kot samostojna entiteta. Podatke je mogoče izvoziti v pdf ali csv formatu. Omogočeno je tudi shranjevanje na strežnik, da se podatki ob menjavi naprave ne izgubijo. Uporabniška izkušnja je zelo dobra. Aplikacija omogoča osnoven nabor funkcionalnosti, je enostavna za uporabo, ter lepa na izgled [18].

Money Lover: Expense Tracker & Budget Planner

Ponuja branje računov, vendar je le to narejeno na strežniku. Aplikacija zajame sliko in jo posreduje strežniku, ki potem po določenem času vrne prebrane rezultate, kar sodeč po testiranjih traja nekaj minut. Uporabnik lahko med tem časom nemoteno uporablja aplikacijo. V testnih primerih je bila pravilno prebrana le cena, vendar ne v vseh primerih. Imamo na voljo seznam z vsemi transakcijami, ki ga lahko filtriramo po obdobjih. V aplikaciji so zajete tudi nekatere statistične obdelave transakcij, kot so: graf porabe po mesecih, ter graf transakcij po kategorijah. Omogočen je tudi vnos ponavljajočih se transakcij, ki se izvedejo samodejno. Uporabnik si

lahko tudi nastavi opomnik o bližanju roka nekega plačila, ali pa opomnik o presežku porabe v nekem obdobju. Podatki se samodejno shranjujejo tudi na strežnik, tako da je izguba podatkov s tem preprečena. Aplikacija res ponuja širok nabor funkcionalnosti, vendar bi lahko bila bolj enostavna za končnega uporabnika. Slabost pri skeniranju računov je čas trajanja, ter slabša zanesljivost pri prepoznavanju podatkov na računu [12].

Receipt bank: Auto Bookkeeping & Receipt Scanner

Aplikacija je plačljiva, vendar imajo uporabniki na voljo 14 dnevno preizkusno dobo. Omogoča vnos transakcij, ki se shranijo na strežnik. Transakcije lahko vnesemo ročno, ali pa skeniramo račun. Skeniranje v testnih primerih ni delovalo. Uporabnik do vseh transakcij lahko dostopa preko tabelaričnega seznama. Nabor funkcionalnosti je omejen le na shranjevanje in pregled podatkov o transakcijah, kar je v primerjavi s konkurenčnimi aplikacijami slabost [17].

Poglavje 3

Uporabljena orodja in tehnologije

Mobilna aplikacija za vodenje osebnih financ je bila razvita za operacijski sistem Android. Izbira operacijskega sistema je bila narejena na podlagi razširjenosti med uporabniki, ter dostopnosti razvijalskih orodij. Aplikacija je bila razvita in testirana na napravi Samsung Galaxy S9, na katerem teče operacijski sistem Android 8.0.0 (Oreo).

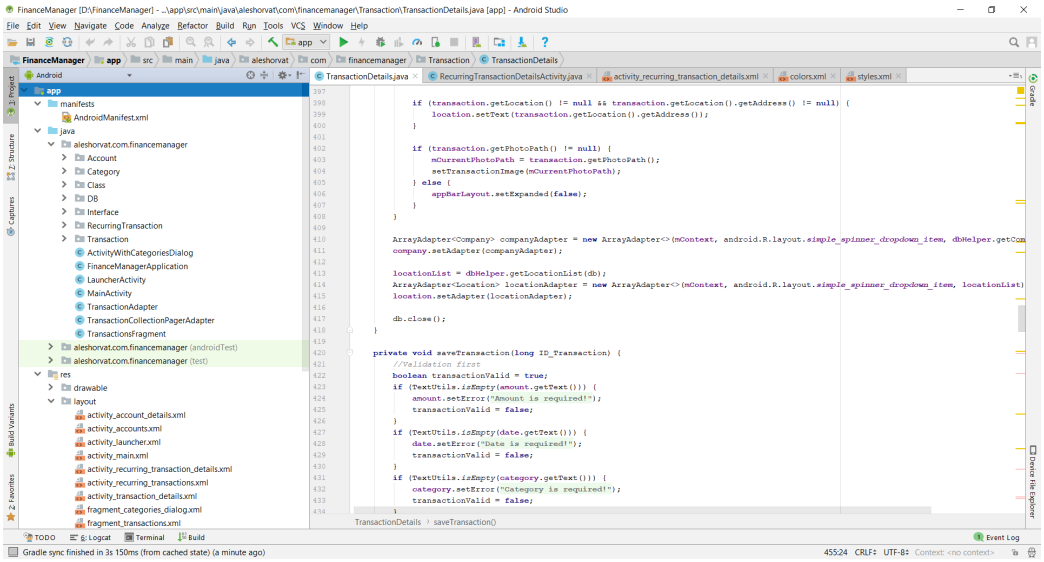
Celotna aplikacija, z izjemo podatkovnega modela, je bila razvita v Android Studio IDE-ju. Logični del aplikacije je bil napisan v programskem jeziku Java, uporabniški vmesnik pa definiran z označevalnim jezikom XML. Podatkovna baza je bila načrtovana z orodjem PowerDesigner.

3.1 Android Studio

Android Studio je uradno integrirano razvijalsko okolje (IDE) za razvoj aplikacij na Googlovem operacijskem sistemu imenovanim Android. Temelji na IntelliJ Idea IDE-ju, ki ga je razvilo podjetje JetBrains. Android Studio je v bistvu različica okolja IntelliJ Idea, ki je prilagojena za razvoj Android aplikacij. Orodje je povsem brezplačno, izdano pa je bilo leta 2014 kot nadomestilo za Eclipse Android Development Tools, predhodnim uradnim IDE-jem

za razvoj na platformi Android [4]. Okolje je prikazano na sliki 3.1.

Omogoča enostavno programiranje v jezikih Java, C/C++ in Kotlin s pomočjo inteligentnega urejevalnika kode, zajema standandardni razvijalski paket za razvoj Android aplikacij (Android SDK) in ponuja grafični urejevalnik uporabniških vmesnikov. Podpira tudi sisteme za kontroliranje verzij aplikacij kot so GitHub, Git in Google Cloud Source Repositories [5].



Slika 3.1: Razvijalsko okolje Android Studio

3.2 Java

Java je splošno namenski objektno orientiran programski jezik. Razvit je bil tako, da je čim bolj neodvisen od zunanje programske opreme. Največja prednost Jave je prenosljivost med platformami, kar pomeni da lahko teče na vseh platformah, ki podpirajo Java VM. Program napisan v Javi se najprej prevede v bajtno kodo, katero lahko neodvisno od strojne opreme poženemo s pomočjo Java VM programske opreme [10, 11]. To pomeni, da Java lahko teče

na vseh najbolj uporabljanih operacijskih sistemih, Windows, Mac, Linux in pa tudi Android.

S poznejšimi izdajami se je Java razčlenila na več različic, prilagojenih za različne tipe platform. Java Card je namenjena pametnim karticam, Java ME (Micro Edition) napravam z omejenimi viri (npr.: mobilne naprave), Java SE (Standard Edition) za osebne računalnike in Jave EE (Enterprise Edition) večjim organizacijskim omrežjem.

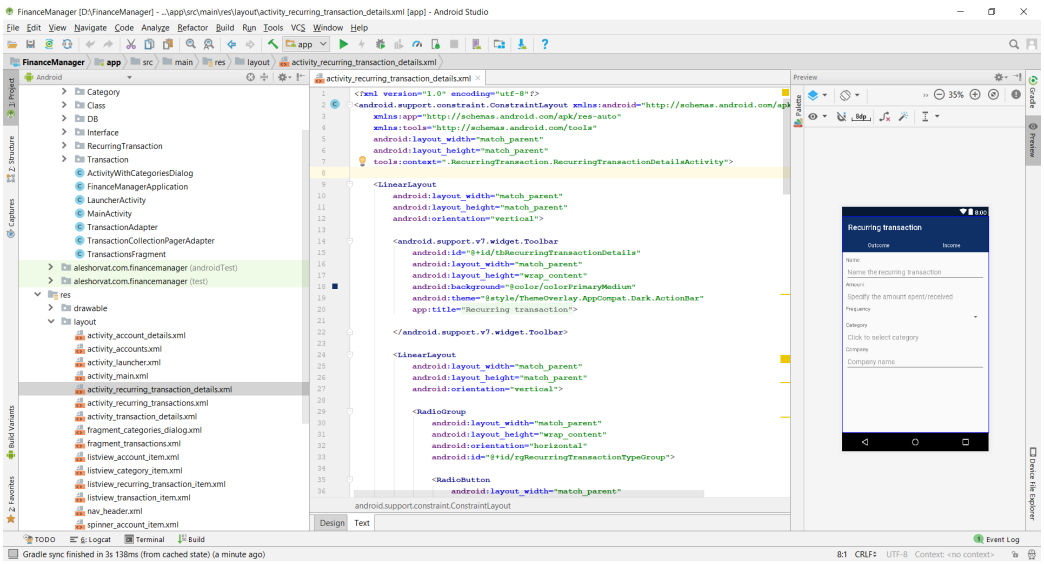
Sintaksa izvira iz vsem dobro poznanih jezikov C in C++, s tem da za razliko od njih ponuja abstrakcijo nižje nivojskih operacij, kot so kazalci na pomnilniška mesta, ki jih v omenjenih jezikih mora obvladovati programer. V primeru Jave pa za to poskrbi prevajalnik.

3.3 Razširljivi označevalni jezik

XML je razširljiv označevalni jezik, ki v računalništvu služi kot format za opis strukturiranih podatkov ali pa kot struktura za prenos podatkov med omrežji. Razdeljen je na 3 dele [23]:

- podatkovni,
- deklarativni in
- predstavitveni

Android SDK uporablja XML z namenom opisa strukturiranih podatkov, konkretno uporabniških vmesnikov. Razvijalsko orodje zajema predhodno zgrajene komponente za uporabniški vmesnik, ki jih pri razvoju aplikacije s pomočjo XML sintakse umestimo v strukturo uporabniškega vmesnika. Primer uporabe XML-ja za definiranje uporabniškega vmesnika prikazuje slika 3.2.



Slika 3.2: Uporaba XML-ja za definiranje uporabniških vmesnikov

3.4 PowerDesigner

PowerDesigner je programska oprema, ki se uporablja za modeliranje procesov in podatkov, načrtovanje aplikacij, ter obdelavo arhitekturnih metapodatkov. Orodje je bilo razvito s strani podjetja Sybase, katerega je potem kupilo nemško podjetje SAP. Slednji si trenutno lasti in še naprej razvija programsko orodje PowerDesigner [16].

Ponuja širok nabor funkcionalnosti, ki se uporablja predvsem na višjem nivoju načrtovanja aplikacij. Omogoča kreiranje procesnih, podatkovnih in fizičnih modelov.

Poglavje 4

Razvoj aplikacije

Predno smo se lotili programiranja, je bilo najprej potrebno narediti načrt aplikacije, ki je zajemal seznam funkcionalnosti, ki jih bo aplikacija ponujala uporabniku. Na podlagi funkcionalnosti iz načrta je bil s pomočjo orodja PowerDesigner, izdelan podatkovni model. V naslednjem koraku smo narisali zaslonske maske, ki so potrebne za implementacijo vseh funkcionalnosti. Na podlagi načrta smo se potem lotili dejanskega razvoja aplikacije.

Pri razvoju smo se najprej lotili implementacije podatkovne baze, ki je bila implementirana v sistemu za upravljanje s podatkovnimi zbirkami, ki se imenuje SQLite. Ko je bila podatkovna baza implementirana, smo izdelali še potrebne zaslonske maske, ter napisali logiko, ki omogoča uporabo aplikacije.

4.1 Podatkovna baza

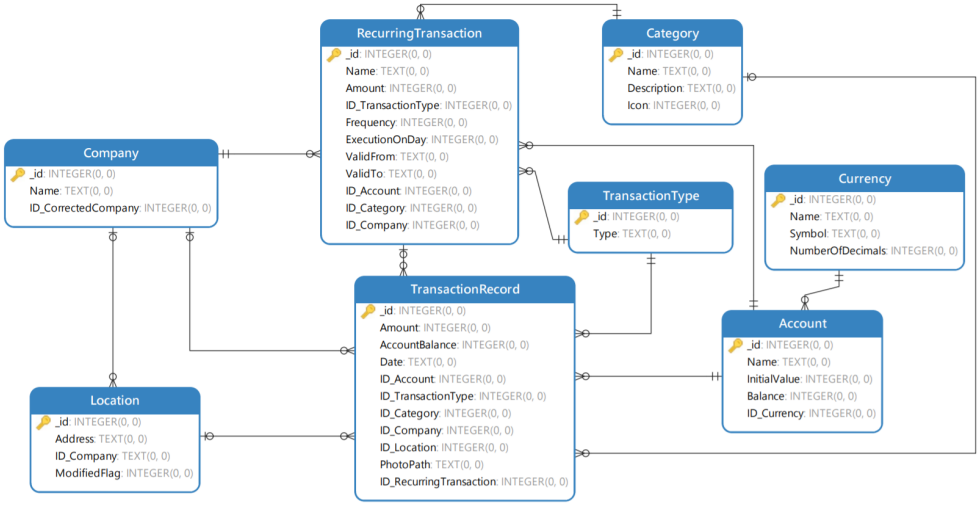
Glavni namen aplikacije je, da uporabniku pomaga pri beleženju njegovih finančnih transakcij. Osnovna entiteta v aplikaciji se imenuje transakcija. Zajema podatke o tipu transakcije (dohodna ali odhodna), znesku, datumu in kategoriji. Opcijski podatki na transakciji so lokacija ter podjetje. Transakcija je vezana na natanko en račun. Torej uporabnik mora najprej kreirati račun, ki ima začetno stanje, trenutno stanje in privzeto valuto. Uporabnik ima lahko tudi več računov. Na račun so vezane tudi redne transakcije, ki

skrbijo za avtomatičen vnos ponavljajočih se transakcij v podatkovno bazo. Lastnosti redne transakcije so sledeče: interval ponavljanja (leto, mesec, teden, dan), čas izvedbe (mesec - dan v mesecu, teden - dan v tednu, itd.), veljavnost (od - do), ter ostali podatki transakcije.

4.1.1 Model

Model je sestavljen iz osmih enitet, ki so skupaj z atributi naštete v sledečem seznamu. Slika 4.1 prikazuje povezave med naštetimi entitetami.

- transakcija (znesek, datum, račun, vrsta, kategorija, podjetje, lokacija, slika),
- vrsta transakcije (vrsta),
- kategorija (naziv, ikona, opis),
- račun (naziv, začetno stanje, trenutno stanje, valuta),
- valuta (naziv, simbol, število decimalk),
- podjetje (naziv),
- lokacija (naziv, podjetje),
- redna transakcija (naziv, znesek, vrsta, interval, čas izvedbe, velja od, velja do, račun, kategorija, podjetje).



Slika 4.1: Model s povezanimi entitetami

4.1.2 Implementacija podatkovne baze

Podatkovna baza, ki jo hočemo uporabljati v naši aplikaciji je lokalna, kar pomeni, da moramo poskrbeti za kreiranje le te na uporabnikovi napravi. V aplikaciji smo definirali razred, ki se imenuje **FinanceManagerDBHelper** in implementira metode iz razreda **SQLiteOpenHelper** [21]. **SQLiteOpenHelper** je del platforme Android za razvoj, ki razvijalcem pomaga pri delu s podatkovno bazo SQLite, katera je integrirana v sam operacijski sistem. Razred nam pomaga pri kreiranju in nadgradnji baze. Implementira metodo **OnCreate**, ki se pokliče samo ob prvem poskusu uporabe podatkovne baze (sistem išče željeno bazo, in v primeru neobstoja pokliče to metodo). Ta metoda mora torej zajemati logiko za kreiranje podatkovne baze. Baza s tabelami iz zgoraj navedenega modela se kreira s pomočjo skript SQL, ki se preko metode **OnCreate** poženejo na bazi (slika 4.2).

```
@Override
public void onCreate(SQLiteDatabase db) {
    db.execSQL(SQL_CREATE_CURRENCY);
    db.execSQL(SQL_CREATE_ACCOUNT);
    db.execSQL(SQL_CREATE_COMPANY);
    db.execSQL(SQL_CREATE_LOCATION);
    db.execSQL(SQL_CREATE_CATEGORY);
    db.execSQL(SQL_CREATE_TRANSACTION);
    db.execSQL(SQL_CREATE_RECURRING_TRANSACTION);

    preFillDB(db);
}
```

Slika 4.2: Kreiranje podatkovne baze

V razredu so definirane tudi pomožne metode za shranjevanje, spreminjanje, brisanje in pridobivanje podatkov iz kreiranih tabel. Vsaka tabela ima svojo metodo za kreiranje, spreminjanje, brisanje in pridobivanje zapisov po identifikatorju. Nekoliko bolj specifične poizvedbe, kot na primer izbira vseh transakcij v nekem obdobju, so prav tako napisane v metodah omenjenega razreda. Te metode se potem uporabljajo skozi celotno aplikacijo, kjer je potrebno delo s podatki iz baze. S takšnim načinom dobimo večjo preglednost in zanesljivost, saj poizvedbo napišemo samo enkrat in jo večkrat uporabimo. Primer metode prikazuje slika 4.3.

```
public long insertCategory(Category category, SQLiteDatabase db){
    ContentValues values = new ContentValues();
    values.put(FinanceManagerContract.Category.COLUMN_NAME, category.getName());
    values.put(FinanceManagerContract.Category.COLUMN_DESCRIPTION, category.getDescription());
    values.put(FinanceManagerContract.Category.COLUMN_ICON, category.getIcon());

    return db.insert(FinanceManagerContract.Category.TABLE_NAME, nullColumnHack: null, values);
}
```

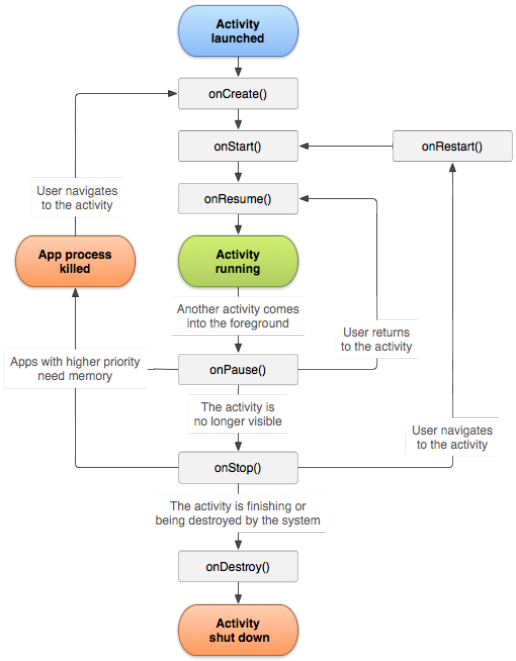
Slika 4.3: Primer kode za delo s podatki

4.2 Aktivnosti

Aktivnost predstavlja funkcionalnost, ki je v aplikaciji na voljo uporabniku. To je lahko celozaslonsko okno, pomanjšano okno ali pa tudi nima uporabniškega vmesnika [1]. V svojem življenjskem ciklu je lahko v enem od naslednjih stanj:

- aktivna,
- začasno ustavljena,
- ustavljena.

Aktivnost je v bistvu razred, ki implementira metode iz diagrama na sliki 4.4 - ovalni liki predstavljajo stanja, pravokotni pa metode, ki se kličejo ob spremembi stanja.



Slika 4.4: Življenjski cikel aktivnosti

Uporabniški vmesnik aktivnosti definiramo z označevalnim jezikom XML. V posebno datoteko hierarhično naštejemo vse elemente, jim določimo id-je (na njih se sklicujemo v kodi), ter nastavimo stile, kot prikazuje slika 4.5.

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Use DrawerLayout as root container for activity -->
<android.support.v4.widget.DrawerLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true">

    <!-- Layout to contain contents of main body of screen (drawer will slide over this) -->
    <FrameLayout
        android:id="@+id/content_frame"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <RelativeLayout
            android:layout_width="match_parent"
            android:layout_height="match_parent">

            <android.support.v7.widget.Toolbar
                android:id="@+id/toolbar"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:layout_alignParentTop="true"
                android:background="@color/colorPrimaryMedium"
                android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar" >

                <LinearLayout
                    android:layout_width="match_parent"
                    android:layout_height="match_parent"
```

Slika 4.5: Definicija uporabniškega vmesnika

Nato kreiramo razred, ki deduje metode iz razreda **Activity**, v katerem implementiramo metodo **OnCreate**, ki se kliče ob kreiranju aktivnosti. V njej kot uporabniški vmesnik najprej nastavimo datoteko XML v kateri smo definirali vse potrebne gradnike in potem pridobimo še reference do posameznih elementov, katerim nato dodelimo metode, ki se izvedejo ob različnih dogodkih (poslušalci), ter v njih obdelamo vnešene podatke. Primer implementacije aktivnosti prikazuje slika 4.6.

RecurringTransactionsActivity.java

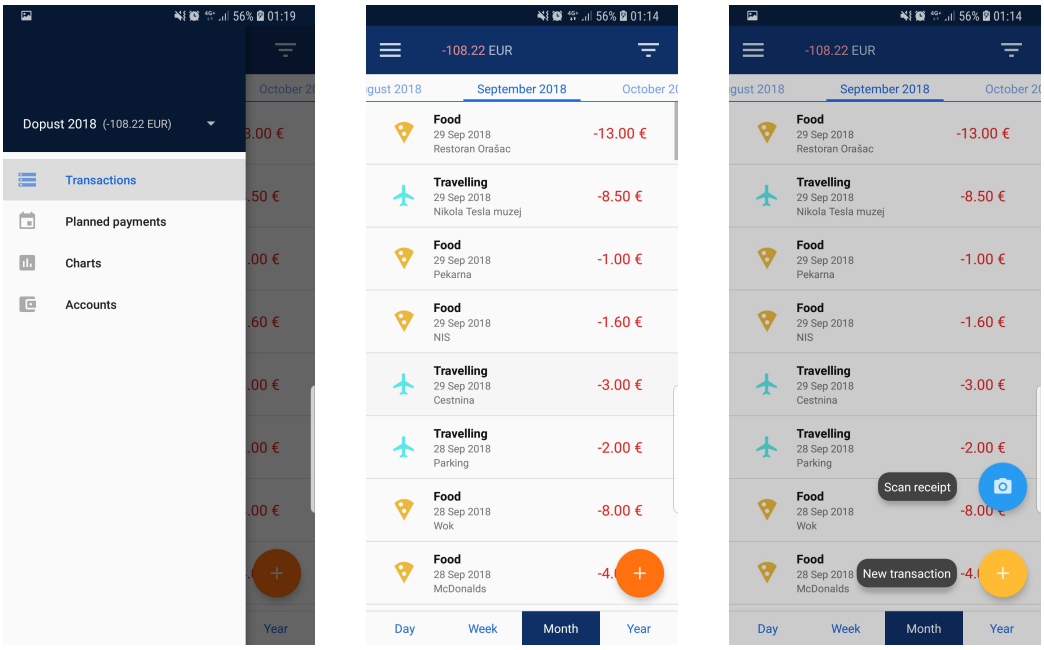
```
package aleshorvat.com.financemanager.RecurringTransaction;

import ...

public class RecurringTransactionsActivity extends AppCompatActivity {

    private FloatingActionButton recurringTransactionAddButton;
    private ListView recurringTransactionList;
    private Context mContext;
    private FinanceManagerDbHelper dbHelper;
    private RecurringTransactionAdapter recurringTransactionAdapter;
    private List<RecurringTransaction> recurringTransactions;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_recurring_transactions);
        mContext = this;
        dbHelper = new FinanceManagerDbHelper( context: this);

        Toolbar toolbar = findViewById(R.id.tbRecurringTransactions);
        setSupportActionBar(toolbar);
        getSupportActionBar().setDisplayHomeAsUpEnabled(true);

        recurringTransactionList = findViewById(R.id.lvRecurringTransactions);
        recurringTransactionAddButton = findViewById(R.id.fabRecurringTransactionsAdd);
        recurringTransactionAddButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent addRecurringTransactionIntent = new Intent(mContext, RecurringTransactionDetailsActivity.class);
                startActivityForResult(addRecurringTransactionIntent, addRecurringTransaction);
            }
        });
```

Slika 4.6: Primer implementacije aktivnosti

4.2.1 Transakcije

Ob zagonu aplikacije se uporabniku odpre glavna aktivnost, ki ponuja tabelarični pregled transakcij. Na vrhu se nahaja orodna vrstica, ki na levi strani vsebuje gumb za prikaz navigacije, na sredini stanje na trenutno izbranem računu, na desni pa gumb za napredno filtriranje seznama. Na samem dnu se nahaja še ena orodna vrstica, ki vsebuje gumbe za obdobja po katerih lahko grupiramo transakcije (dan, teden, mesec, leto). Uporabniški vmesnik prikazuje slika 4.7. Na sredini zaslona se nahaja kontrola imenovana **ViewPager** [2], ki vsebuje fragmente (fragmenti so opisani v nadaljevanju podpoglavja). Posamezen fragment vsebuje seznam transakcij v izbranem obdobju iz zavihka. Med zavihki se premikamo s pomočjo horizontalnega drsanja po zaslonu. Če imamo za obdobje izbran mesec, potem to pomeni da imamo v enem zavihku vse transakcije iz enega meseca. Vsak seznam oziroma zavihek torej predstavlja mesec. Med meseci se premikamo hori-

zontalno (z drsanjem levo ali desno). Element v seznamu predstavlja eno transakcijo. Prikazuje znesek, kategorijo, podjetje in datum transakcije. Za več podrobnosti oziroma urejanje zapisa kliknemo na element v seznamu in odpre se nov zaslon z vsemi podrobnostmi posamezne transakcije. Dodajanje novega zapisa nam omogoča lebdeči gumb (ang. floating action button), ki se nahaja v spodnjem desnem predelu zaslona (Slika 4.7 na sredini). Ob kliku na ta gumb se nam odpre podmeni, ki nam ponudi izbiro ročnega vnašanja transakcije ali pa skeniranja računa (Slika 4.7 desno). V primeru izbire ročnega vnosa se odpre neizpolnjen zaslon podrobnosti transakcije, če pa izberemo skeniranje računa, se nam odpre privzeta aplikacija za slikanje, s katero zajamemo račun in aplikacija nato sama iz slike prebere potrebne podatke na računu, ter izpolni vnosna polja na zaslonu podrobnosti transakcije (Slika 4.9).



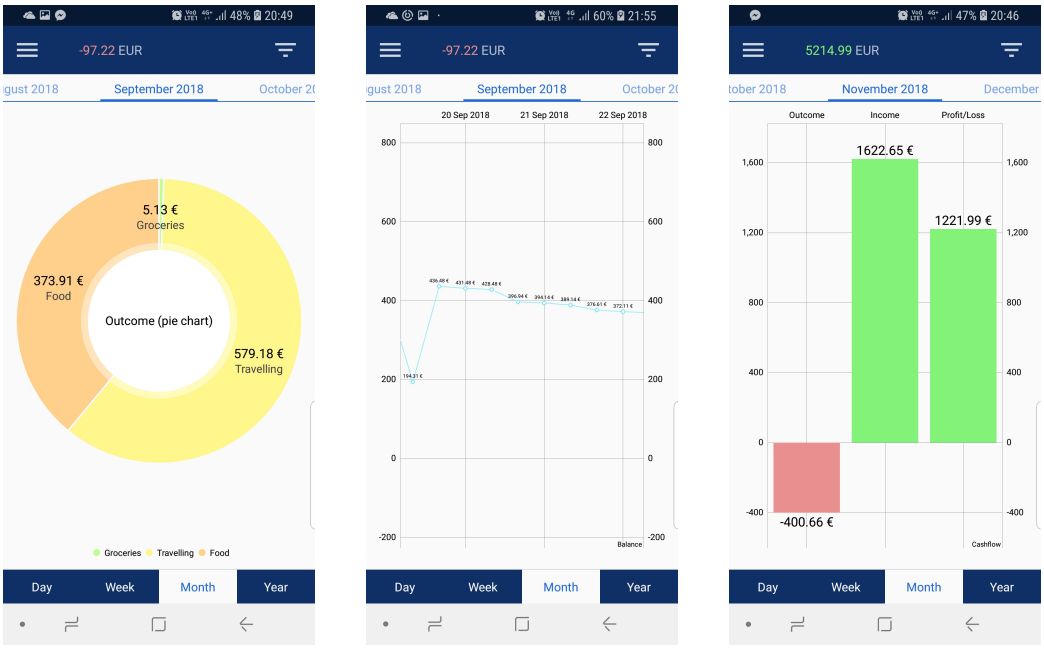
Slika 4.7: Primer aktivnosti, ki prikazuje seznam stransakcij. Možnost dodajanja nove transakcije (srednja slika) in možnost ročnega ali avtomatskega zajema (desna slika)

Vsebina zavihka znotraj **ViewPager** kontrole je implementirana s pomočjo komponente, ki se imenuje **Fragment**. Fragment v Android aplikaciji predstavlja del zaslonske maske in funkcionalnosti, ki jo ponuja aktivnost. V eni aktivnosti lahko s pomočjo fragmentov uporabniški vmesnik razdelimo na več manjših delov, hkrati pa en fragment lahko uporabimo v več aktivnostih [6]. Zavihki, ki zajemajo vsebino po obdobjih so v bistvu fragmenti, ki se dinamično kreirajo in brišejo ob menjavi zavihkov. V enem trenutku je v pomnilniku aktivnih 5 fragmentov (trenutno izbrani zavihek, dva predhodna in dva naslednja), zato da uporabniku ob menjavi zavihka ni potrebno čakati na inicializacijo vsebine. Ob menjavi zavihka se torej v ozadju na ločeni niti kreira nov in izbriše eden od obstoječih fragmentov, tako da sta poleg trenutnega fragmenta na vsaki strani na voljo še dva naslednja. Ob kreiranju, fragmentu v konstruktorju podamo argumente, od katerih je odvisna prikazana vsebina, fragment pa nato naredi poizvedbo na bazo, ter s pridobljenimi podatki napolni vsebino (seznam transakcij). V primeru polnjenja seznama transakcij, v argumentih podamo datum začetka obdobja, datum konca obdobja, ter identifikator računa, in fragment nato naredi poizvedbo na bazo, ki mu vrne vse transakcije v izbranem obdobju, ki so vezane na trenutni kontekst računa.

4.2.2 Statistični pregled transakcij

Aktivnost, ki prikazuje statistični pregled transakcij je zgrajena na podoben način kot glavna aktivnost, ki prikazuje sezname transakcij. Na vrhu ima orodno vrstico, ki vsebuje gumb za navigacijo, stanje trenutno izbranega računa, ter gumb za filtriranje. Na dnu je orodna vrstica z bližnjicami za grupiranje transakcij po obdobjih, največji del zaslona pa zajema kontrola, s pomočjo katere se lahko sprehajamo med grafi, ki prikazujejo podatke o transakcijah v različnih obdobjih. Uporabniški vmesnik je prikazan na sliki 4.8.

Aplikacija trenutno zajema tri statistične obravnave transakcij: tortni diagram, ki prikazuje odhodne stroške po kategorijah za izbrano obdobje, iz katerega lahko vidimo, kolikšen delež naših stroškov zavzema posamezna kategorija, črtni diagram, ki prikazuje gibanje stanja na računu v nekem obdobju, ter stolpični diagram iz katerega je razvidna primerjava odhodnih in dohodnih stroškov, ter profit oziroma izguba v izbranem obdobju.



Slika 4.8: Statistični pregled transakcij: tortni prikaz (levo), črtni (sredina) in prikaz profita oz. izgube (desno)

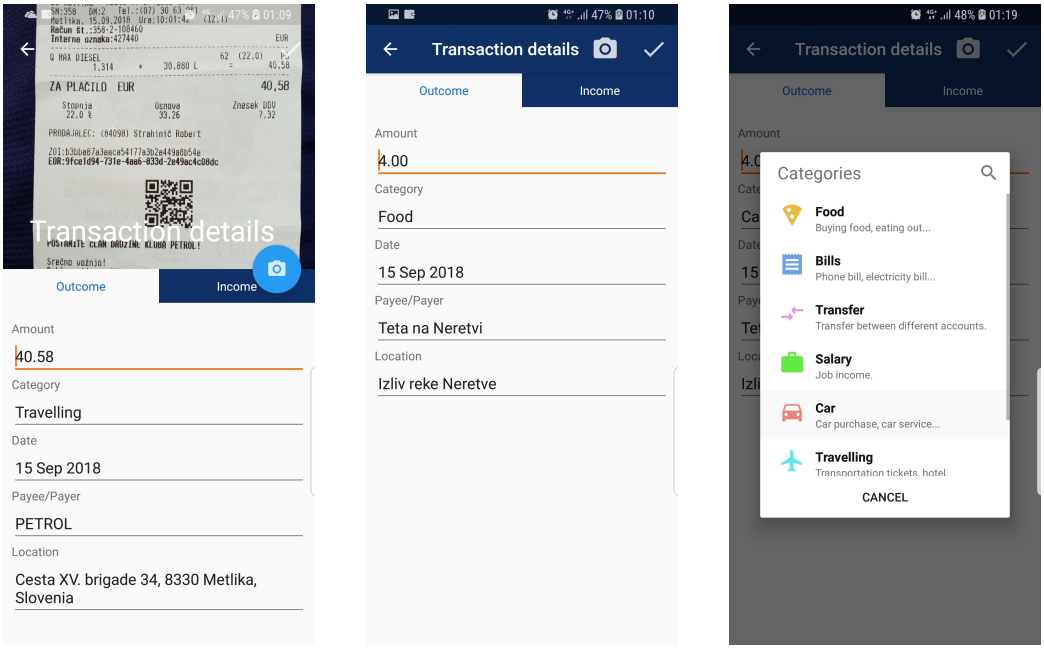
Grafi so implementirani s pomočjo knjižnice MPAndroidChart [13]. Knjižnica je povsem brezplačna in omogoča enostaven API za kreiranje črtnih, tortnih, slapovnih in stolpičnih diagramov. Grafi kreirani s to knjižnico so interaktivni, kar pomeni da omogočajo funkcionalnosti kot so povečava, dogodke ob kliku na določen del grafa, animacije in podobno.

Vsak tip grafa je implementiran kot ločena kontrola, kar pomeni da se v fragmentu, ki prikazuje diagram odločimo katero kontrolo bomo dodali na zaslonsko masko, ter katero poizvedbo na bazo moramo narediti, da pridobimo ustrezne podatke s katerimi lahko napolnimo graf. Vsak izmed treh implementiranih grafov prikazuje druge podatke, tako da ima vsak svojo namensko metodo, ki naredi ustrezno poizvedbo na bazo. Podatke o tipu grafa, obdobju za katerega delamo diagram, ter kontekst izbranega računa fragmentu podamo kot argumente v konstruktorju.

4.2.3 Podrobnosti transakcije

Ta aktivnost uporabniku omogoča vnos nove oziroma urejanje obstoječe transakcije. Uporabniški vmesnik prikazuje slika 4.9. Na vrhu se nahaja orodna vrstica, ki vsebuje gumb za nazaj, potrditev, ter skeniranje računa. Pod orodno vrstico se nahajajo vnosna polja v katera vnesemo podatke o računu:

- Vrsta transakcije - radijski gumb (odhodek, dohodek)
- Znesek - tekstovno polje omejeno na decimalno število
- Kategorija - ob kliku na vnosno polje se nam odpre dialog s seznamom kategorij
- Datum - ob kliku se nam odpre dialog s koledarjem za izbiro datuma
- Podjetje - tekstovno polje z možnostjo samodokončanja
- Lokacija - tekstovno polje z možnostjo samodokončanja

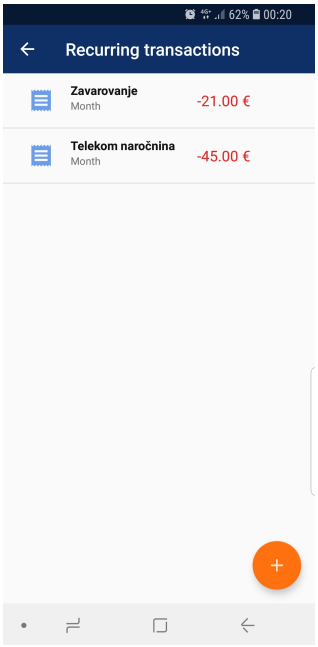


Slika 4.9: Prikaz podrobnosti transakcije skupaj s sliko računa (leva slika), ročni vnos podatkov (srednja slika) in izbira kategorije (desna slika)

Če uporabimo funkcijo skeniranja računa, aplikacija analizira zajeto sliko in sama napolni vnosna polja, ter shrani sliko računa, ki je vidna na vrhu aktivnosti. Uporabnik po potrebi popravi skenirane podatke, ter shrani transakcijo. Podrobnosti o funkcionalnosti skeniranja računa so opisane v naslednjem podpoglavju 4.3.

4.2.4 Redne transakcije

Redne transakcije predstavljajo tiste transakcije, ki se ponavljajo v nekih rednih intervalih. Primer takšne je odplačilo telefona po obrokih (15. dan, vsak mesec). Pri definiranju redne transakcije moramo izbrati interval v katerem se transakcija izvaja in enoto v intervalu. Aplikacija redne transakcije samodejno vnaša v bazo glede na določen interval, tako da se uporabniku ni potrebno ukvarjati z vnašanjem ponavljajočih se transakcij. Slika 4.10 prikazuje uporabniški vmesnik aktivnosti.



Slika 4.10: Aktivnost - redne transakcije

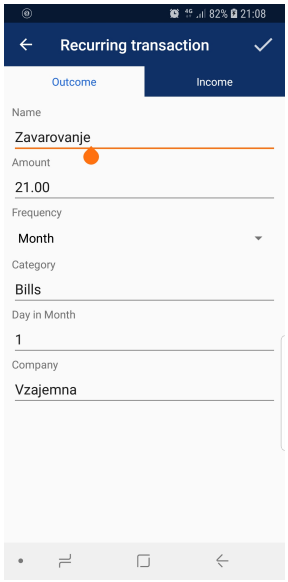
V tej aktivnosti uporabnik lahko vidi seznam vseh svojih rednih transakcij. Element v seznamu prikazuje ime transakcije, interval in znesek. Ob kliku na zapis v seznamu se nam odprejo podrobnosti redne transakcije. Nov zapis dodamo preko lebdečega gumba, brisanje zapisa je izvedljivo preko kontekstnega menija.

4.2.5 Podrobnosti redne transakcije

S pomočjo te aktivnosti, uporabnik vnese novo redno transakcijo oziroma uredi obstoječo. Polja, na zaslonski maski so sledeča (Slika 4.11):

- Vrsta transakcije - radijski gumb (odhodek, dohodek)
- Znesek - tekstovno polje omejeno na decimalno število
- Interval - spustni meni (interval izvedbe transakcije)
- Kategorija - ob kliku na vnosno polje se nam odpre dialog s seznamom kategorij

- Čas izvedbe transakcije (vsak dan, dan v tednu, dan v mesecu, dan v letu)
- Podjetje - tekstovno polje z možnostjo samodokončanja



Slika 4.11: Aktivnost za prikaz podrobnosti redne transakcije

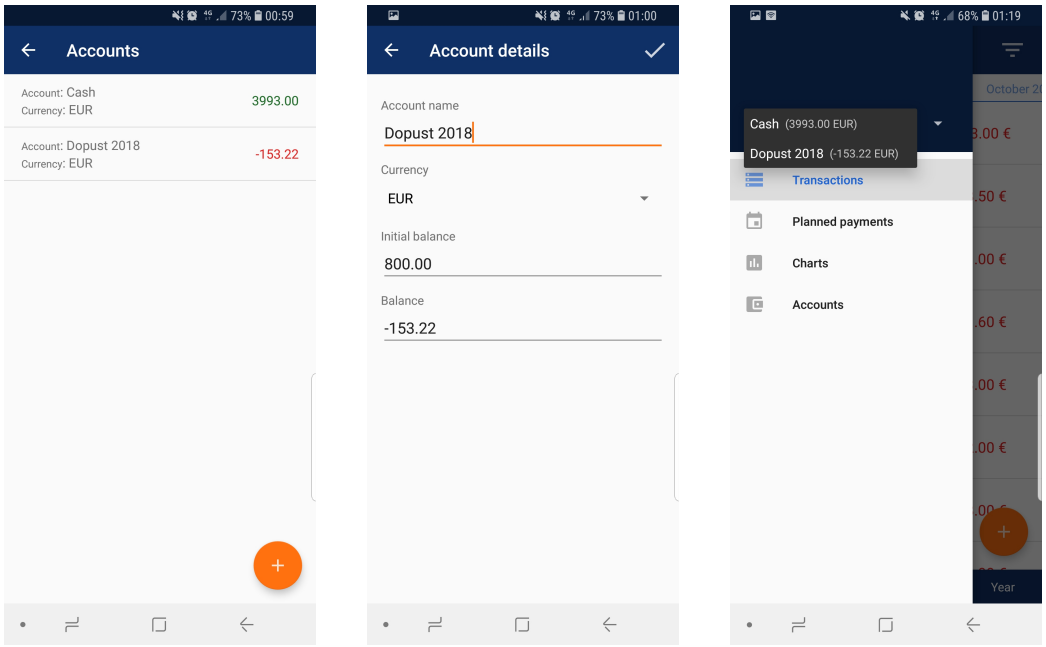
4.2.6 Računi

Uporabnik si mora v aplikaciji kreirati vsaj en račun na katerega so vezane vnešene transakcije. Lahko si kreira tudi več računov. Ta aktivnost prikazuje seznam vseh uporabnikovih računov. Omogoča brisanje, urejanje in dodajanje novih računov. Preklapljanje med računi je omogočeno v meniju aplikacije. Uporabniški vmesnik je prikazan na sliki 4.12.

4.2.7 Podrobnosti računa

Aktivnost, ki skrbi za podrobnosti računa omogoča urejanje obstoječega in dodajanje novega računa. Sestavljena je iz naslednjih polj:

- Naziv računa - tekstovno polje
- Privzeta valuta - spustni meni s podprtimi valutami
- Začetno stanje - tekstovno polje omejeno na decimalno število
- Trenutno stanje - tekstovno polje omejeno na decimalno število (samo za branje)



Slika 4.12: Računi, podrobnosti računa in menjava konteksta računa

4.3 Branje računa

Avtomatskega branja podatkov iz računa smo se lotili s pomočjo uporabe Google Mobile Vision API-ja, ki omogoča branje teksta iz zajete slike. Torej aplikacija mora omogočati tudi zajemanje slik.

4.3.1 Zajemanje slike

V operacijskem sistemu Android lahko sliko zajamemo tako, da najprej kreiramo datoteko v katero bo slika shranjena, nato kreiramo objekt tipa **Intent**, ki zažene aktivnost s privzeto aplikacijo za slikanje na napravi, kateremu priložimo še pot do kreirane datoteke [3]. Ko je slika zajeta in potrjena, se kliče metoda z imenom **OnActivityResult** v kateri potem zajeto sliko preberemo v objekt tipa **Bitmap**, ki ga pretvorimo v objekt tipa **Frame** in ga posredujemo knjižnici OCR, kar prikazuje slika 4.13.

```
Frame frame = new Frame.Builder().setBitmap(currentPhotoImage).build();
TextRecognizer textRecognizer = new TextRecognizer.Builder(this.mContext).build();
SparseArray<TextBlock> items = textRecognizer.detect(frame);
```

Slika 4.13: Branje teksta iz zajete slike

4.3.2 Uporaba Google Mobile Vision API-ja

Kot rezultat obdelave slike nam knjižnica OCR vrne polje objektov tipa **TextBlock**, v katerem posamezen element predstavlja odstavek oz. smiselno združeno komponento prebranega teksta, ki je nato razdeljen na več manjših komponent, prav tako tipa **TextBlock**, ki znotraj starša predstavljajo vrstice, ter še na nižjem nivoju, besede. Vsak tak objekt zajema tudi podatke o poziciji, in sicer odmik od levega, desnega, zgornjega in spodnjega roba [9].

Iz računa želimo prebrati končni znesek, datum izdaje računa, naziv podjetja, ki je izdalo račun, morebitno lokacijo (kje je bil račun izdan), ter v

katero kategorijo spadajo izdelki na računu. Za vsakega od podatkov je bil kreiran razred z implementacijo metode, ki prejme objekt tipa **TextBlock**. Vsak razred s pomočjo te metode v prejetih podatkih išče enega od naštetih atributov računa. Tekstovne bloke aplikacija najprej sortira, tako da si sledijo od zgoraj navzdol in od desne proti levi, ter se skozi njih sprehodi v zanki, v kateri potem vsaki instanci razreda, ki je zadolžena za iskanje specifičnega podatka v tekstu poda tekstovni blok - koda je prikazana na sliki 4.14.

```
for (Pair<Integer, Integer> orderedItem : orderedDetections) {
    dateEvaluator.evaluate(items.get(orderedItem.second));

    priceEvaluator.Evaluate(items.get(orderedItem.second));

    addressEvaluator.evaluate(items.get(orderedItem.second));
}

//Company is usually somewhere near the top, so the first block should contain it
companyEvaluator.evaluate(items.get(orderedDetections.get(0).second));
```

Slika 4.14: Procesiranje prebranega teksta

4.3.3 Algoritmi za skeniranje računa

Znesek

Kako izluščimo končni znesek iz podanega računa? Prvi korak je da izločimo tiste dele računa za katere smo sigurni, da to niso. Zagotovo lahko trdimo, da iščemo neko numerično vrednost, torej lahko iz teksta izločimo vrstice, ki ne vsebujejo numeričnih znakov. Vendar to ni dovolj, saj se številke pojavljajo tudi v drugih primerih, kot sta na primer datum in telefonska številka. Po analizi nekoliko večjega števila računov smo ugotovil, da so zneski vedno navedeni z decimalnim številom, tako da v aplikaciji lahko predpostavimo tudi to. Sedaj imamo samo vrstice, ki vsebujejo decimalna števila, ampak še vedno se nismo znebili takšnih primerov, kot je na primer datum. Zneski se od tovrstnih primerov razlikujejo po tem, da je pred celim delom zneska vedno presledek, ali pa je število prvi znak. Tukaj opazimo še to, da znesek ponavadi nima pripone, če pa jo že ima so to največ 3 (abecedni) znaki, ki predstavljajo valuto. Oznaka za valuto je od zneska lahko ločena tudi s presledkom.

Lahko rečemo, da smo s temi predpostavkami skozi filter spustili samo zneske na računu, tukaj mislim tudi zneske posameznih izdelkov, in če je to res, potem pomeni da je največje število med njimi tudi končni znesek. Na koncu se seveda izkaže, da temu ni tako. Na računih se med drugimi nahajajo podatki o odstotku DDV-ja in količina kupljenih izdelkov (včasih je zapisana z decimalko), kar pomeni da so v teh primerih lahko to največje številke na računu in da prej postavljena hipoteza pade, kar pomeni da bo algoritem potrebno še nekoliko nadgraditi.

Za veliko večino računov drži, da so posamezni zneski izdelkov vedno navadeni eden pod drugim in so desno poravnani, z istimi lastnostimi pa jim sledi tudi končni znesek. Tukaj lahko izkoristimo koordinate, ki nam jih poleg teksta vrne naša knjižnica OCR. Glede na to, da imamo desno poravnavo zneskov, lahko zneske, ki imajo približno enak odmik od desnega roba spravimo v skupen seznam. Tako dobimo sezname zneskov, ki v bistvu

predstavljajo stolpce. Če to vse drži, vzamemo zadnjo vrstico iz stolpca, ki ima najmanjši odmik od desnega roba in to naj bi bil naš končni znesek. Pa se spet izkaže, da pridemo do primerov, ko algoritem vrne napačen podatek. V stolpcu, kjer se nahajajo zneski izdelkov, ter končni znesek, je včasih tudi znesek s katerim je kupec plačal račun, ter vrnjen znesek. V tem primeru je aplikacija kot končni znesek prepoznala vrnjen znesek ali pa v primeru, da je le ta imel negativen predznak, plačan znesek.

Rešitev tega problema je, da seštejemo cene izdelkov in potem seštevek primerjamo s potencialnim končnim zneskom. Algoritem je sledeč: z zanko se po seznamu sprehodimo v smeri od zadnjega do prvega elementa. Trenutni element v zanki je n. Če je element na n-tem mestu končni znesek, pomeni da je seštevek elementov na mestih $n-1$, $n-2$, $n-x$ enak elementu na n-tem mestu. Slika 4.15 prikazuje algoritem za iskanje končnega zneska na računu.

```
float result = -1;
for(int i = amounts.size() - 1; i >= 0; i--){
    float finalAmount = amounts.get(i);
    float productSum = 0;
    boolean amountFound = false;

    // Add product prices
    for(int j = i-1; j >= 0; j--){
        productSum += amounts.get(j);
        if(productSum == finalAmount){
            amountFound = true;
            break;
        }
    }

    if(amountFound){
        result = finalAmount;
        break;
    }
}
```

Slika 4.15: Iskanje končnega zneska na računu

Sedaj lahko rečemo, da je algoritem že skoraj zadovoljiv. Problem nastane še, če imamo na katerem od izdelkov popust. V takih slučajih je med zneski izdelkov na računu tudi preračunan popust (znesek z negativnim predzna-

kom). Če skozi začeten filter spustimo čez tudi negativna decimalna števila, potem se algoritem obnese tudi v takih primerih.

Knjižnica OCR v določenih primerih napačno prebere tekst iz slike. Kar se lahko zgodi, je da na primer med decimalnim ločilom in številko zazna presledek, čeprav ga tam ni, ali pa recimo namesto števila 0 prebere znak O in podobno. Takšne napake OCR-ja moramo popraviti v prvem koraku, saj v primeru, da jih ne, že takoj na začetku naš filter lahko izloči vrstice, ki so potencialne za znesek.

Datum

Že prej smo omenili, da prebran tekst iz računa velikokrat vsebuje napake, saj so si določeni znaki med seboj preveč podobni. Ključno je, da te napake ne vplivajo na pravilnost algoritma za iskanje podatka med prebranim tekstom. Iz OCR-ja pogosto dobimo nazaj Z (črko) namesto 7 (število), presledek med dvema znakoma, čeprav ga dejansko ni, in tako naprej. Če v tekstu iščemo datum, v bistvu iščemo neko zaporedje številk in ločil, tako da lahko preventivno zamenjamo vse potencialno napačno prebrane znake s takšnimi ki ustrezajo našemu iskanemu podatku, kar prikazuje koda na sliki 4.16.

```
// OCR sometimes mixes up "." with ",". Change all "," to "."
candidate = candidate.replaceAll( regex: ",", replacement: ".");

// OCR can sometimes recognize spaces after/before date separators, so we need to clean them
candidate = candidate.replaceAll( regex: " ", replacement: "");

// OCR can sometimes mistake similar characters and numbers. Replace all letters with numbers.
candidate = candidate.replaceAll( regex: "O", replacement: "0");
candidate = candidate.replaceAll( regex: "B", replacement: "8");
candidate = candidate.replaceAll( regex: "I", replacement: "1");
candidate = candidate.replaceAll( regex: "Z", replacement: "7");
candidate = candidate.replace( target: "?", replacement: "7");
```

Slika 4.16: Preventivno popravljanje OCR napak

Formati v katerih se lahko pojavi datum so sledeči:

- DD.MM.YYYY
- DD-MM-YYYY
- YYYY/MM/DD
- DD/MM/YYYY
- YYYY.MM.DD
- YYYY-MM-DD

Formati imajo različna ločila med komponentami. Za lažje procesiranje jih poenotimo, tako da bodo sedaj datumi vedno v formatu DD.MM.YYYY ali YYYY.MM.DD (slika 4.17). Leto v datumu se lahko pojavi tudi v skrajšani obliki (YY). V takšnem primeru algoritem oceni kateri del datuma predstavlja letnico, ter mu pripne še prvi del (tisočletje in stoletje) iz trenutnega datuma.

```
// Map all date separators to . for easier processing
candidate = candidate.replaceAll( regex: "/", replacement: ".");
candidate = candidate.replaceAll( regex: "-",  replacement: ".");
```

Slika 4.17: Poenotenje ločil med komponentami datumov

Potem preverimo ali vrstica ki jo trenutno procesiramo vsebuje niz znakov, ki bi ustrezal enemu od dveh možnih formatov, ter iz nje izluščimo samo tekst, ki predstavlja datum. Na koncu s pomočjo formata kreiramo instanco razreda **Date**, ki predstavlja datum izdaje skeniranega računa (slika 4.18).

Lahko se zgodi, da je na računu več podatkov, ki ustrezajo enemu od pričakovanih formatov datuma. V takšnem primeru se vzame datum, ki je najbližji trenutnemu datumu.

Naziv podjetja

Naziv podjetja se običajno nahaja nekje pri vrhu računa, v večini primerov kar v prvi vrstici, tako, da je algoritem za iskanje naziva izdajatelja računa zelo enostaven. Vzamemo prvi tekstovni blok na računu, ki ponavadi vsebuje naziv podjetja, naslov podjetja, morebitni naslov enote, telefonsko številko in pa identifikator za DDV, ter ga procesiramo vrstico po vrstico. Prva vrstica je včasih samo niz znakov (*,-,., itd.), ki predstavljajo črto katera označuje začetek računa. Takšno vrstico algoritem za iskanje naziva zazna in izloči

```
Matcher m = pattern.matcher(candidate);
if(m.find()){
    // Trim everything around date
    dateString = candidate.substring(m.start(), m.end());
}else{
    continue;
}

// Find correct format
if(dateString.matches( regex: regexDay + regexDot + regexMonth + regexDot + regexYear)){
    dateFormat = new SimpleDateFormat( pattern: "dd.MM.yyyy");
}
else if(dateString.matches( regex: regexYear + regexDot + regexMonth + regexDot + regexDay))
{
    dateFormat = new SimpleDateFormat( pattern: "yyyy.MM.dd");
}

if(dateFormat != null)
{
    try {
        date = dateFormat.parse(dateString);
        finished = true;
        break;
    } catch (ParseException e) {
        e.printStackTrace();
    }
}
```

Slika 4.18: Kreiranje datuma

iz nabora kandidatov. To stori tako, da preveri, če trenutna vrstica vsebuje več kot 3 enake zaporedne znake in jo v takem primeru preskoči. Algoritem nato preveri če se katera koli izmed vrstic v prvem bloku ujema z imenom katerega od obstoječih podjetij v bazi. Če v tem primeru najde ujemanje, zaključi procesiranje in uporabniku vrne najdeno podjetje. V nasprotnem primeru predvideva, da se naziv nahaja v prvi vrstici.

Lokacija

Lokacijo prav tako iščemo na vrhnjem delu računa. V prvem koraku moramo samo zmanjšati nabor kandidatov, ki sploh lahko ustrezajo formatu naslovov, se pravi da moramo izločiti vse vrstice, ki ne vsebujejo potencialnega imena ulice in hišne številke (zaporedje več abecednih znakov, presledek in pozitivno število z morebitno pripono, ki označuje deljeni naslov). Zapise filtriramo z regularnim izrazom na sliki 4.19.

```
candidate = candidate.replaceAll( regex: ",", replacement: " ");
//Address can be in format Street 1, Street 1a, Street 1/a, Str. 1, Str. 1a, Str 1/a
if(candidate.matches( regex: ".*(\\p{L}){3} +\\d+((/?[a-zA-Z])|( .*))?") ||
        candidate.matches( regex: ".*(\\p{L})+\\. +\\d+((/?[a-zA-Z])|( .*))?"))
{
    addressCandidates.add(candidate);
}
```

Slika 4.19: Filtriranje kandidatov za naslov

Drugi korak je validacija kandidatov, ki ustrezajo našemu filtru. Naslov namreč ni zapisan v nekakšnem unikatnem formatu, saj je to le neko zaporedje besed in številk, ki nam brez predhodnega znanja ne pove veliko. Na izgled enak zapis (na primer: Miza 1048, Natakar 1) lahko aplikaciji izgleda kot legitimen naslov, vendar to ni. Za ločevanje med dejanskimi naslovi in zapisi, ki le izgledajo kot naslov bomo torej potrebovali neko predhodno znanje oziroma vir podatkov, s pomočjo katerega bomo razlikovali med takšnimi primeri.

Ko razmišljamo o podatkovni bazi, ki bi vsebovala čim večjo količino naslovov, bi si upal trditi, da veliki večini ljudem takoj na misel pride Google maps. Raziskali smo če Google mogoče že ponuja kakšno storitev, ki bi razvijalcem omogočala dostop do njihove baze podatkov. Izkaže se da res obstaja javna spletna storitev, ki se imenuje Geocoding API [7].

Storitev je dostopna preko HTTP protokola, z metodo GET. Pri zahtevku moramo podati format v katerem bi radi nazaj dobili podatke, na voljo pa sta JSON in XML.

Parametri, ki jih storitev prejme so sledeči:

- naslov (obvezen)
- regija
- komponente
- jezik

Aplikacija za vsak potencijalen naslov pošlje zahtevek do Geocoding API-ja, v katerem naslov poda kot parameter. Storitev nato nazaj vrne seznam

formatiranih naslovov. Če je seznam prazen, pomeni, da tekst, ki smo ga poslali ni veljaven naslov. Na računu se običajno nahajata eden do dveh veljavnih naslovov: naslov podjetja in naslov prodajne enote. Aplikacija se mora odločiti kateri od naslovov ima večjo verjetnost, da je pravilen. To stori tako, da vedno da prednost tistemu, ki v aplikaciji še ni bil popravljen s strani uporabnika, saj si v primeru, da uporabnik popravi podatek iz optičnega branja računa, to tudi zabeležimo v bazo. Algoritem za iskanje naslova je prikazan na sliki 4.20.

```
// Address
Location location = new Location();
List<String> autoCompleteScannedAddresses = new LinkedList();
for(String addressCandidate: addressEvaluator.getAddressCandidates())
{
    // Call Geocoding api
    String formattedAddress = AddressEvaluator.getFormattedAddress(addressCandidate);

    if(formattedAddress != null){
        autoCompleteScannedAddresses.add(formattedAddress);
        // Initially select the first address
        if(location.getAddress() == null){
            location.setAddress(formattedAddress);
        }else{
            // Check if previous address was at some point modified by user,
            // in that case next one has higher chance of being the correct one
            if(dbHelper.getLocationByAddress(location.getAddress(), db).isModifiedFlag()){
                location.setAddress(formattedAddress);
            }
        }
    }
}
// If we were not able to get location from scanned receipt, check in database
// for location connected to scanned company
if(location.getAddress() == null || location.getAddress().length() == 0){
    List<Location> existingLocations = dbHelper.getLocationListByCompany(company, db);
    if(existingLocations.size() > 0){
        location = existingLocations.get(0);
    }
}
```

Slika 4.20: Iskanje pravilnega naslova

Kategorija

Kategorijo v katero bi uporabnik umestil skenirani račun lahko v teoriji ugotovimo glede na podatke o izdajatelju računa, ter navedenih izdelkih na računu, vendar se v tem primeru ne moremo zanašati na izdelke, saj bi potem v podatkovni bazi morali hraniti zelo veliko število zapisov, saj so enaki izdeleki na računih različnih izdajateljev navedeni z različnimi nazivi. Algoritem torej uporabniku ponudi najbolj verjetno izbiro kategorije, glede na izdajatelja računa. Verjetnost pravilne izbire ne bo tako velika kot pri ostalih podatkih, saj tukaj gre za nekaj kar ni eksplicitno navedeno na računu, prav tako pa se kategoriziranje enakih računov lahko razlikuje med uporabniki, ki imajo v aplikaciji tudi možnost ustvarjanja svojih kategorij. Algoritem iskanja tega podatka je torej odvisen od predhodnih podatkov, zato bo na začetku uporabe aplikacije odstotek pravilno izpolnjenih kategorij najnižji, s časom pa se bo ta postopoma višal, ker se bo polnila tudi uporabnikova baza podatkov.

Algoritem poišče vse kategorije, ki so do trenutka skeniranja bile povezane na izdajatele računov, ter prešteje kolikokrat je bil izdajatelj povezan z neko kategorijo. Aplikacija nato izbere tisto, ki ima največje število zadetkov (slika 4.21). Če izdajatelja in izdelkov še nimamo v bazi, mora za kategorijo poskrbeti uporabnik.

```
public Category getCategoryForCompany(Company company){
    Category category = this.dbHelper.getMostCommonCategoryForCompany(company.getName(),
            dbHelper.getReadableDatabase());

    return category;
}
```

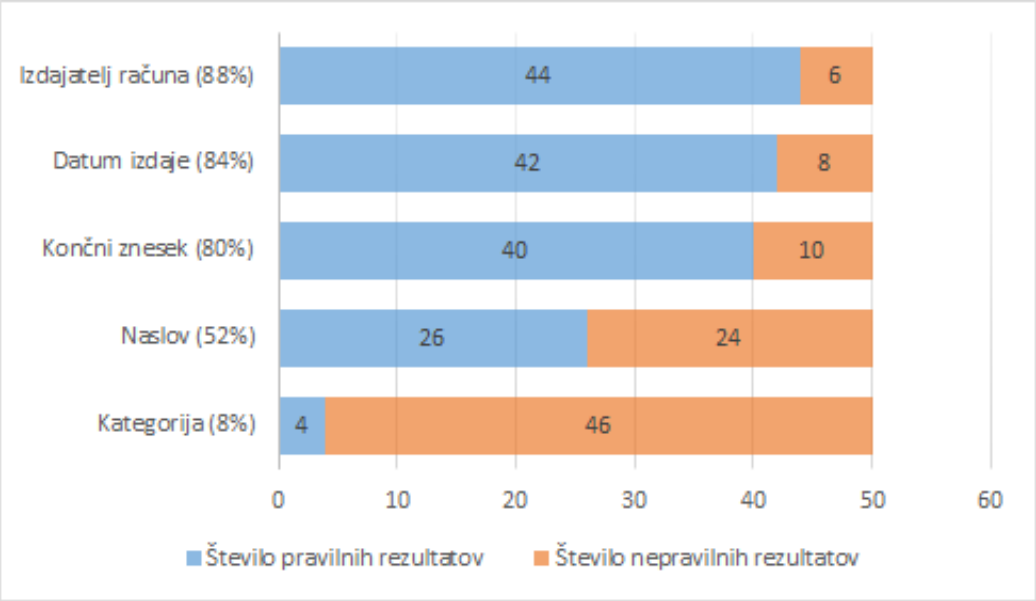
Slika 4.21: Postopek ugotavljanja kategorije

4.3.4 Učinkovitost algoritmov za skeniranje računa

Skeniranje računa s pomočjo tehnologije OCR je proces, katerega učinkovitost skorajda ne more biti sto odstotna. Glavni razlog za to je, da izdan račun nima nekega predpisanega formata na katerega bi se algoritmi za iskanje podatkov lahko zanašali. To pomeni, da se aplikacija pri prepoznavanju podatkov iz računa lahko tudi zmoti. Sledi statistična analiza pravilnosti skeniranih podatkov iz računa. Pri testiranju pravilnosti algoritmov smo uporabili 50 računov, ki med razvojem aplikacije niso bili uporabljeni v namen prilagajanja skeniranja.

Testiranje je bilo izvedeno tako, da smo vseh 50 računov poslikali, vse slike shranili v skupno mapo, ter vsaki sliki zdraven priložili tekstovno datoteko z enakim nazivom, v katero smo ročno zapisali podatke iz računa na sliki. V namen testiranja je bila napisana funkcija v aplikaciji, ki prebere vse slike iz podane mape, ter zraven vsake priloži datoteko v katero zapiše skenirane parametre računa. Tekstovne datoteke smo nato med sabo primerjali, ter si podatke o pravilnosti zabeležili v Excel-ov dokument, v katerem je nato bil kreiral tudi grafični prikaz pravilnosti algoritmov za skeniranje.

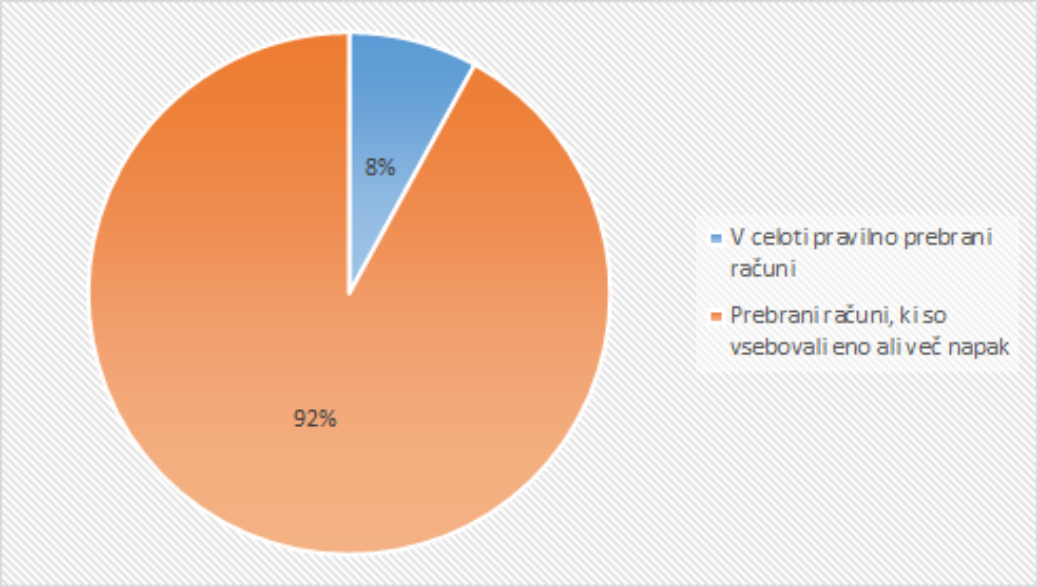
Končni znesek računa je bil pravilno prebran v 80 odstotkih (40 računov), datum izdaje v 84 odstotkih (42 računov), izdajatelj računa v 88 odstotkih (44 računov), lokacija oziroma naslov na katerem je bil račun izdan v 52 odstotkih (26 računov), pri kategoriji pa je odstotek pravilnosti znašal le 8 odstotkov (4 računi). Tukaj velja pripomniti, da je umeščanje računa v kategorijo odvisno od obstoječih podatkov v bazi, pravilnost prikazanega podatka pa je subjektivno mnenje posameznika. Dokaj nizek odstotek pravilnosti lokacije izdaje računa, pa je posledica dejstva, da se na računu v določenih primerih nahajata naslov podjetja in naslov prodajne enote, saj je 50 odstotkov napačno prebranih naslovov, posledica ravno tega, da je aplikacija namesto naslova prodajne enote izbrala naslov podjetja. Slika 4.22 prikazuje graf uspešnosti skeniranja po postavkah računa.



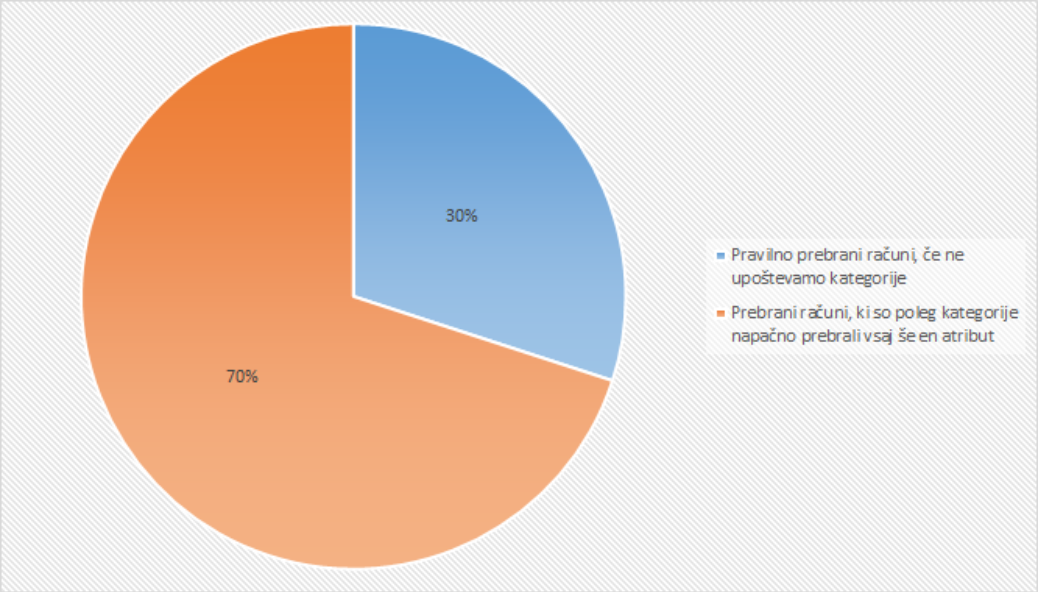
Slika 4.22: Učinkovitost algoritmov skeniranja računa po atributih

Število v celoti pravilno prebranih računov je bilo 4 od 50, kar je 8 odstotkov. Slab odstotek zanesljivosti lahko pripišemo kategoriji, ki je podatek ki ni eksplicitno naveden na računu in se lahko razlikuje od uporabnika do uporabnika. Če pri številu pravilno prebranih računov izvzamemo kategorijo, dobimo nekoliko večji delež uspešnosti skeniranja računa, in sicer 15 od 50 računov (30 odstotkov), kar prikazujeta grafa na slikah 4.23 in 4.24.

Z uporabo aplikacije je predvideno tudi povečevanje odstotka uspešnosti skeniranja računov, saj je uspešnost iskanja nekaterih elementov računa odvisna tudi od obstoječih podatkov v bazi.



Slika 4.23: Odstotek v celoti pravilno prebranih računov



Slika 4.24: Odstotek pravilno prebranih računov z izjemo kategorije

Poglavje 5

Sklepne ugotovitve

Aplikacija, ki je bila razvita v sklopu diplomske naloge zajema funkcionalnosti, ki uporabniku omogočajo beleženje njegovih transakcij, ter pomagajo pri organizaciji finančnih sredstev. Vnos transakcij je uporabniku olajšan s pomočjo skeniranja računov, kar je ena od prednosti pred konkurenčnimi aplikacijami, saj je trenutno na trgu zelo majhen delež aplikacij, ki ponujajo takšno storitev z zadovoljivo učinkovitostjo. Slabost aplikacije je, da ne omogoča izvoza podatkov in sinhronizacije lokalne podatkovne baze s strežnikom, med tem ko večina konkurenčnih aplikacij omenjeni storitvi ponuja. Za nekatere uporabnike je varnostna kopija oziroma prenosljivost podatkov med napravami zelo pomembna, zato bi bilo obe funkcionalnosti smiselno vključiti v eno od naslednjih različic aplikacije.

V možne izboljšave aplikacije bi tako na prvo mesto umestil izvoz podatkov v standardizirane formate, ter sinhronizacijo podatkovne baze na strežnik, saj sta to ključni funkcionalnosti za velik delež uporabnikov. Med funkcionalnosti, ki bi predstavljale dodano vrednost aplikacije, pa spadajo interaktivni grafi, sinhronizacija z bančnimi sistemi, opomniki o bodočih transakcijah, možnost kreiranja podkategorij, ter umeščanje transakcije v več kategorij hkrati.

Učinkovitost skeniranja računov je prav tako funkcionalnost, ki ima še

kar nekaj prostora za izboljšave. Potrebno bi bilo zgraditi občutno večjo bazo testnih primerov, kot smo jo uporabljali pri izdelavi aplikacije v sklopu diplomske naloge, ki bi služila za prilagajanje algoritmov iskanja podatkov na računih. Bazo testnih računov bi lahko nadgrajevali uporabniki sami, tako da bi v aplikaciji imeli možnost pošiljanja podatkov povezanih s skeniranjem računa na naš strežnik, z namenom izboljšanja storitve.

Literatura

[1] Android - aktivnost. Dosegljivo: https://developer.android.com/reference/android/app/Activity.

[2] Android - ViewPager. Dosegljivo: https://developer.android.com/training/animation/screen-slide.

[3] Android - camera API. Dosegljivo: https://developer.android.com/guide/topics/media/camera.

[4] Android Studio. Dosegljivo: https://en.wikipedia.org/wiki/Android_Studio.

[5] Android Studio - Predstavitev. Dosegljivo: https://developer.android.com/studio/intro/.

[6] Fragment. Dosegljivo: https://developer.android.com/guide/components/fragments.

[7] Google - Geocoding API. Dosegljivo: https://developers.google.com/maps/documentation/geocoding/intro.

[8] Google Mobile Vision. Dosegljivo: https://developers.google.com/vision/introduction.

[9] Google Mobile Vision - documentation. Dosegljivo: https://developers.google.com/vision/android/getting-started.

[10] Java - wikipedia. Dosegljivo: https://en.wikipedia.org/wiki/Java_(programming_language).

[11] Java VM. Dosegljivo: https://www.javaworld.com/article/3272244/core-java/what-is-the-jvm-introducing-the-java-virtual-machine.html.

[12] Money Lover: Expense Tracker & Budget Planner. Dosegljivo: https://play.google.com/store/apps/details?id=com.bookmark.money&hl=sl.

[13] MPAndroidChart. Dosegljivo: https://github.com/PhilJay/MPAndroidChart.

[14] OCR - kako deluje. Dosegljivo: https://www.explainthatstuff.com/how-ocr-works.html.

[15] OpenCV. Dosegljivo: https://docs.opencv.org/2.4/modules/core/doc/intro.html.

[16] PowerDesigner. Dosegljivo: http://powerdesigner.de/en/overview/.

[17] Receipt Bank: Auto Bookkeeping & Receipt Scanner. Dosegljivo: https://play.google.com/store/apps/details?id=com.receiptbank.android&hl=en_US.

[18] Smart Receipts. Dosegljivo: https://play.google.com/store/apps/details?id=wb.receipts&hl=en_US.

[19] SQLite - features. Dosegljivo: https://www.sqlite.org/features.html.

[20] SQLite - wikipedia. Dosegljivo: https://en.wikipedia.org/wiki/SQLite.

[21] SQLiteOpenHelper. Dosegljivo: https://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.

[22] Tesseract OCR. Dosegljivo: https://github.com/tesseract-ocr/docs/blob/master/tesseracticdar2007.pdf.

[23] XML. Dosegljivo: https://sl.wikipedia.org/wiki/XML.