Mobilna aplikacija za pretvorbu slike s tekstom u tekst primjenom tehnologije optičkog prepoznavanja znakova

Grudić, Tomislav

Undergraduate thesis / Završni rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Electrical Engineering and Computing / Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:168:587919

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-14**

Repository / Repozitorij:

FER Repository - University of Zagreb Faculty of Electrical Engineering and Computing repozitory

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1263

MOBILNA APLIKACIJA ZA PRETVORBU SLIKE S TEKSTOM U TEKST PRIMJENOM TEHNOLOGIJE OPTIČKOG PREPOZNAVANJA ZNAKOVA

Tomislav Grudić

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1263

MOBILNA APLIKACIJA ZA PRETVORBU SLIKE S TEKSTOM U TEKST PRIMJENOM TEHNOLOGIJE OPTIČKOG PREPOZNAVANJA ZNAKOVA

Tomislav Grudić

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

ZAVRŠNI ZADATAK br. 1263

Pristupnik: **Tomislav Grudić (0036537916)**
Studij: Elektrotehnika i informacijska tehnologija i Računarstvo
Modul: Računarstvo
Mentorica: prof. dr. sc. Ljiljana Brkić

Zadatak: **Mobilna aplikacija za pretvorbu slike s tekstom u tekst primjenom tehnologije optičkog prepoznavanja znakova**

Opis zadatka:

Aplikacije koje pomoću tehnologije optičkog prepoznavanja znakova (Optical Character Recognition - OCR) pretvaraju sliku s tekstom u stvarni tekst omogućuju korisnicima da brzo i jednostavno pretvore dokument u digitalni format i koriste ga u različite svrhe. Nakon što se tekst prepozna, moguće ga je pretraživati, uređivati i dijeliti. Potrebno je proučiti i usporediti funkcionalnosti postojećih mobilnih aplikacija za pretvorbu slike s tekstom u tekst primjenom tehnologije OCR. Potom je potrebno je osmisliti i implementirati vlastitu mobilnu aplikaciju koja će pomoću tehnologije OCR pretvarati sliku s tekstom u stvarni tekst i potom omogućiti uređivanje teksta u ugrađenom tekstualnom editoru. Radi poboljšanja kvalitete prepoznavanja teksta, aplikacija prije pretvorbe slike treba obaviti jednostavno pretprocesiranje slike koje bi uključivalo povećanja kontrasta i pretvorbu slike u crno-bijelu kako bi se olakšalo prepoznavanje teksta. Dodatno, aplikacija treba omogućiti provjeru pravopisa za zadani jezik, spajanje više stranica u isti tekst (primjerice, pri skeniranju knjiga), dodatno formatiranje teksta, i izvoz prepoznatog teksta u PDF formatu.

Rok za predaju rada: 14. lipnja 2024.

Iskreno se zahvaljujem mentorici prof. dr. sc. Ljiljani Brkić na pomoći i podršci koju mi je pružila tijekom izrade završnog rada.

Sadržaj

Uvod 1

1. Funkcionalni zahtjevi 2

2. Nefunkcionalni zahtjevi 3

3. Korištene tehnologije i alati 4

 3.1. Osnovne tehnologije i alati 4

 3.2. Vanjske biblioteke 4

4. Arhitektura sustava 5

 4.1. Opis i model baze podataka 6

 4.2. Implementacija programskog rješenja 6

  4.2.1. Opća organizacija klasa i upravljanje bazom podataka 6

  4.2.2. Pretprocesiranje fotografija 13

  4.2.3. Implementacija tekstualnog uređivača 15

5. Upute za korištenje i instalaciju 18

 5.1. Korisničke upute 18

 5.2. Instalacija i pokretanje aplikacije 29

Zaključak 31

Literatura 32

Sažetak 33

Summary 34

Uvod

Razvojem pametnih telefona i širokom dostupnošću interneta u svijetu se sve više koriste mobilne aplikacije koje postaju neophodan dio svakodnevnog života. Razvojem mobilnih aplikacija raznih namjena, mobilni uređaji ne služe više samo za komunikaciju već za obavljanje različitih poslova iz svakodnevnog života i za zabavu.

Tehnološki napredak omogućio je razvoj sofisticiranih alata koji mogu razlučiti uzorke i znakove iz slika i pohraniti ih u digitalnom formatu. Ta tehnologija, poznata kao optičko prepoznavanje znakova (OCR - Optical Character Recognition), pruža korisnicima mogućnost brzog i preciznog pretvaranja teksta iz fotografija u digitalni format.

Mobilne aplikacije za prepoznavanje teksta iz slike su praktično rješenje za problem konverzije fizičkih dokumenata u digitalni oblik. Brojni moderni pametni telefoni imaju OCR tehnologiju direktno implementiranu u kameru uređaja, dok su kod nekih modela za prepoznavanje teksta iz slike potrebne mobilne aplikacije.

U okviru ovog završnog rada, implementirana je mobilna aplikacija za sustav Android koja korisnicima nudi mogućnost prepoznavanja teksta iz jedne ili više slika te dodatno uređivanje teksta u ugrađenom tekstualnom uređivaču. Uz to aplikacija nudi mogućnost lakog dijeljenja prepoznatog sadržaja te izvoz u PDF formatu.

Motivacija za izbor ove teme proizlazi iz želje za učenjem načina na koji se razvijaju i implementiraju mobilne aplikacije za sustav Android te upoznavanje s pravilima, principima i obrascima koji vrijede u programiranju mobilnih aplikacija. Kroz ovaj projekt, bio mi je cilj steći praktično iskustvu u razvoju Android aplikacija, od početnih koraka planiranja i učenja osnova, preko implementacije do konačnog testiranja aplikacije.

1. Funkcionalni zahtjevi

Mobilna aplikacija treba služiti kao alat za prepoznavanje teksta iz fotografije te prikazivanje prepoznatog teksta u jednostavnom ugrađenom tekstualnom uređivaču. Fotografije za obradu trebaju se moći priložiti iz datotečnog sustava mobilnog uređaja s instaliranim operacijskim sustavom Android ili direktnim fotografiranjem kamerom uređaja. Prilikom učitavanja fotografije treba postojati mogućnost izrezivanja fotografije da bi tekst koji je potrebno prepoznati bio u fokusu. Aplikacija treba zatražiti dopuštenje korisnika za pristup kameri uređaja. Aplikacija mora moći obraditi više slika od jednom, odnosno da spoji tekst iz više slika u jedinstven tekst. Ako se priloži više fotografija, treba postojati mogućnost promjene redoslijeda obrade fotografija. Prije same obrade fotografija aplikacija treba obaviti pretprocesiranje slika kako bi poboljšala kvalitetu prepoznavanja znakova, pretprocesiranje treba obaviti povećanje kontrasta izvorne slike i pretvorbu slike u crno bijelu. Tekst bi se trebao moći kopirati u među spremnik i dijeliti putem instaliranih aplikacija koje podržavaju dijeljenje teksta kao što su e-mail, SMS poruke te razne aplikacije za dopisivanje. Uređivač treba nuditi mogućnost mijenjanja veličine i stilova teksta, te dodavanje novog teksta ili brisanje postojećeg. Prilikom uređivanja teksta aplikacija bi trebala koristiti ugrađenu provjeru pravopisa operacijskog sustava za jezik postavljen u postavkama sustava. Provjera pravopisa u uređivaču crvenom linijom treba podcrtati riječi za koje sustav smatra da su pogrešno napisane te predložiti zamjenske riječi. To će pružiti dodatnu mogućnost korisniku da provjeri ako se prepoznavanje teksta iz slike ispravno izvršilo. Aplikacija treba implementirati funkcionalnost za pretvaranje teksta u uređivaču u PDF dokument koji će se pohraniti u datotečni sustav uređaja. Nakon generacije i preuzimanja PDF dokumenta, korisniku se treba ponuditi opcija da dokument otvori u zadanom PDF pregledniku. Generirani PDF dokument treba sadržavati primijenjene stilove za tekst, ako je tekst bio uređivan u ugrađenom uređivaču. To mora uključivati stilove za font u koje se ubraja promjena stila i veličine fonta, kurziv, podcrtani i podebljani tekst te poravnavanje teksta. Aplikacija treba korisniku ponuditi opciju spremanja uređivanih zapisa lokalnu u bazu podataka. Spremljene zapise korisnik mora moći kasnije uređivati, brisati i dijeliti. Baza podataka mora se obrisati deinstalacijom aplikacije.

2. Nefunkcionalni zahtjevi

Nefunkcionalni zahtjevi koje aplikacija mora ispuniti su:

- Mobilna aplikacija mora biti brza i responzivna
- Korisničko sučelje mora biti intuitivno za korištenje
- Korisničko sučelje aplikacije mora biti prilagodljivo uređajima različitih veličina i rezolucija ekrana (mobitel, tablet)
- Aplikacija mora biti podržana na uređajima s instaliranim operacijskim sustavom Android 5.1 (Lollipop) i novijim verzijama operacijskog sustava Android

3. Korištene tehnologije i alati

3.1. Osnovne tehnologije i alati

Aplikacija je pisana u programskim jezicima Kotlin [3] i Java. Java je dugo vremena bila glavni jezik za razvoj Android aplikacija sve dok se nije pojavio programski jezika Kotlin koji je interoperabilan s Javom jer se oba jezika kompajliraju u isti Java bytecode. Budući da se Kotlin zbog svoje jednostavnije sintakse, boljom podrškom za funkcijsko programiranje i brojnih drugih razlika u odnosu na Javu sve više preporučuje za razvoj modernih Android aplikacija, većina programskog koda aplikacije napisana je u Kotlinu.

Za razvoj i testiranje aplikacije korišteno je razvojno okruženje Android Studio. Službena Googleova stranica za razvoj Android aplikacija [1] sadrži detaljne upute za korištenje razvojnog okruženja kao i brojnu dokumentaciju za razvoj. Aplikacija je testirana na virtualnim uređajima unutar razvojnog okruženja te na fizičkom uređaju s instaliranim operacijskim sustavom Android.

Aplikacija je razvijena koristeći Android SDK (software development kit) komplet alata koji sadrži brojne biblioteke, emulator, vodiče, predloške koda i brojne druge funkcionalnosti za razvoj aplikacije, Android SDK je dio razvojnog okruženja Android Studio.

3.2. Vanjske biblioteke

Ključne vanjske biblioteke korištene u aplikaciji:

- Room biblioteka korištena za kreiranje i upravljanjem lokalnom bazom podataka SQLite
- UCrop biblioteka [5] korištena za funkcionalnost izrezivanja fotografija
- iText biblioteka [6] korištena za generiranje PDF dokumenata
- ML kit text recognition biblioteka [2] korištena za OCR odnosno za prepoznavanje teksta iz fotografije

4. Arhitektura sustava

Arhitektura aplikacije temelji se na konceptu MVVM (Model–View–ViewModel) [4]. Arhitekturalni obrazac MVVM izrazito je popularan obrazac u razvoju mobilnih aplikacija. MVVM sastoji se od tri glavne komponente:

- View komponenta predstavlja korisničko sučelje (UI) aplikacije. Glavna gradivna komponenta Android aplikacije je aktivnost. Aktivnost predstavlja jedan zaslon korisničkog sučelja koji reagira na unose korisnika. Aktivnost ima svoj životni ciklus, ona se pokreće, pauzira i uništava. Android sustav upravlja životnim ciklusom aktivnosti putem metoda kao što su onCreate(), onPause(), onDestroy(), itd. Fragmenti predstavljaju modularne komponente korisničkog sučelja unutar neke aktivnosti. Korištenjem fragmenata korisničko sučelje se razdvaja na manje nezavisne i ponovno upotrebljive komponente koje imaju vlastiti životni ciklus. U modernom razvoju manjih mobilnih aplikacija preporučuje se korištenje jedne aktivnosti koja upravlja s više fragmenata, ova mobilna aplikacija slijedi taj princip.
- Model komponenta predstavlja sloj aplikacije zadužen za poslovnu logiku i za upravljanje podacima. Ovaj sloj nije u doticaju s View-om odnosno korisničkim sučeljem aplikacije. U okviru ove mobilne aplikacije model bi mogao uključivati klasu History koja definira entitet baze podataka te klasu HistoryRepo koja definira CRUD (create, read, update, delete) operacije nad bazom podataka.
- ViewModel komponenta predstavlja posrednički sloj između View-a i Modela. Ona dohvaća podatke iz modela i priprema ih za prikaz na korisničkom sučelju. Prilikom ažuriranja podataka Model obavještava ViewModel o promijeni koji dalje šalje informacije korisničkom sučelju. Glavna prednost ViewModela je ta što preživljava promjene konfiguracije aplikacije. Kada se u aplikaciji desi promjena konfiguracije poput rotacije zaslona, instance fragmenta i aktivnosti se uništavaju i ponovno stvaraju, no ViewModel zadržava svoje stanje što smanjuje potrebu za implementacijom složenih rješenja poput spremanja i obnavljanja stanja sustava.

4.1. Opis i model baze podataka

Aplikacija koristi SQlLite bazu podataka. Room je biblioteka koja olakšava upravljanje bazama podataka u Android aplikacijama. Definiranjem klasa entiteta i DAO (Data access object) sučelja u kodu, room biblioteka generira SQLite kod koji definira potrebne tablice i sheme u bazi podataka. Room baza podataka je lokalna, što znači da se briše prilikom deinstalacije aplikacije i idealna je za aplikacije koje ne trebaju pohranjivati velike količine podataka i čije baze podataka nisu kompleksne. Prednost korištenja lokalne baze podataka je da aplikacija ne treba internetsku vezu za spremanje i dohvaćanje zapisa.

Entitet „History " (Tablica 4.1) sadrži spremljene tekstualne zapise korisnika.

Tablica 4.1 entitet „History"

Naziv atributa	Tip	Opis
id (PK)	INTEGER	Jedinstveni identifikator spremljenog zapisa
title	TEXT	Naslov spremljenog zapisa
text	TEXT	Sadržaj spremljenog zapisa
timestamp	INTEGER	Vrijeme zadnje promjene zapisa
stylingMap	TEXT	JSON objekt koji predstavlja primijenjene promjene u uređivaču, npr. promjena stila fonta

4.2. Implementacija programskog rješenja

4.2.1. Opća organizacija klasa i upravljanje bazom podataka

Aplikacija je implementirana kao jedna aktivnost, klasa MainActivity koja služi kao centralna točka za navigaciju različitim dijelovima aplikacije. Aktivnost upravlja

fragmentima koji su implementirani na način da svaki fragment obavlja određen skup funkcionalnosti.

Aplikacija ima definirana 3 glavna fragmenta:

- MainFragment je fragment koji predstavlja glavni dio aplikacije koji služi za učitavanje, pretprocesiranje i izrezivanje fotografija
- EditorFragment je fragment koji predstavlja dio aplikacije koja sadrži tekstualni uređivač, u njemu se implementirane funkcionalnosti uređivanja, spremanja, dijeljenja i izvoza teksta u PDF formatu.
- HistoryFragment je fragment koji sadrži zaslon za upravljanje spremljenim tekstualnim zapisima u lokalnoj bazi. Nudi funkcionalnost učitavanja, preimenovanja i brisanja spremljenih zapisa.

Za definiranje izgleda korisničkog sučelja aplikacija koristi XML (Extensible Markup Language) datoteke. One definiraju poziciju elemenata unutar zaslona, stil elemenata, pozadinu, izgled gumbova, itd. Ovo je slično načinu na koji HTML datoteke definiraju izgled korisničkog sučelja web aplikacija no android XML datoteke koriste različite oznake od HTML datoteka, na primjer oznaka "p" za paragraf u HTML dokumentu bi se u Android XML datoteci označila s "TextView". XML datoteka koja definira izgled glavne aktivnosti prikazana je na slici (Sl. 4.1) . Element "Toolbar" definira alatnu traku na vrhu zaslona. Element "fragment" služi kao spremnik prilikom mijenjanja fragmenata, odemo li na tekstualni uređivač, ovaj element će sadržavati XML dokument vezan uz klasu EditorFragment. NavigationView element služi za navigaciju unutar aplikacije.

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.drawerlayout.widget.DrawerLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">

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

        <fragment
            android:id="@+id/fragment"
            android:name="androidx.navigation.fragment.NavHostFragment"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            app:defaultNavHost="true"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@id/toolbar"
            app:navGraph="@navigation/app_nav" />

    </androidx.constraintlayout.widget.ConstraintLayout>

    <com.google.android.material.navigation.NavigationView...>

</androidx.drawerlayout.widget.DrawerLayout>
```

Sl. 4.1 Datoteka activity_main.xml

Klasa `History` definira tablicu u bazi podataka (Sl. 4.2)

```kotlin
@Entity(tableName = "history_table")  Tomislav Grudic
data class History(
    @PrimaryKey(autoGenerate = true)
    val id: Int,
    val title: String,
    val text: String,
    val timestamp: Long,
    val stylingMap: MutableMap<Pair<Int, Int>, HtmlElementDto>
)
```

Sl. 4.2 Klasa entiteta `History`

Budući da klasa entiteta `History` sadrži varijablu `stylingMap` kompleksnog tipa prilikom spremanja i dohvaćanja baza koristi se klasa `StylingMapTypeConverter` (Sl. 4.3) za pretvaranje mape u znakovni niz koji može spremiti u SQLite bazu podataka. Ista se klasa koristi i prilikom dohvaćanja podataka iz baze za ponovnu pretvorbu u mapu.

```
class StylingMapTypeConverter {
    @TypeConverter
    fun mapToJson(map: MutableMap<Pair<Int, Int>, HtmlElementDto>): String {
        val gson = Gson()
        val mapWithStringKeys = map.mapKeys { key -> "${key.key.first}-${key.key.second}" }
        return gson.toJson(mapWithStringKeys)
    }
    @TypeConverter
    fun jsonToMap(jsonMap: String): MutableMap<Pair<Int, Int>, HtmlElementDto> {
        val gson = Gson()
        val type = object : TypeToken<Map<String, HtmlElementDto>>() {}.type
        val map = gson.fromJson<Map<String, HtmlElementDto>>(jsonMap, type)
        return map.mapKeys {
            val parts = it.key.split( ...delimiters: "-")
            Pair(parts[0].toInt(), parts[1].toInt())
        }.toMutableMap()
    }
}
```

Sl. 4.3 Kod klase `SytlingMapConverter`

Klasa `HistoryDao` (Sl. 4.4) definira metode za upite nad bazom podataka. Anotacije poput `@Update`, `@Delete` i `@Insert` uklanjaju potrebu za ručnim pisanjem upita.

```
@Dao
interface HistoryDao {
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    fun addRecord(history: History)

    @Query("SELECT * FROM history_table ORDER BY id ASC")
    fun readAllData(): LiveData<List<History>>

    @Update
    fun updateRecord(history: History)

    @Delete
    fun deleteRecord(history: History)

    @Query("SELECT * FROM history_table WHERE history_table.id = :id")
    fun findById(id: Int): LiveData<History>
}
```

Sl. 4.4 Kod klase `HistoryDao`

Apstraktna klasa `HistoryDatabase` (Sl. 4.5) upravlja stvaranjem baze podataka. Ona nasljeđuje klasu `RoomDatabase`. Ona prati singleton obrazac odnosno osigurava da postoji samo jedna instanca baze u cijeloj aplikaciji što koristi manje resursa sustava. Definiranje apstraktne funkcije koja vraća `HistoryDao` omogućava Room biblioteci da prilikom pokretanja generira implementaciju sučelja koje će služiti za upravljanje bazom podataka.

```kotlin
@Database(entities = [History::class], version = 1, exportSchema = false)
@TypeConverters(StylingMapTypeConverter::class)
abstract class HistoryDatabase : RoomDatabase() {
    abstract fun historyDao(): HistoryDao

    companion object {
        @Volatile
        private var INSTANCE: HistoryDatabase? = null

        fun getDatabase(context: Context): HistoryDatabase {
            val tempInstance = INSTANCE
            if (tempInstance != null) {
                return tempInstance
            }
            synchronized( lock: this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    HistoryDatabase::class.java,
                     name: "history_database"
                ).build()
                INSTANCE = instance
                return instance
            }
        }
    }
}
```

Sl. 4.5 Kod klase `HistoryDatabase`

Klasa `HistoryRepo` (Sl. 4.6) služi kao posrednik između `HistoryViewModel` klase i `HistoryDao` klase. U konstruktoru prima referencu na `HistoryDao` objekt. Varijabla `readAllData` koristi se za čitanje svih zapisa u bazi podataka. Tip podataka `LiveData` omogućava praćenje promjena podataka u bazi u stvarnom vremenu. Klase korisničkog sučelja poput aktivnosti i fragmenta mogu promatrati ovu varijablu i mijenjati korisničko sučelje po potrebi svaki put kada se desi promjena u bazi podataka.

```kotlin
class HistoryRepo(private val historyDao: HistoryDao) {
    val readAllData: LiveData<List<History>> = historyDao.readAllData()

    suspend fun addRecord(history: History) {
        historyDao.addRecord(history)
    }

    suspend fun updateRecord(history: History) {
        historyDao.updateRecord(history)
    }

    suspend fun deleteRecord(history: History) {
        historyDao.deleteRecord(history)
    }

    suspend fun findRecordById(id: Int): LiveData<History> {
       return historyDao.findById(id)
    }
}
```

Sl. 4.6 Kod klase `HistoryRepo`

Klasa `HistoryViewModel` (Sl. 4.7) uzima instancu baze podataka te izvršava metode klase `HistoryRepo` koje izvršavaju upite nad bazom podataka. Klasa dodatno definira da se operacija nad bazom podataka izvršavaju u pozadini te time ne blokiraju glavnu dretvu aplikacije što poboljšava optimizaciju. Svaka klasa koja komunicira s bazom podataka to radi putem instance klase `HistoryViewModel`.

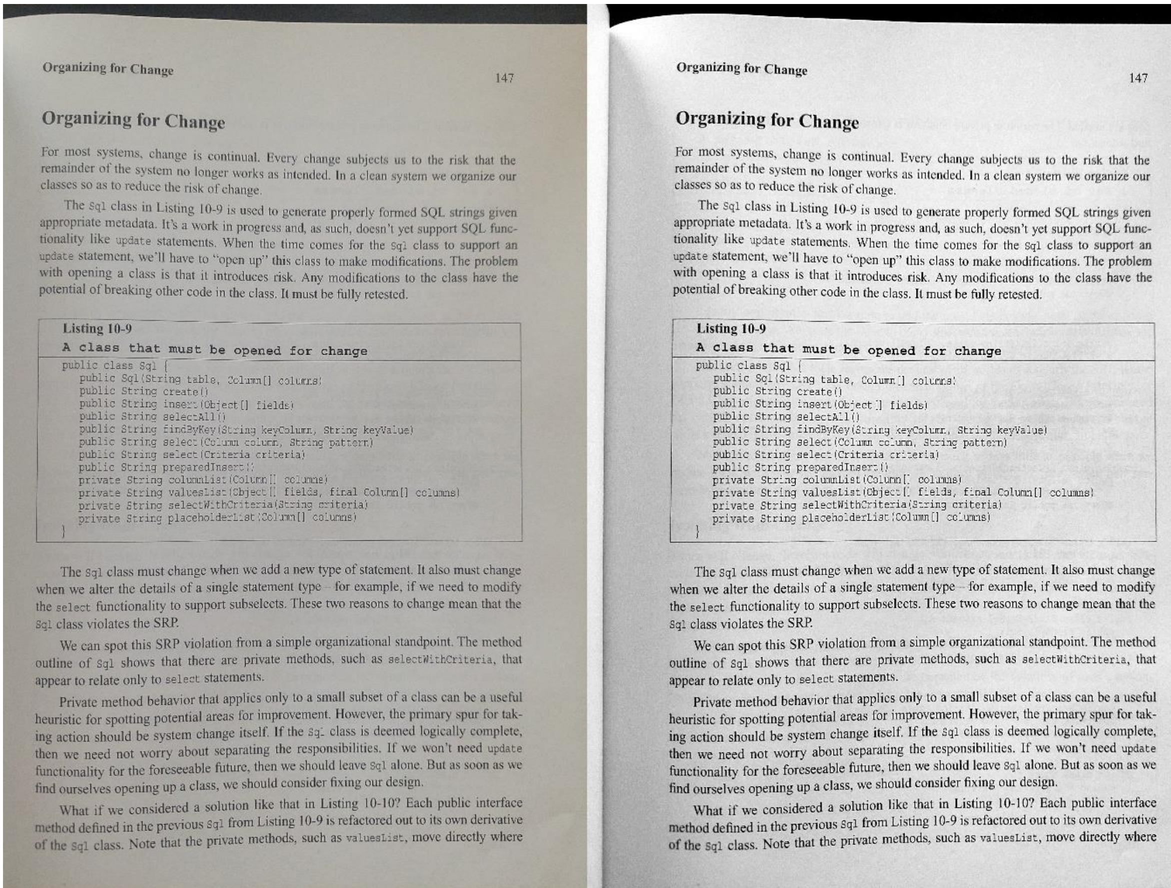
```kotlin
class HistoryViewModel(application: Application) : AndroidViewModel(application) {
    val readAllData: LiveData<List<History>>
    private val repository: HistoryRepo

    init {
        val historyDao = HistoryDatabase.getDatabase(application).historyDao()
        repository = HistoryRepo(historyDao)
        readAllData = repository.readAllData
    }

    fun addRecord(history: History) {
        viewModelScope.launch(Dispatchers.IO) {
            repository.addRecord(history)
        }
    }

    fun updateRecord(history: History) {
        viewModelScope.launch(Dispatchers.IO) {
            repository.updateRecord(history)
        }
    }

    fun deleteRecord(history: History) {
        viewModelScope.launch(Dispatchers.IO) {
            repository.deleteRecord(history)
        }
    }
}
```

Sl. 4.7 Kod klase HistoryViewModel

4.2.2. Pretprocesiranje fotografija

Da bi se poboljšala kvaliteta prepoznavanja teksta iz fotografije, prije same obrade zovu se metode koje pretvaraju fotografiju u crno-bijeli format i povećavaju kontrast u fotografiji. Time tekst u fotografiji postaje bolje vidljiv što omogućava točnije prepoznavanje teksta iz fotografije.

Na slici (Sl. 4.8) prikazan je rezultat pretprocesiranja fotografije, s lijeve strane se nalazi fotografija prije pretprocesiranja, a s desne strane fotografija nakon pretprocesiranja.



Organizing for Change 147

Organizing for Change

For most systems, change is continual. Every change subjects us to the risk that the remainder of the system no longer works as intended. In a clean system we organize our classes so as to reduce the risk of change.

The `Sql` class in Listing 10-9 is used to generate properly formed SQL strings given appropriate metadata. It's a work in progress and, as such, doesn't yet support SQL functionality like `update` statements. When the time comes for the `Sql` class to support an `update` statement, we'll have to "open up" this class to make modifications. The problem with opening a class is that it introduces risk. Any modifications to the class have the potential of breaking other code in the class. It must be fully retested.

Listing 10-9
A class that must be opened for change

```
public class Sql {
   public Sql(String table, Column[] columns)
   public String create()
   public String insert(Object[] fields)
   public String selectAll()
   public String findByKey(String keyColumn, String keyValue)
   public String select(Column column, String pattern)
   public String select(Criteria criteria)
   public String preparedInsert()
   private String columnList(Column[] columns)
   private String valuesList(Object[] fields, final Column[] columns)
   private String selectWithCriteria(String criteria)
   private String placeholderList(Column[] columns)
}
```

The `Sql` class must change when we add a new type of statement. It also must change when we alter the details of a single statement type – for example, if we need to modify the `select` functionality to support subselects. These two reasons to change mean that the `Sql` class violates the SRP.

We can spot this SRP violation from a simple organizational standpoint. The method outline of `Sql` shows that there are private methods, such as `selectWithCriteria`, that appear to relate only to `select` statements.

Private method behavior that applies only to a small subset of a class can be a useful heuristic for spotting potential areas for improvement. However, the primary spur for taking action should be system change itself. If the `Sql` class is deemed logically complete, then we need not worry about separating the responsibilities. If we won't need `update` functionality for the foreseeable future, then we should leave `Sql` alone. But as soon as we find ourselves opening up a class, we should consider fixing our design.

What if we considered a solution like that in Listing 10-10? Each public interface method defined in the previous `Sql` from Listing 10-9 is refactored out to its own derivative of the `Sql` class. Note that the private methods, such as `valuesList`, move directly where

Sl. 4.8 Rezultat pretprocesiranja fotografija

Za implementaciju ove funkcionalnosti glavne korištene klase su ugrađene Android klase `Bitmap`, `Canvas`, `ColorMatrix`, `ColorMatrixFilter` i `Paint`. Klasa `Bitmap` predstavlja fotografiju, ona sadrži podatke o svakom pikselu fotografije i informacije o boji. Klase `Canvas` i `Paint` služe za „crtanje“ u `Bitmap` objekt. Ključnu ulogu ima klasa `ColorMatrix` koja definira promjene koje će se učiniti nad svakim pikselom. `ColorMatrixFilter` klasa stvara filter koji će biti primijenjen prilikom crtanja.

Svaki piksel sadrži 4 komponente za njegov opis (RGBA), gdje je svaka reprezentirana brojem od 0 do 255, one su:

- R – komponenta koja predstavlja izražaj crvene boje
- G – komponenta koja predstavlja izražaj zelene boje
- B – komponenta koja predstavlja izražaj plave boje
- A – komponenta koja predstavlja transparentnost piksela

Na slici (Sl 4.9) prikazan je isječak koda implementacija promjene kontrasta fotografije.

```
val processedBitmap = Bitmap.createBitmap(bitmap.width, bitmap.height, bitmap.config)
val canvas= Canvas(processedBitmap)
val paint = Paint()
val colorMatrix = ColorMatrix(
    floatArrayOf(
        contrast, 0f, 0f, 0f, brightness,
        0f, contrast, 0f, 0f, brightness,
        0f, 0f, contrast, 0f, brightness,
        0f, 0f, 0f, 1f, 0f
    )
)
val filter = ColorMatrixColorFilter(colorMatrix)
paint.colorFilter = filter
canvas.drawBitmap(bitmap,  left: 0f,  top: 0f, paint)
```

Sl 4.9 Isječak koda za promjenu kontrasta fotografije

Stvara se prazni `Bitmap` objekt, zatim se definira `Canvas` i `Paint` objekt. Matrica `colorMatrix` definira vrijednosti kojima će se množiti RGBA komponente svakog piksela. Primjerice komponenta za crvenu boju će pomnožiti s vrijednošću varijable `contrast` te će se na rezultat nadodati vrijednost varijable `brightness`, odnosno prvi redak matrice će se pomnožiti s matricom 1x5 sa stupcima R,G,B,A,1. Time će se s većom vrijednosti varijable `contrast` povećati izražajnost boje u pikselu, a manjom će se izražajnost smanjiti. Prilikom stvaranja nove fotografije svaki piksel izvorne slike se mijenja na temelju definirane matrice. Time postižemo promjenu kontrasta fotografije.

4.2.3. Implementacija tekstualnog uređivača

Tekstualni uređivač koristi grafički element EditText definiran u XML datoteci vezanoj uz `EditorFragment` klasu kao komponentu za prikaz i spremanje tekstualnog sadržaja. `SpannableStringBuilder` je tip podataka koji omogućava primjenu stilova nad dijelovima teksta koji se prikazuje u grafičkim elementima .

`SpannableStringBuilder` varijabla se inicijalizira s trenutnim tekstom u elementu. Metoda `getSpans` dohvaća listu trenutnih stilova primijenjenih na dijelove teksta u definiranom rasponu. Metoda `setSpan` primjenjuje stil na odabrani dio teksta.

Na slici (Sl. 4.10) prikazana je inicijalizacija `SpannableStringBuilder` elementa i dohvat trenutnih stilova primijenih na dio teksta koji započinje na indeksu

`editTextView.selectionStart` i završava na indeksu `editTextView.selectionEnd`.

```
val spannableStringBuilder = SpannableStringBuilder(editTextView.text)
val allSpans = spannableStringBuilder.getSpans(
    editTextView.selectionStart,
    editTextView.selectionEnd,
    Any::class.java
)
```

Sl. 4.10 Način dohvaćanja promjena nad tekstom

Na slici (Sl. 4.11) se na isti dio teksta primjenjuje stil podcrtavanja teksta. Promjena je vidljiva kada u EditText element učitamo varijablu spannableStringBuilder. Na sličan način je ostvareno primjenjivanje ostalih stilova teksta kao što su kurziv i promjena stila fonta.

```
spannableStringBuilder.setSpan(
    UnderlineSpan(),
    editTextView.selectionStart,
    editTextView.selectionEnd,
    Spannable.SPAN_EXCLUSIVE_EXCLUSIVE
)
```

Sl. 4.11 Kod za primjenu stila podcrtavanja teksta

Za funkcionalnost generiranja PDF dokumenata dohvaćaju se sve promjene učinjene nad tekstom te se iz njih generira tekst u HTML formatu. Razlog tome je što biblioteka iText generira PDF dokument iz HTML dokumenta. Da bi se stvorila reprezentacija HTML dokumenta aplikacija iterira kroz sve promjene i stvara objekt tipa

`MutableMap<Pair<Int, Int>, HtmlElementDto>`. Struktura elementa `HtmlElementDto` prikazana je na slici (Sl. 4.12).

```
@Parcelize
data class HtmlElementDto(
    var prefix: String,
    var word: String,
    var suffix: String
): Parcelable
```

Sl. 4.12 Struktura pomoćne klase HtmlElementDto

Svaki element mape reprezentira promjenu učinjenu nad tekstom u uređivaču u HTML formatu. Primjerice ako na prvu riječ u uređivaču koja je duljine četiri znakova, npr. riječ „word“ primijenimo stil kurziv, element mape će imati ključ `Pair(0,4)` i vrijednost `HtmlElementDto("<i>", "word", "</i>")`. Na taj način aplikacija pamti sve promjene i može lako generirati HTML reprezentaciju uređivanog teksta. Svaki dio teksta nad kojim je učinjena promjena stila omotava se odgovarajućom HTML oznakom koja po potrebi sadrži i inline CSS. Ovaj se objekt koristi i prilikom spremanja zapisa u bazu podataka za spremanje promjena nad tekstom, pri učitavanju se HTML oznake parsiraju i iz njih se izvlači stil koji primjenjujemo na tekst.

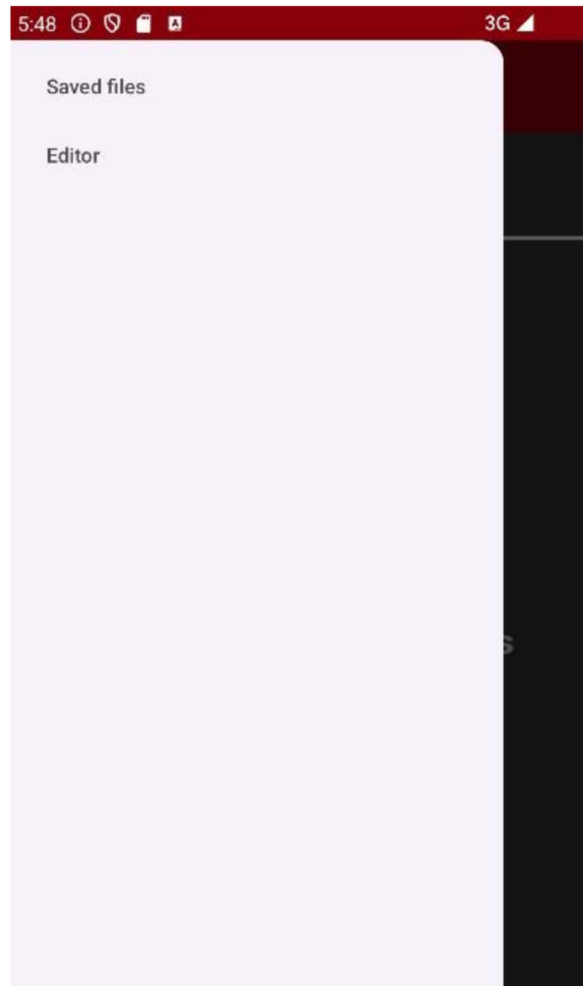
5. Upute za korištenje i instalaciju

5.1. Korisničke upute

Prilikom prvog pokretanja aplikacija traži od korisnika dopuštenje za pristup kameri, ako korisnik ne odobri dopuštenje, neće moći koristiti kameru unutar aplikacije. Aplikacija ne traži dopuštenje za pristup fotografijama spremljenim na uređaju jer time upravlja sustav. Na glavnom zaslonu aplikacije prikazanom na slici (Sl. 5.1) korisnik može učitati fotografije na dva načina. Odabirom ikone fotoaparata sustav otvara kameru uređaja i korisnik može uslikati te zatim učitati fotografiju u aplikaciju. Odabirom ikone fotografije otvara se galerija uređaja i korisnik može izabrati jednu ili više fotografija koje će biti učitane u aplikaciju. Nakon učitavanja fotografija one će biti vidljive na središnjem zaslonu. Gumb s natpisom "SCAN" pokreće aktivnost prepoznavanja teksta iz fotografija. Pritiskom na gumb s tri vodoravne crte u gornjem lijevom kutu otvaramo izbornik (Sl. 5.2) koji služi za navigaciju unutar aplikacije. Pritiskom na "Saved files" otvara se preglednik spremljenih zapisa dok se pritiskom na gumb s natpisom "Editor" otvara tekstualni uređivač.

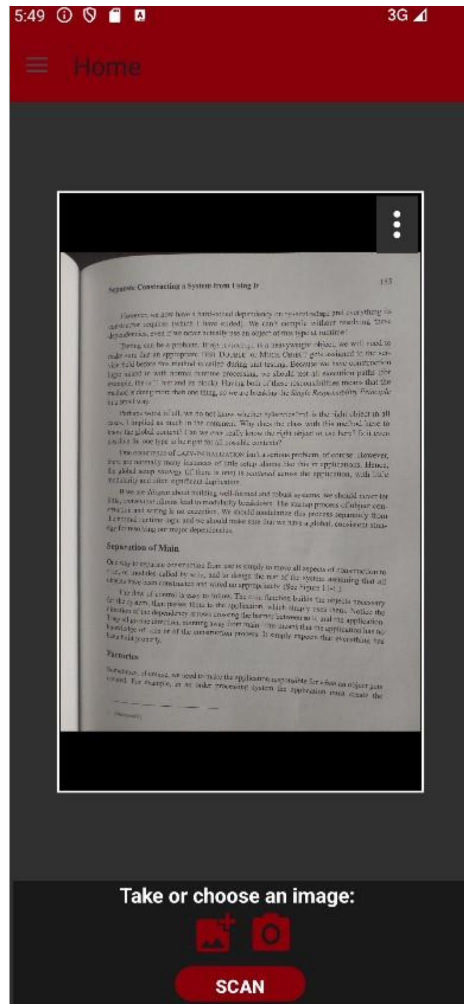
cen

Sl. 5.1 Glavni prozor aplikacije



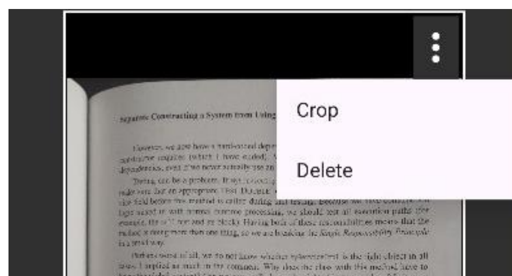
Sl. 5.2 Izbornik za navigaciju unutar aplikacije

Horizontalnim pomicanjem zaslona korisnik može vidjeti učitane fotografije (Sl. 5.3.) Aplikacija će provesti prepoznavanje teksta iz svih učitanih fotografija te sadržaj svake fotografije spojiti u jedinstven tekst. Spajanje teksta će se provesti redoslijedom kojim su fotografije prikazane na središnjem zaslonu s lijeva na desno. Ako korisnik želi promijeniti redoslijed obrađivanja fotografija to može učiniti pritiskom i držanjem fotografije na zaslonu i povlačenjem u lijevu ili desnu stranu. Time će se izvršiti animacija pomaka fotografija i promijeniti redoslijed obrađivanja fotografija odnosno redoslijed spajanja teksta iz fotografija.

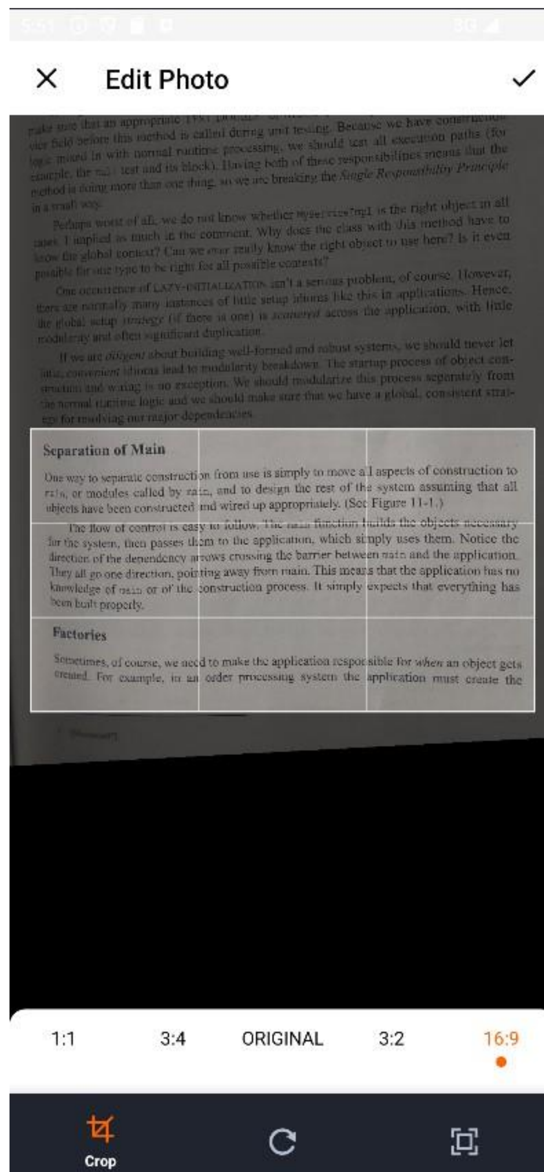


Sl. 5.3 Prikaz učitanih fotografija

U gornjem kutu svake učitane fotografije nalazi se gumb koji prikazuje dodatne opcije dostupne za svaku fotografiju (Sl. 5.4.) Pritiskom na opciju “Delete” fotografija se briše iz aplikacije i neće biti obrađena. Pritiskom na opciju “Crop” pokreće se aktivnost izrezivanja fotografije (Sl. 5.5).

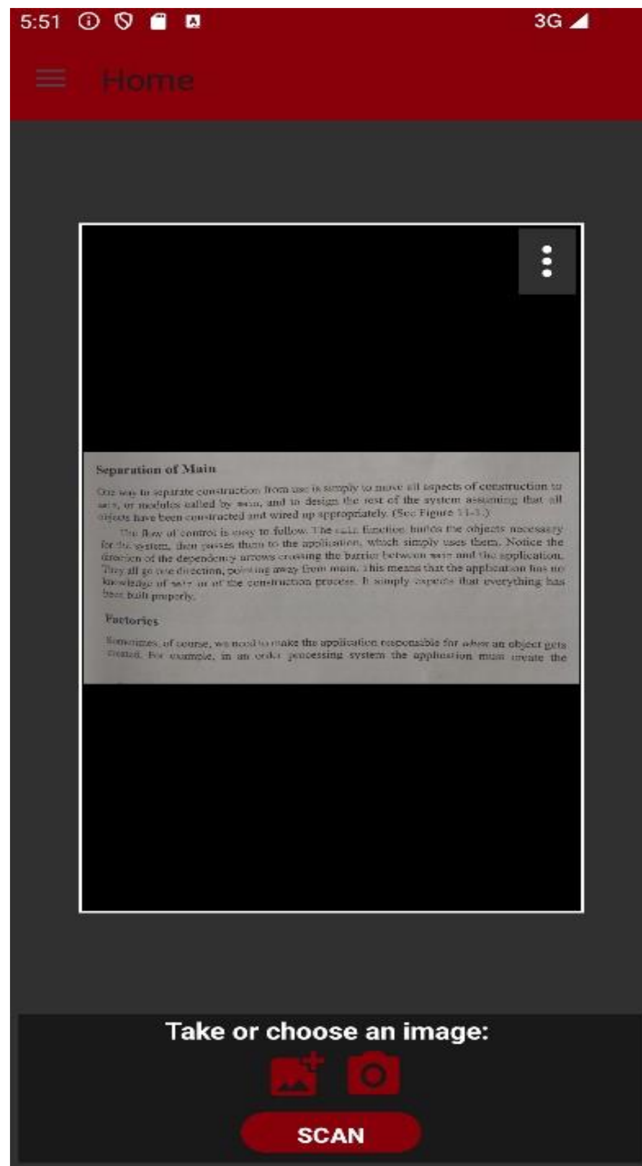


Sl. 5.4 Prikaz opcija za učitanu fotografiju



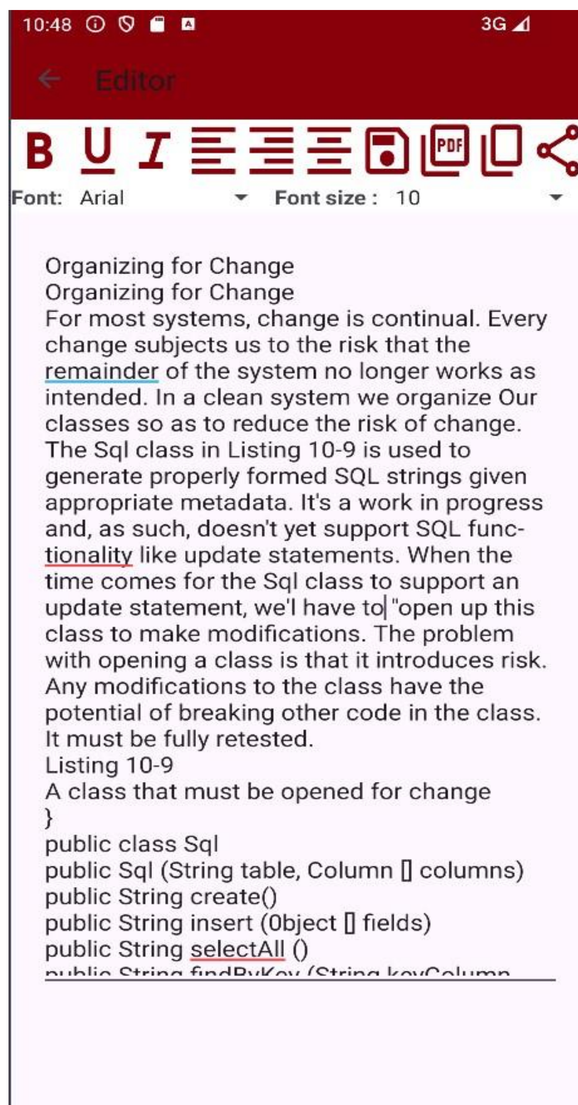
Sl. 5.5 Aktivnost za uređivanje fotografije

U aktivnosti je moguće urediti učitanu sliku da prikazuje samo željeni dio fotografije što je korisna opcija za korisnika ako ne želi da aplikacija prepozna sav tekst iz fotografije već samo dio. Sučelje je intuitivno za korištenje. Željeni dio teksta odabire se zumiranjem ili odabirom opcija na donjoj alatnoj traci. Osim opcije izrezivanja korisniku se nudi mogućnost rotiranja slike. Bitno je napomenuti da će aplikacija prepoznavati tekst onako kako je prikazan na fotografiji odozgo prema dolje zbog čega će rezultat biti neočekivan ako korisnik učita obrnuto okrenutu fotografiju. Nakon što uređivanje završi u središnji zaslon će biti učitana nova fotografija na istom mjestu gdje se nalazila izvorna fotografija (Sl. 5.6).



Sl. 5.6 Pregled uređivane fotografije

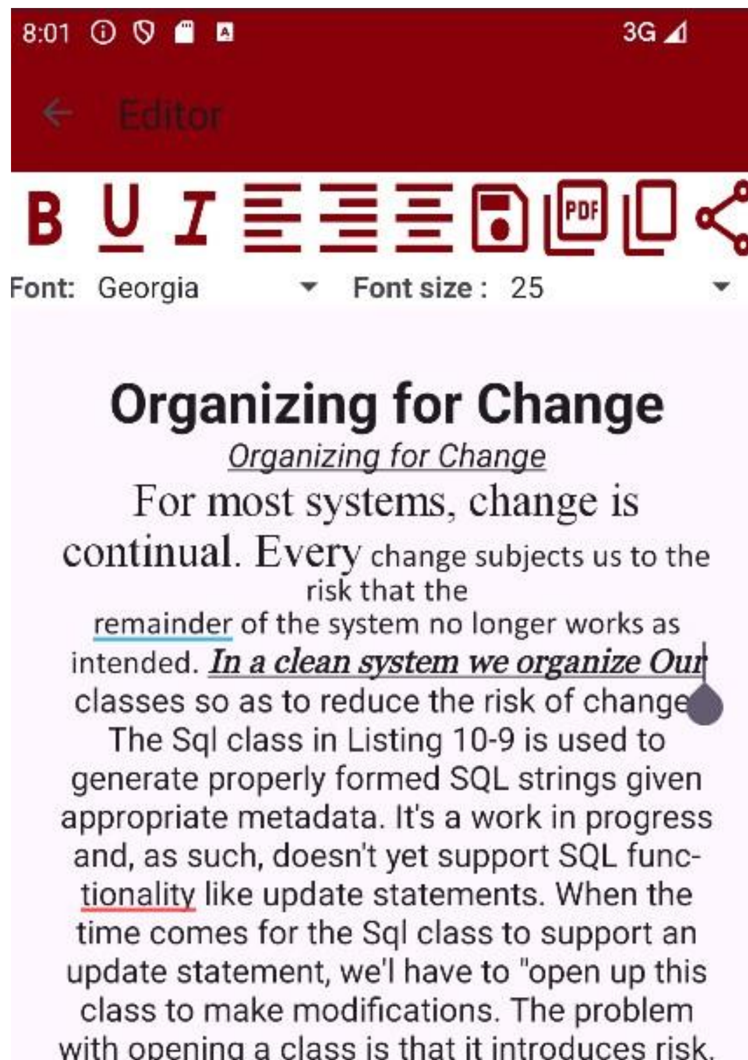
Kada korisnik učita i uredi fotografije koje želi obraditi pritiskom na gumb s natpisom "SCAN" pokreće se obrada i otvara se zaslon tekstualnog uređivača. Nakon obrade teksta otvara se tekstualni uređivač (Sl. 5.7) i u njega se učitava tekst. Prilikom učitavanja izvršava se i provjera pravopisa, riječi za koje smatra da su pogrešno napisane podcrtavaju se crvenom bojom dok se plavom bojom podcrtavaju riječi za koje sustav ima alternativne prijedloge.



Sl. 5.7 Prozor tekstualnog uređivača

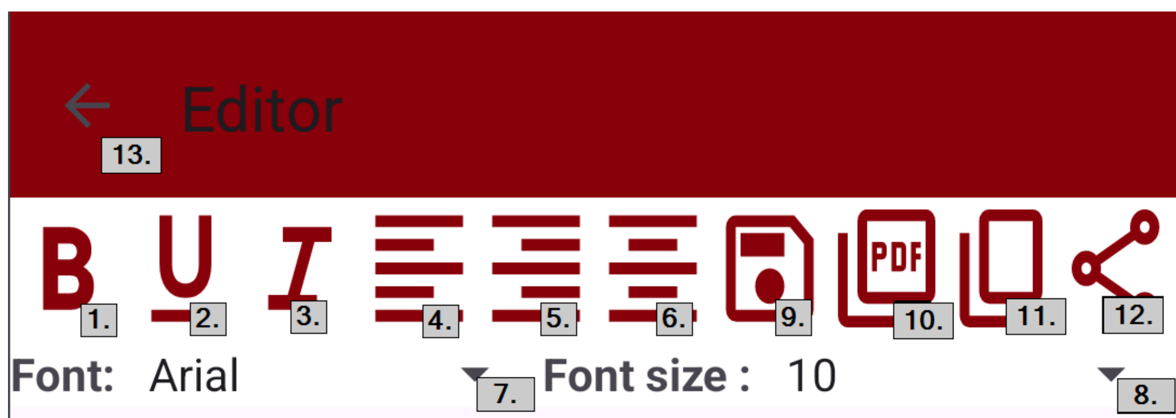
Nakon učitavanja teksta korisniku se nude opcije da uređuje učitani tekst.

Za primjenu stila potrebno je prvo odabrati tekst na koji se stil želi primijeniti. Primjer teksta nakon uređivanja prikazan je na slici (Sl. 5.8).



Sl. 5.8 Primjer uređenog teksta u tekstualnom editoru

Alatna traka na vrhu zaslona sadrži sve opcije koje se korisniku nude za učitani tekst.

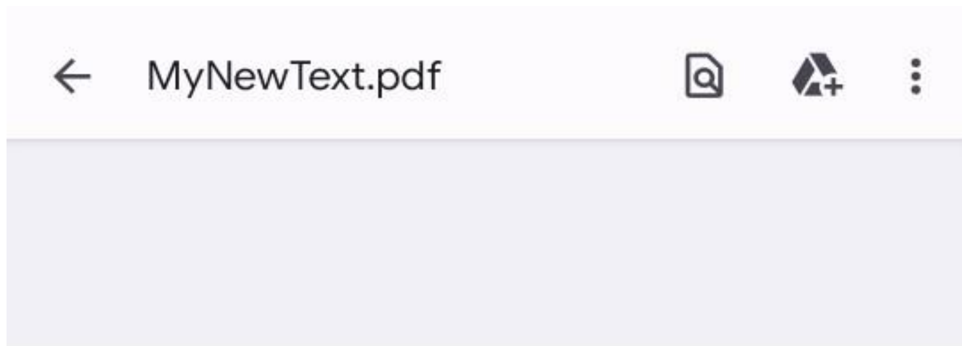


Sl. 5.9 Opcije tekstualnog uređivača

Opis i funkcija gumbova označenih na slici (Sl. 5.9)

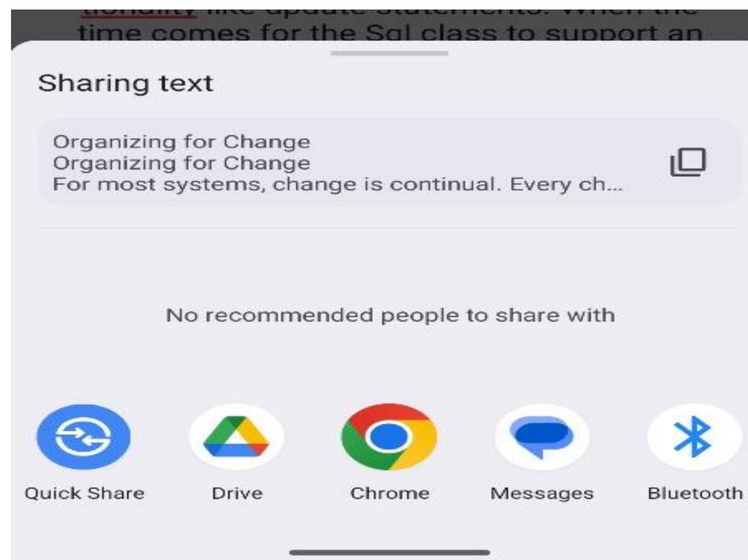
1. Podebljava odabrani tekst
2. Podcrtava odabrani tekst
3. Odabrani tekst postavlja u kurziv
4. Cjelokupni tekst poravnava u lijevo
5. Cjelokupni tekst poravnava u desno
6. Centrira cjelokupni tekst
7. Otvara padajući izbornik gdje korisnik odabire font koji želi primijeniti na odabrani tekst, dostupni fontovi su Arial, Georgia, Candara, Trebuchet MS, Times New Roman, Verdana i Calibri
8. Otvara padajući izbornik gdje korisnik odabire veličinu fonta u pikselima koju želi primijeniti na odabrani tekst
9. Sprema uređeni tekst u bazu podataka, ako tekst nije spremljen otvara se prozor za imenovanje zapisa (Sl. 5.10).
10. Pokreće se generiranje PDF dokumenta koji se zatim preuzima i otvara u zadanom pregledniku za PDF dokumente. Tekst u sadržaju generiranog dokumenta sadrži primijenjene stilove istovjetne prikazu u uređivaču (Sl. 5.11).
11. Kopira cjelokupan tekst u međuspremnik uređaja
12. Otvara se izbornik za dijeljenje cjelokupnog teksta putem aplikacija koje podržavaju dijeljenje teksta (Sl. 5.12)
13. Pritiskom se Korisnik vraća na prethodni zaslon

Sl. 5.10 Prozor za unos naslova zapisa



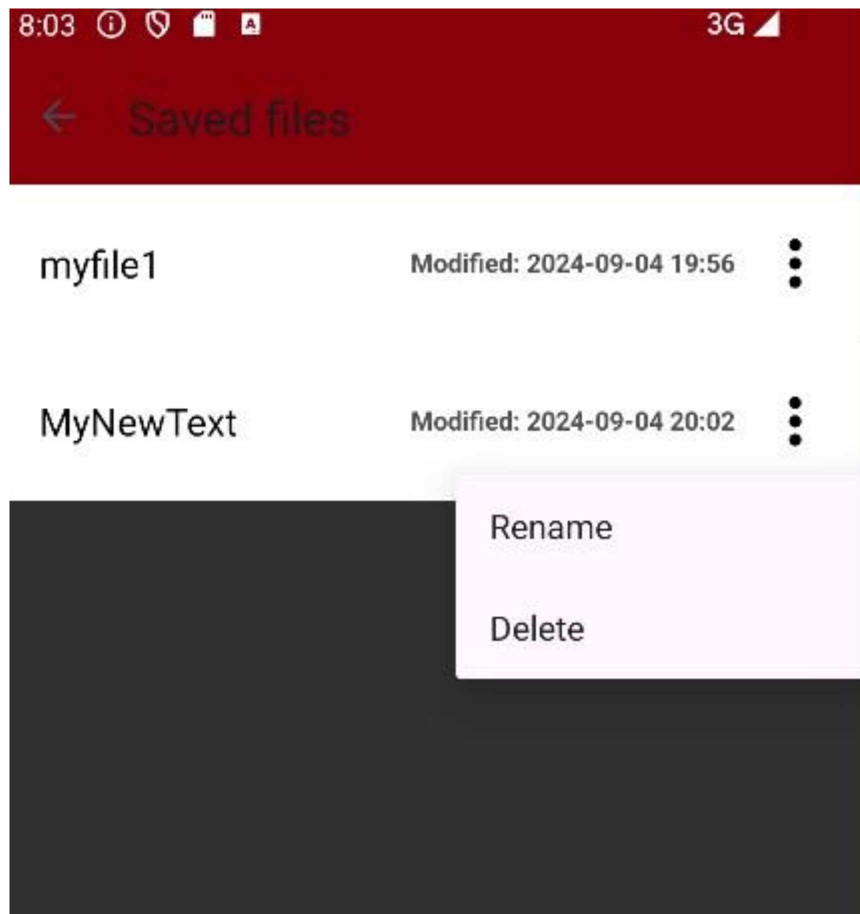
Organizing for Change *Organizing for Change* For most systems, change is continual. Every change subjects us to the risk that the remainder of the system no longer works as intended. ***In a clean system we organize Our*** classes so as to reduce the risk of change. The Sql class in Listing 10-9 is used to generate properly formed SQL strings given appropriate metadata. It's a work in progress and, as such, doesn't yet support SQL func- tionality like update statements. When the time comes for the Sql class to support an update statement, we'l have to "open up this class to make modifications. The problem with opening a class is that it introduces risk. Any modifications to the class have the potential of breaking other code in the class. It must be fully retested. Listing 10-9 A class that must be opened for change } public class Sql public Sql (String table, Column [] columns) public String create() public String insert (0bject [] fields) public String selectAll () public String findByKey (String keyColumn, String keyValue) public String select (Column column, String pattern) public String select (Criteria criteria) public String preparedInsert () private String columnList (Column[] columns) private String valuesList (0bject [] fields, final Column [] columns) 147 private String selectWithCriteria (String criteria) private String placeholderList (Column (] columns) The Sal class must change when we add a new type of statement. It also must change when we alter the details of a single statement type for example, if we need to modify the select functionality to support subselects. These two reasons to change mean that the Sql class violates the SRP. We can spot this SRP violation from a simple organizational standpoint. The method outline of sgl shows that there are private methods, such as selectWithCriteria, that appear to relate only to select statements. Private method behavior that applies only to a small subset of a class can be a useful heuristic for spotting potential areas for improvement. However, the primary spur for tak- ing action should be system change itself. If the sql class is deemed logically complete, then we need not worry about separating the responsibilities. If we won't need update functionality for the foreseeable future, then we should leave Sql alone. But as soon as we find ourselves opening up a class, we should consider fixing our design. What if we considered a solution like that in Listing 10-10? Each public interface method defined in the previous Sql from Listing 10-9 is refactored out to its own derivative of the Sal class. Note that the private methods, such as valuesList, move directly where

Sl. 5.11 Primjer generiranog PDF dokumenta



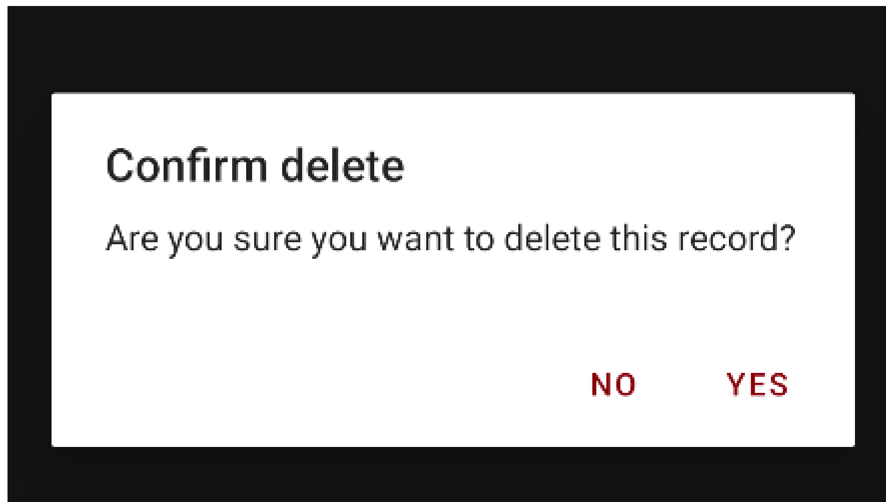
Sl. 5.12 Izbornik za dijeljenje teksta

Klikom na gumb "Saved files" u izborniku na slici (Sl. 5.2) otvara se pregled zapisa spremljenih u bazi podataka (Sl. 5.13) Svaki redak prikazuje naslov spremljenog zapisa, vrijeme zadnje promjene te gumb predstavljen s tri vertikalne točke koji otvara padajući izbornik gdje nudi opcije "Rename" i "Delete".

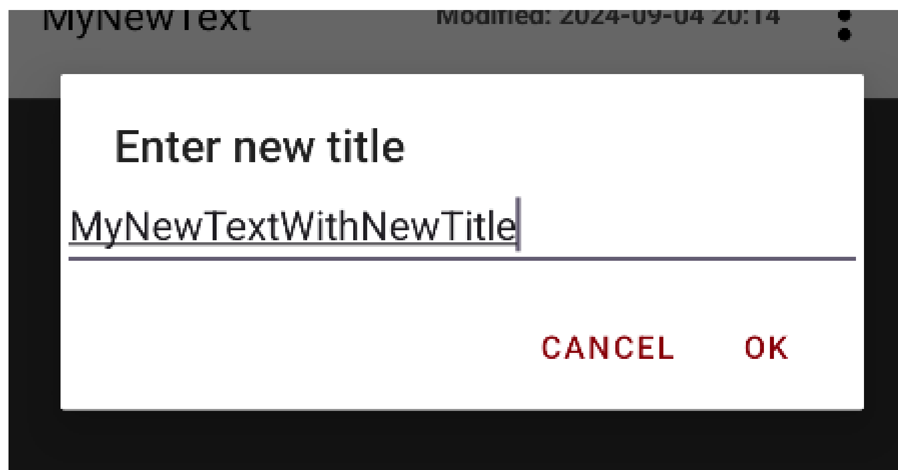


Sl. 5.13 Pregled spremljenih zapisa

Pritiskom na naslov zapisa učitava se odgovarajući zapis u tekstualnom uređivaču. Korisniku se u uređivaču nude sve prethodno opisane opcije za uređivanje, dijeljenje i izvoz teksta. Jedina razlika je ta što pritiskom na gumb za spremanje koji je označen rednim brojem devet na slici (Sl. 5.9) aplikacija korisniku prikazuje poruku sa sadržajem "*File saved!*" umjesto prozora za unos naslova zato što zapis već postoji u bazi podataka te se izvršava akcija ažuriranja baze podataka. Klikom na opciju *Delete* zapis se briše iz baze podataka ako se akcija potvrdi u iskočnom prozoru (Sl. 5.14). Klikom na opciju *Rename* otvara se prozor za preimenovanje zapisa (Sl. 5.15) te se zapis ažurira u bazi podataka.



Sl. 5.14 Prozor za potvrdu brisanja zapisa



Sl. 5.15 Prozor za unos novog naslova zapisa

5.2. Instalacija i pokretanje aplikacije

Za instalaciju i pokretanje aplikacije potrebno je preuzeti razvojno okruženje Android Studio sa službene stranice i preuzeti izvorni kod aplikacije. Prilikom instalacije Android Studija dovoljno je pratiti zadane opcije alata za instalaciju. Nakon instalacije razvojnog okruženja potrebno je pokrenuti Android Studio te u izborniku "File" izabrati opciju "Open" te odabrati datoteku koja sadrži cjelokupan kod aplikacije. Za pokretanje aplikacije na virtualnom uređaju potrebno je u izborniku s desne strane odabrati "Device manager" zatim odabrati gumb "+" i odabrati opciju "Create Virtual Device". U izborniku je potrebno odabrati uređaj, na primjer mobitel ili tablet i pritisnuti "Next". Nakon toga se otvara izbornik "System Image" gdje je najbolje odabrati neku od opcija u "Recommended" sekciji. Ako na računalu nije prisutan ni jedan system image potrebno je preuzeti neki system image, gumb se nalazi

pored imena u stupcu “Release name”. Kada se system image preuzme potrebno je pritisnuti “Next” i zatim “Finish”. Time smo konfigurirali virtualni uređaj u našem Android Studiju. Za pokretanje aplikacije potrebno je u gornjoj programskoj traci izabrati uređaj na kojem želimo aplikaciju pokrenuti te pritisnuti “Run”. Time će se aplikacija instalirati na uređaj i pokrenuti će se emulator gdje možemo koristiti aplikaciju. Za navigaciju u virtualnom uređaju koristimo miš te pomoćne gume iznad prikazanog zaslona. Za instalaciju i pokretanje aplikacije na fizičkom uređaju u postavkama uređaja je potrebno omogućiti opciju “USB debugging” ili “Wireless debugging” u “Developer options” te spojiti uređaj na računalo, odnosno na Android Studio. Fizički uređaj možemo spojiti žično ili bežično. Nakon spajanja uređaj će se pojaviti kao jedna od destinacija na kojoj možemo pokrenuti aplikaciju pritiskom na gumb "Run". Treba uzeti u obzir da će aplikacija raditi na uređajima s instaliranim operacijskim sustavom Android 5.1 (Lollipop) ili novijom verzijom.

Zaključak

U okviru ovog završnog rada implementirana je Android mobilna aplikacija za prepoznavanje teksta iz fotografije korištenjem OCR tehnologije te uređivanje, spremanje, dijeljenje i izvoz prepoznatog teksta u PDF formatu. Aplikacija je implementirana koristeći programske jezike Java i Kotlin te Android Studio kao programsko okruženje. Aplikacija koristi brojne vanjske biblioteke od kojih su najbitnije Room, iText, ML kit text recognition i UCrop. Aplikacija prati MVVM arhitekturalni obrazac i koristi lokalnu bazu podataka SQLite uz pomoć Room biblioteke.

Cilj aplikacija je olakšati pretvaranje fizičkih dokumenata i knjiga u digitalni format, te ponuditi korisnicima dodatnu mogućnost spremanja i uređivanja teksta.

Smjernice za budući razvoj i poboljšanje aplikacije:

- Učiniti aplikaciju kompatibilnom sa starijim verzijama operacijskog sustava Android
- Pratiti nove smjernice u razvoju mobilnih aplikacija koje se svakodnevno mijenjaju i prilagođavati aplikaciju nadolazećim verzijama operacijskog sustava Android
- Dodavanje korisničkih računa i spremanje sigurnosnih kopija baze podataka na cloud
- Dodatna optimizacija aplikacije
- Poboljšanje dizajna i responzivnosti korisničkog sučelja aplikacije
- Dodavanje novih opcija u tekstualni uređivač kao što su razdvajanje teksta u više stranica, numeriranje stranica, umetanje tablica i fotografija, itd.

Literatura

[1] Google, *Android Developers.* Poveznica: https://developer.android.com/studio/intro; Pristupljeno 5. rujna 2024.

[2] Google, *Google for Developers*. Poveznica: https://developers.google.com/ml-kit/vision/text-recognition/v2; Pristupljeno 5. rujna 2024.

[3] JetBrains, *Kotlin Docs*. Poveznica: https://kotlinlang.org/docs/home.html; pristupljeno 5. rujna 2024.

[4] GeeksforGeeks, *MVVM (Model View ViewModel) Architecture Pattern in Android*. Poveznica: https://www.geeksforgeeks.org/mvvm-model-view-viewmodel-architecture-pattern-in-android/; Pristupljeno 5. rujna 2024.

[5] Yalantis, *uCrop Github repository*. Poveznica: https://github.com/Yalantis/uCrop; pristupljeno 5. rujna 2024.

[6] iText Software, *uCrop Github repository*. Poveznica: https://kb.itextPDF.com/it5kb/; pristupljeno 5. rujna 2024.

Sažetak

Naslov: Aplikacija za prepoznavanje teksta iz fotografija

Sažetak: Ovaj završni rad opisuje mobilnu aplikaciju za sustav Android koja prepoznaje tekst iz fotografije te implementira tekstualni uređivač s mogućnosti izvoza teksta u PDF formatu. Definira funkcionalne i nefunkcionalne zahtjeve korisnika te opisuje bazu podataka. U radu je opisana arhitektura sustava i implementacija programskog rješenja. Zadnje poglavlje sadrži korisničke upute za korištenje aplikacije i instalaciju.

Ključne riječi: mobilna aplikacija, OCR, Android, Kotlin, baze podataka, MVVM

Summary

Title: Application for Text Recognition from Photos

Summary: This paper describes a mobile application for the Android system that recognizes text from a photo and implements a text editor with the ability to export text in PDF format. It defines functional and non-functional user requirements and describes the database. The paper also describes the system architecture and the implementation of the software solution. The final chapter contains user instructions for using the application and installation.

Keywords: mobile application, OCR, Android, Kotlin, database, MVVM