UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

ReceiptScan

-

Aplicație Android pentru digitalizarea bonurilor fiscale

Coordonator științific
prof. Ioana Leuștean

Absolvent
Lucian Boacă

București, septembrie, 2019

Cuprins

Capitolul 1

Introducere

Această lucrare prezintă **ReceiptScan**, o aplicație mobilă de scanare a bonurilor fiscale și vizualizare a cheltuielilor, disponibilă pentru platforma *Android*.

1.1 Motivație

Urmărirea și analizarea cheltuielilor este o sarcină importantă pentru alcătuirea unui buget personal și pentru o organizare mai bună a activităților financiare. Popularizarea în ultimii ani a plăților electronice și cu cardul a facilitat apariția a tot mai multe astfel de servicii automate. Majoritatea băncilor oferă astăzi o aplicație mobilă cu funcționalitatea de a urmări și clasifica tranzacțiile clienților.

Cea mai mare problemă pe care aceste servicii o întâmpină este lipsa unor date mai bogate, fără de care valoarea pe care o pot aduce este limitată. Într-adevăr, soluțiile existente valorifică un set limitat de date disponibile tranzacțiilor, printre care numele comerciantului și suma totală. Fără a avea acces la conținutul tranzacției, serviciile existente nu pot oferi o listă comprehensivă cu toate achizițiile utilizatorului.

O altă problemă a serviciilor oferite de bănci pentru urmărirea cheltuielilor este disponibilitatea datelor. Utilizatorii acces limitat la datele ce le aparțin. Aceste date sunt disponibile fie doar în aplicația băncii, fie pot fi exportate în formate ce nu pot fi valorificate mai departe, cum ar fi documente PDF. În acest caz, o întrebare simplă, cum ar fi *Cât am cheltuit în această lună pe pâine?* devine greu de răspuns.

Având în vedere dezavantajele soluțiilor curente, am dezvoltat ReceiptScan, o aplicație care să ofere o mai mare vizibilitate asupra tranzacțiilor financiare. Aceasta permite scanarea bonurilor fiscale, înțelegerea automată a acestora, prezentarea grafică și stocarea acestora într-o bază de date locală și exportul acestora în cloud, de unde pot fi descărcate pentru o analiză mai amănunțită utilizând uneltele utilizatorului.

1.2 Obiective

În procesul de dezvoltare a aplicației ReceiptScan mi-am propus:

1. **Dezvoltarea și îmbunătățirea algoritmului de extragere de informații**: Extragerea informațiilor structurate din imagini este un proces complex, ce nu poate avea o soluție standard, care să funcționeze în orice caz. În plus, lipsa unei metode formale de evaluare a sarcinii de înțelegere a conținutului chitanțelor din imagini acceptată la scară largă îngreunează dezvoltarea de noi metode. Acest proiect aplică un set de metode euristice, cu scopul de înțelege o gamă cât mai largă de bonuri fiscale populare în România și de a necesita un efort minim din partea utilizatorului;
2. **Valorificarea confidențialității utilizatorului**: Datele financiare ale utilizatorilor pot fi fructificate de către agenții de publicitate din mediul online și de aceea utilizatorii pot fi reticenți în a folosi o aplicație care are acces la acestea. Am gândit această aplicație astfe încât comunicarea cu un server să se facă doar voluntar și complet anonim, astfel încât informațiile despre achiziții să nu poată fi legate de un anumit utilizator;
3. **Implementarea unor standarde înalte ale structurii codului**: Calitatea codului și organizarea acestuia are un puternic impact asupra succesului unui proiect software pe termen mediu și lung. În cadrul acestui proiect mi-am propus explorarea bunelor practici în dezvoltarea aplicațiilor *Android* și construirea unei arhitecturi care să faciliteze testarea și decuplarea sistemului și care să fie ușor de înțeles și implementat.

1.3 Lucrări asemănătoare

Analizarea și procesarea chitanțelor financiare pe baza imaginilor obținute folosind camera foto a telefoanelor a stârnit un interes moderat în comunitatea științifică și tehnică. *Janssen et al.* prezintă în lucrarea *Receipts2Go* [3] o abordare bazată pe OCR și expresii regulate pentru a extrage datele despre tranzacții, dar nu include lista de produse. *Raoui-Outach et al.* prezintă în lucrarea *Deep Learning for automatic sale receipt understanding* [9] o abordare de procesare bazată pe *deep learning*.

1.4 Structura lucrării

Această lucrare este structurată după cum urmează. Capitolul 2 prezintă specificațiile aplicației ReceiptScanîntr-un mod semi-formal. Capitolul 3 motivează platforma aleasă și descrie tehnologiile folosite, arhitectura aplicației și detaliile de persistență. În capitolul 4 sunt prezentate detaliile de implementare și provocările tehnice întâmpinate. Lucrarea

se încheie în capitolul 5 prin prezentarea concluziilor și a observațiilor de final.

Capitolul 2

Specificațiile aplicației

În cadrul dezvoltării unui produs software, definirea specificațiilor are un rol crucial în succesul proiectului. Această etapă poate implica studii laborioase de piață, iar specificațiile pot trece prin mai multe etape, de la discuții cu utilizatorii până la documente formale ce ajung la programatori. În cazul de față, specificațiile sunt dictate de nevoile personale și pot fi formulate într-un limbaj mai apropiat de cel tehnic, gata de implementare.

Specificațiile sunt definite în jurul noțiunii de *usecase* (caz de utilizare), așa cum acestea sunt definite în lucrarea *Structuring use cases with goals* [1] și inspirate din exemplul prezentat în *The Pragmatic Programmer* [2]. Așadar, **scopul** acestor *usecases* este de a defini specificațiile, **conținutul** este proza consistentă (descriere în cuvinte astfel încât să nu apară contradicții), **pluralitatea** este de unul sau mai multe scenarii per *usecase*, iar **structura** este semi-formală.

În continuare este prezentat fiecare caz de utilizare astfel: o scurtă descriere a funcționalității însoțită de *screenshot-uri* din aplicație, acolo unde sunt necesare care este urmată de o prezentare schematică. Această schemă dă structura în care este implementată funcționalitatea și ajută la definirea testelor. Din punct de vedere al terminologiei, **mențiunile** dau detalii și cerințe non-funcționale, **principalul scenariu** descrie pașii necesari pentru realizarea scopului, **variațiile** definesc puncte ce pot fi implementate în moduri multiple, cum ar fi mai multe surse de date și care sunt relevante pentru scop, iar **extensiile** sunt scenarii secundare, ce servesc scopul principal.

2.1 Capturarea și înțelegerea imaginilor

Aceasta este principala funcționalitate a aplicației și are ca scop extragerea informațiilor despre tranzacție dintr-o imagine cu un bon fiscal. Interfața cu utilizatorul este reprezentată de un vizor pentru camera principală a dispozitivului, ce afișează în timp real si

textul detectat în imagine în spatele unor chenare. Capturarea imaginii se face prin gestul *tap* pe ecran. Funcționalitatea permite și folosirea unei imagini din galerie, dar și folosirea *flash-ului* în condițiile de iluminare slabă. Odată capturată o imagine, procesarea acesteia se face pe un *thread* secundar, în timp ce un ecran de încărcare este afișat. Figura 2.1 prezintă ecranul de scanare și chenarele de text recunoscute în imagine.

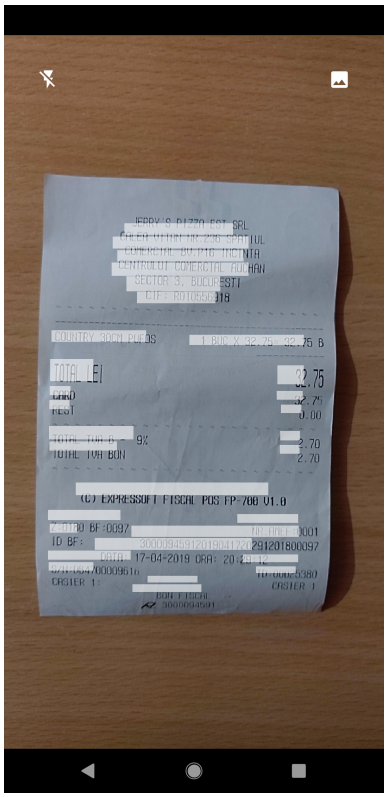


Figura 2.1: Ecranul de Scanare

- **Scop**: Capturarea unei imagini și extragerea informațiilor relevante din aceasta;
- **Condiție de succes**: Prezența unei înregistrări în baza de date ce modelează bonul fiscal în starea de *draft*;
- **Condiții de eșec**: Imaginea nu poate fi capturată; modulul OCR nu funcționează; o imagine este deja în curs de procesare;
- **Precondiții**: Valorile predefinite pentru categorie și monedă;

Mențiuni

Informațiile relevante de extras dintr-o imagine sunt:

- nume comerciant;
- data tranzacției;
- suma totală;
- moneda;
- categoria tranzacției;
- produse:
  - numele produsului;
  - prețul aferent;
- elementele OCR:
  - coordonatele casetelor de text;
  - textul aferent;

Imaginile se salvează astfel încât să nu fie accesibile din galerie.

Principalul scenariu

1. Utilizatorul capturează o imagine;
2. Modulul OCR este apelat; Textul și chenarele aferente sunt extrase;
3. Rezultatul OCR este procesat pentru a obține conținutul bonului;
4. Bonul fiscal este salvat în stadiu de draft pentru a fi editat; (Specificație 2.2)

Variații

- Imaginea poate fi capturată utilizând camera telefonului sau importată din galerie;
- Moneda și categoria pot avea valori prestabilite, ce se modifică din setări; (Specificație 2.3)

Extensii

- Pentru a ajuta utilizatorul atunci când folosește camera, procesarea imaginilor venite de la cameră se face continuu, la o rată maximă configurabilă;
- Nu pot fi procesate mai multe imagini în același timp. Starea ultimei procesări este accesibilă permanent. Dacă se primește o cerere de procesare înainte ca ultima să se fi încheiat este semnalată o eroare.

2.2 Editare draft

Întrucât extragerea informațiilor nu este un proces robust, datele extrase trebuie validate de utilizator. Odată ce imaginile sunt procesate, datele extrase sunt salvate în baza de date, sub categoria *drafts*. În acest moment, bonurile sunt editabile. Figura 4.1a prezintă ecranul unde utilizatorul vede toate drafturile, iar figura 4.1b ilustrează ecranul de editare.

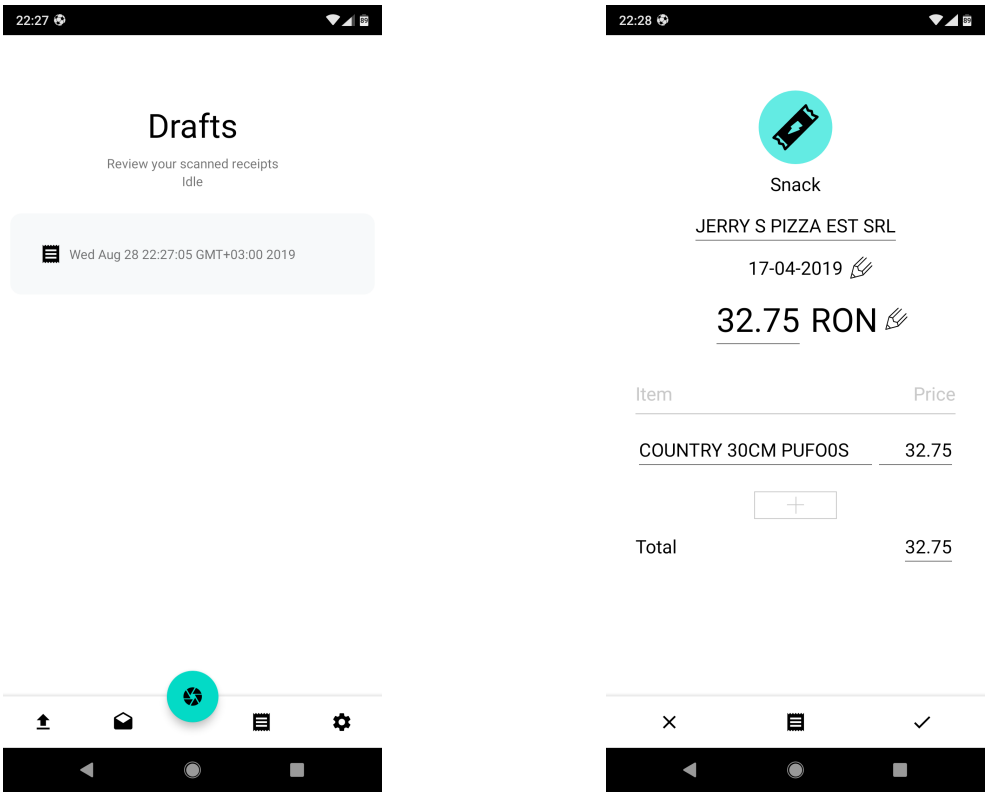
- **Scop**: Validarea informațiilor extrase din imagine de către utilizator;
- **Condiție de succes**: Modificările făcute de utilizator se reflectă în baza de date; Bonul este validat și marcat ca final;
- **Condiții de eșec**: Modificările nu pot fi persistate; Modificările sunt invalide;

Mențiuni

Algoritmul de validare poate fi subiectul unor modificări ulterioare și trebuie să fie ușor de înlocuit.

Validarea considerată la momentul scrierii presupune ca niciun câmp să nu fie null sau fără conținut.

Asupra unui draft, utilizatorul are la dispoziție următoarele opțiuni:



(a) Ecranul de listare (b) Ecranul de editare

Figura 2.2: Ecranele de gestionare a drafturilor

- modificarea categoriei, prin apăsarea pe ilustrația corespunzătoare;
- modificarea numelui comerciantului;
- modificarea datei, prin folosirea unui *date picker*;
- modificarea prețului total și a monedei;
- modificarea numelui sau prețului unui produs;
- ștergerea unui produs, prin gestul de *swipe*;
- adăugarea unui produs, prin apăsarea butonului de adăugare;
- ștergerea sau validarea *draft-ului* și vizualizarea imaginii aferente prin butoanele din bara de opțiuni;

Principalul scenariu

1. Utilizatorul accesează un bon;
2. Utilizatorul modifică câmpurile dorite;
3. Utilizatorul cere validarea bonului; Validarea se efectuează cu succes;
4. Bonul este scos din lista *drafts* și pus în lista bonurilor validate;

Variații

- Utilizatorul poate modifica valori, dar fără a valida bonul;

- Utilizatorul poate valida bonul, ceea ce îl scoate din lista de *drafts* și îl pune în lista de bonuri valide;
- Accesarea unui bon se face fie prin alegerea acestuia din listă, fie în urma scanării unei imagini; (Înțelegerea imaginilor)

Extensii

- Utilizatorul poate vedea imaginea capturată, cu și fără elementele OCR;
- Utilizatorul poate vedea toate bonurile din lista *drafts* și poate naviga către unul din ele;

2.3 Gestionare setări

- **Scop**: Modificarea și accesarea unor valori folosite în diferite puncte ale aplicației;
- **Scenariu de succes**: Modificările făcute de utilizator sunt persistate și pot fi accesate;
- **Scenarii de eșec**: Modificările nu pot fi persistate; Valorile nu pot fi accesate;

Setările considerate sunt:

- Valoarea predefinită pentru categorie;
- Valoarea predefinită pentru monedă;
- Activarea sau dezactivarea colectării anonime de date;

2.3.1 Principalul scenariu

1. Utilizatorul accesează setările
2. Utilizatorul modifică valoarea unei setări;
3. Noua valoare este persitată și accesibilă;

2.4 Colectare bonuri fiscale

- **Scop**: Utilizatorul salvează un bon, acesta este sincronizat în cloud numai dacă utilizatorul permite colectarea de date;
- **Condiție de succes**: Bonul este trimis cu succes către server;
- **Condiții de eșec**: Colectarea este permisă, utilizatorul salvează un bon, acesta nu este sincronizat în cloud; Datele nu pot fi accesate la momentul sincronizării;
- **Precondiții**: Colectarea este permisă sau nu

2.4.1 Mențiuni

Acțiunea de sincronizare se face în background, fără ca atenția utilizatorului să fie atrasă. Sincronizarea se face numai pe conexiune Wi-Fi și poate fi amânată până când conexiunea este disponibilă.

Se sincronizează toate informațiile aferente bonului, inclusiv imaginea și elementele OCR.

2.4.2 Principalul scenariu

1. Utilizatorul finalizează salvarea unui bon cu succes;
2. În consecința acțiunii de salvare, precondiția este interogată;
3. Dacă este permisă colectarea, bonul este sincronizat în cloud;

2.5 Export

- **Scop**: Accesarea datelor în afara aplicației și a dispozitivului;
- **Condiție de succes**: Utilizatorul selectează formatul, conținutul și perioada pentru export și primește un link la care poate accesa datele;
- **Condiție de eșec**: Nu există date înregistrate în perioada selectată; Datele nu sunt trimise cu succes; Utilizatorul nu primește link-ul aferent;

2.5.1 Mențiuni

Pentru a consuma cât mai puține resurse (timp, baterie), exportul se face cu minim de procesare pe dispozitiv;

Datele salvate pe cloud au o dată de expirare, după care sunt șterse;

Odată ce datele sunt încărcate și procesate în cloud, aplicația primește o notificare ce conține link-ul de descărcare;

Datele pot fi descărcate într-o arhivă zip;

2.5.2 Variații

- Pentru a oferi maximum de flexibilitate utilizatorilor, datele pot fi accesate in format JSON sau CSV și pot conține fie doar text, fie text și imagini;

2.5.3 Extensii

- În cazul lipsei de conectivitate, acțiunea de export este programată pentru o dată ulterioară, odată ce telefonul are conexiune;

- Toate sesiunile de export sunt înregistrare într-o listă și sunt eliminate odată ce datele aferente sunt șterse din cloud;

Capitolul 3

Tehnologii. Arhitectură. Persistență

Un scop secundar al acestui proiect este explorarea unor metode moderne pentru dezvoltarea aplicațiilor Android. Menținerea unor reguli și structuri clare în organizarea codului aplicației aduce o serie de beneficii, printre care reducerea numărului de bug-uri și ușurința în menținerea și extinderea aplicației pe termen lung și este crucială pentru succesul proiectelor de dimensiuni medii și mari sau la care lucrează mai multe persoane. Dezavantajul acestora este timpul ce trebuie investit la începutul proiectului și o continuă disciplină și atenție din partea programatorilor.

3.1 Alegerea platformei Android

Decizia de a dezvolta această aplicație pentru platforma Android este susținută de motive atât tehnologice, cât și de oportunitate. Conform StatCounter, în Iulie 2019, sistemul de operare Android deținea 76.08% cotă de piață la nivel global [8], 73.71% la nivelul Europei [6] și 81.3% la nivelul României [7]. Aceste cifre justifică prioritizarea platformei Android în dezvoltarea unei aplicații mobile.

Din punct de vedere tehnologic, opțiunile pentru dezvoltarea unei aplicații mobile în 2019 sunt *Android Nativ*, *iOS Nativ* sau *cross-platform*, folosind una dintre cele câteva soluții populare pentru dezvoltare cross-platform (printre care *React Native*, *Flutter* sau *Native Script*). Nevoile tehnice ale acestei aplicații presupun integrarea unei soluții OCR, iar procesarea să se facă pe dispozitiv. Implementarea unei astfel de soluții într-un framework cross-platform nu este o sarcină trivială datorită lipsei de suport și documentație. Alegând între *Android Nativ* și *iOS Nativ*, dezvoltarea pentru Android se poate face de pe orice sistem de operare major, pe când dezvoltarea pentru iOS necesită sistemul de operare macOs.

Analizând cele două motive de mai sus, am ales *Android Nativ* ca platformă de dezvoltare

datorită popularității sistemului de operare, a stabilității ecosistemului de dezvoltare și a suportului și a documentației extensive disponibile. Pentru o viitoare migrare către *iOS*, framework-ul *Flutter* este considerat ca fiind o soluție viabilă.

3.2 Tehnologii utilizate

Dezvoltarea pe platforma *Android Nativ* oferă acces la întreg ecosistemul *JVM*. Acest lucru a permis utilizarea a mai multor librării care nu au fost dezvoltate special pentru Android. În continuare vor fi prezentate tehnologiile folosite în dezvoltarea aplicației și motivația din spatele lor.

3.2.1 Kotlin

Dezvoltarea aplicațiilor Android nu mai înseamnă doar *Java*. *Kotlin* este un limbaj de programare ce rezolvă multe dintre problemele din Java și care, începând din 2017 este suportat în mod oficial de către *Google* ca limbaj de dezvoltare pentru Android, iar din 2019, considerat limbaj preferat pentru Android. Aceasta înseamnă că noile funcționalități ale SDK-ului Android vor fi dezvoltate și oferite cu prioritate către *Kotlin*.

Principalele caracteristici ale acestui limbaj sunt sistemul de tipuri superior, ce suportă inferența tipurilor, existența tipurilor de date care nu pot fi nule (*null safety*), lipsa excepțiilor *verificate* (*checked exceptions*) și diferențierea clară și ușoară între variabile și constante (prin cuvintele cheie `var` și `val`). Codul scris în *Kotlin* este de cele mai multe ori mai scurt, mai concis, mai sigur și mai ușor de înțeles decât cel scris în Java.

3.2.2 RxJava

Programare reactivă [12] este o paradigmă concentrată în jurul reacționării la modificări în starea unui obiect și a devenit populară în ultimii ani, atât pentru dezvoltarea aplicațiilor grafice, cât și pentru aplicațiile de server care procesează fluxuri de date. Avantajele acesteia sunt facilitarea procesării pe mai multe *thread-uri* și abstractizarea componentelor aplicației (*separation of concerns*).

RxJava implementează o serie de abstractizări ce extind ideea de *Observer*[4] și operatori asupra acestor abstractizări pentru a executa computații asupra valorilor reprezentate. Această aplicație folosește RxJava pentru a reprezenta fiecare operație sau unitate computațională și pentru a orchestra aceste computații pe diferite thread-uri, cu scopul de a nu bloca interfața grafică. De exemplu, surprinderea și extragerea informațiilor dintr-o poză este reprezentată folosind abstractizarea `Single` și este executată pe un thread secundar, în timp ce thread-ul principal afișează un mesaj și răspunde acțiunilor utilizatorului.

3.2.3 Android Architecture Components

Architecture Components este o colecție de librării dezvoltată de Google cu scopul de a oferi uneltele necesare pentru a dezvolta aplicații robuste și testabile. Această aplicație folosește:

- **ViewModel**: gestionează datele aferente unui ecran sau a unei colecții de ecran într-o manieră care ține cont de ciclul de viață al componentelor vizuale (*Activities*, *Fragments*). Folosite pentru a împărtăși date comune între componente vizuale și pentru a nu pierde datele în timpul schimbării configurației, cum ar fi rotirea ecranului.
- **LiveData**: expune date către componentele vizuale în mod reactiv. Această librărie se aseamănă cu RxJava, fără complexitatea aferentă. În schimb, este dependentă de ciclul de viață al componentelor vizuale, ceea ce evită problemele de tipul *memory leak*.
- **DataBinding**: este o metodă prin care datele din *ViewModel* pot fi observate în fișierele de *layout* XML.
- **Room**: este o librărie ce facilitează accesul la baza de date *sqlite* disponibilă pe dispozitiv. Se integrează cu *LiveData* și *RxJava* pentru a oferi actualizări datelor interogate.
- **WorkManager**: programează și execută activități de *background* sub anumite constrângeri, care să fie executate într-un mod eficient din punct de vedere al bateriei. Folosit pentru a colecta bonurile în cloud.

3.2.4 Firebase ML Vision

Google oferă o serie de servicii de machine learning pentru dezvoltatorii de aplicații prin intermediul *Firebase ML Kit*. Unul dintre aceste servicii este *Firebase ML Vision*, ce conține și un modul OCR. Procesarea se poate face atât local, cât și în cloud pentru o performanță sporită. Opțiunea de procesare în cloud este supusă unor tarife, dar procesarea locală este gratuită și oferă o performanță suficient de bună pentru scopul acestei aplicații. Acest serviciu a fost ales în urma unei comparații ce va fi detaliată într-un capitol ulterior.

3.2.5 Firebase Cloud Services

Firebase este o suită de servicii cloud oferită de Google dezvoltatorilor de aplicații mobile și web. Prin folosirea unor servicii cloud este eliminată complexitatea asociată dezvoltării și întreținerii unui serviciu back-end. Dintre serviciile *Firebase*, această aplicație utilizează:

- **Firestore**: o bază de date noSql ce stochează documente (obiecte JSON). Este

folosită pentru funcționalitatea de colectare a bonurilor;

- **Cloud Storage**: un sistem de foldere și fișiere. Folosit pentru a stoca imaginile asociate bonurilor și fișierele pentru export;
- **Cloud Functions**: este un serviciu computațional ce este declanșat de diferite evenimente și rulează un mediu *NodeJS* ce execută un anumit program. Este folosit pentru a arhiva fișierele exportate de aplicație și pentru a trimite o notificare cu link-ul de descărcare.

3.3 Arhitectura aplicației

Robert C. Martin definește arhitectura unui sistem software ca fiind forma care i se dă de către cei care îl construiesc. Această formă este dată de diviziunea sistemului în componente, de aranjamentul acestor componente și de modul în care aceste componente comunică între ele. Scopul acestei forme este de a facilita dezvoltarea, lansarea și întreținerea sistemului software [5].

Arhitectura dezvoltată pentru această aplicație este inspirată de cea prezentată de Robert C. Martin în cartea *Clean Architecture*, dar simplificată și adaptată pentru acest caz. Figura 3.1 prezintă nivelurile conceptuale în care este împărțită aplicația. Primele două niveluri și ultimele două niveluri sunt grupate la nivelul codului după rolul pe care acestea îl îndeplinesc în *domain* și *presentation*.

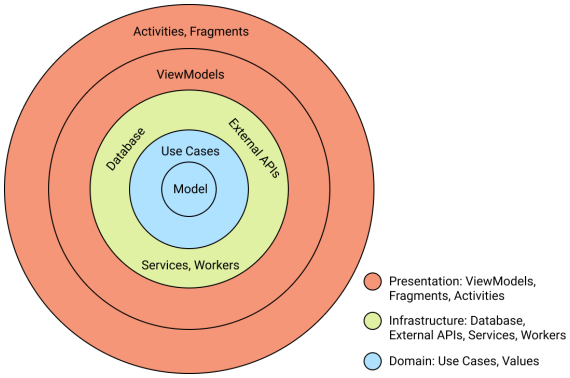


Figura 3.1: Nivelurile conceptuale ale arhitecturii aplicației

O caracteristică importantă a arhitecturii este aceea că abstractizarea descrește din centru înspre margini. Dependințele în cadrul acesteia sunt orientate către centru. Astfel, un nivel mai abstract nu depinde de un detaliu, ci invers, detaliile depind de abstractizări. Această caracteristică se realizează urmând principiul inversării dependințelor. Pentru a inversa dependințele, un nivel mai înalt definește o interfață care este implementată la un nivel inferior. Această interfață este injectată mai apoi în componenta ce necestiă un serviciu implementat la un nivel inferior. Injectarea depentințelor este exemplificată în programul **??**.

Programul 3.1: ReceiptsUseCaseImpl.kt

```
1  class ReceiptsUseCaseImpl @Inject constructor(
2      private val repository: ReceiptsRepository,
3      private val manageFactory: ManageReceiptUseCase.Factory
4  ) : ReceiptsUseCase {
5      override fun list(): Flowable<List<ReceiptListItem>> =
6          repository.listReceipts()
7
8      override fun fetch(receiptId: ReceiptId): ReceiptsUseCase.Manage {
9          return repository
10             .getReceipt(receiptId)
11             .subscribeOn(Schedulers.io())
12             .let { manageFactory.create(it) }
13     }
14 }
15
16 interface ReceiptsRepository {
17     fun listReceipts(): Flowable<List<ReceiptListItem>>
18     fun getReceipt(receiptId: ReceiptId): Flowable<Receipt>
19 }
```

La nivelul unui *usecase* este necesar accesul la baza de date. Dar la acest nivel detaliul implementării bazei de date nu este relevant. Aceasta poate fi *SQL* sau o simplă colecție în memorie. De aceea este definită interfața `ReceiptsRepository` care apoi este implementată la nivelul *infrastructure*.

Metoda recomandată pentru injectarea dependințelor în Android este librăria *Dagger 2*. Dacă majoritatea librăriilor pentru injectarea dependințelor utilizează reflexia la *run-time*, Dagger folosește anotările definite în pachetul `javax.inject` pentru a genera cod la *compile-time*. Avantajul acestei abordări este performanța sporită, dar are dezavantajul necesității de configurare din partea programatorului.

3.4 Persistență

Această aplicație folosește trei medii de persistență pentru stocarea datelor:

- **sqlite**: datele textuale aferente bonurilor;
- **internal storage**: imaginile aferente bonurilor;
- **shared preferences**: datele predefinite și alte configurații;

3.4.1 SQLITE

Aceasta este o bază de date relațională ce este preinstalată pe sistemul de operare Android. Modelul de date stocat în această bază de date este unul simplu, prezentat în Figura 3.2. Aceste tabele sunt generate cu ajutorul librăriei *Room*, folosind codul de mai jos:

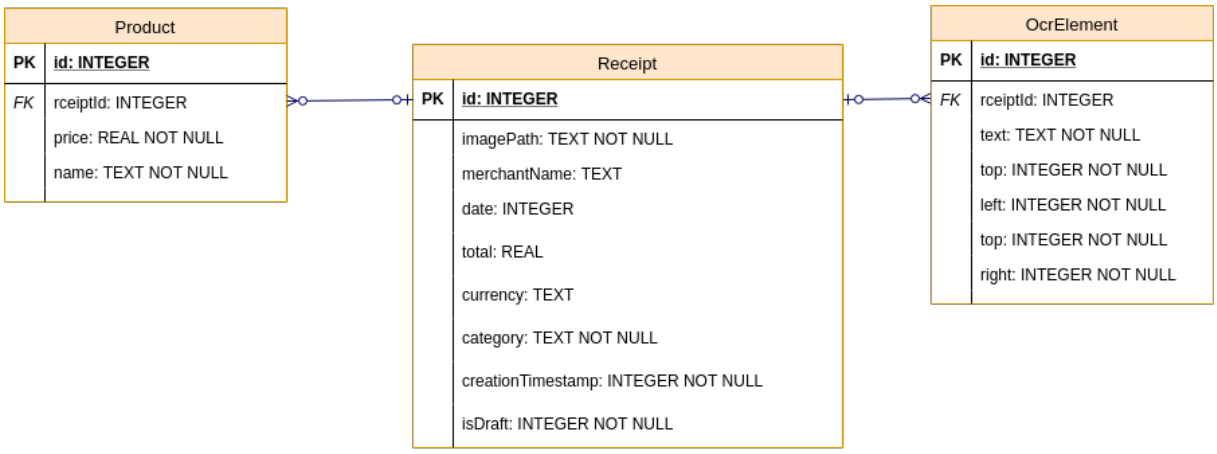


Figura 3.2: Modelul de date SQL

3.4.2 Spațiul de stocare intern

Pe spațiul de stocare intern sunt salvate imaginile aferente bonurilor, fiind inaccesibile altor aplicații. Acestea sunt salvate sub un nume aleatoriu, care este salvat în tabela sql (proprietatea imagePath din tabela Receipt).

3.4.3 Shared Preferences

Shared preferences sunt niște fișiere xml accesibile aplicațiilor Android, unde acestea pot salva valori sub format cheie-valoare. Aici sunt stocate:

- categoria predefinită pentru bonuri;
- moneda predefinită;
- permite sau nu colectarea anonimă a bonurilor;
- un id unic al aplicației, generat la instalare;

Capitolul 4

Detalii și implementare

Secțiunile precedente au prezentat specificațiile aplicației, arhitectura și principalele tehnologii care facilitează implementarea. În continuare vor fi discutate detaliile fiecărei funcționalități și modul în care acestea se reflectă în cod.

4.1 Înțelegerea imaginilor

Așa cum am precizat mai devreme, înțelegerea conținutului bonurilor fiscale din imagini reprezintă principala provocare a acestui proiect. În mod natural, această funcționalitate se desparte în două sarcini:

- Recunoașterea textului;
- Extragerea informațiilor din textul neprocesat;

Metode mai bune pentru a rezolva această provocare pot ține cont de imagini și pentru a doua sarcină și pot folosi metode mai avansate, de *machine learning*, odată cu colectarea mai multor date. De aceea am implementat funcționalitatea de colectare de date, care va fi discutată într-o secțiune viitoare. Aceste opțiuni sunt subiectul unor cercetări viitoare.

4.1.1 Recunoașterea textului

O necesitate pentru rezolvarea acestei sarcini este aceea ca procesarea să se facă pe dispozitiv. Astfel, datele utilizatorului nu părăsesc dispozitivul decât cu acordul său.

O soluție *open source* populară pentru rezolvarea problemelor OCR este *Tesseract* [11]. Pentru dezvoltatorii de aplicații mobile, Google oferă librăria *Firebase Vision*, cu suport gratuit pentru OCR pe dispozitiv. Comparația dintre cele două soluții a fost făcută astfel:

- Firebase Vision a fost rulat folosind un test de instrumentare, întrucât această

librărie nu poate rula decâ pe un dispozitiv mobil;
- Tesseract a fost rulat pe un computerul personal, folosind *Python*;
- Imaginea a fost preprocesată doar pentru Tesseract, întrucât această librărie nu oferă o performanț satisfăcătoare pe imagini neprocesate;
- Preprocesarea a constat în aplicarea unui algoritm care să elimine fundalul, să transforme imaginea î alb-negru și să uniformizeze luminozitatea;
- Asupra ambelor rezultate a fost aplicat un algoritm care să grupeze chenarele de text pe linii;

Anexa A cuprinde cele două script-uri folosite pentru comparație. Rezultatele obținute sunt prezentate în figura ... .

De menționat este și efortul necesar pentru a integra *Tesseract* într-o aplicație mobilă. În același timp, *Firebase Vision* este disponibilă ca o dependință *gradle*.

Performanța superioară a *Firebase Vision* ar fi suficientă pentru a alege această librărie. La aceasta se adaugă și ușurința integrării și lipsa necesității de preprocesare. Dezavantajul major al acestei librării este integrarea unui serviciu extern, care nu este open source în codul aplicației, dar acesta nu este unul foarte mare pentru versiunea curentă a aplicației. Așadar, pentru sarcina de OCR am ales soluția *Firebase Vision*.

4.1.2 Extragerea informațiilor din text

Procesarea textului rezultat în urma procesului de OCR se face pe baza unor reguli observate în majoritatea bonurilor fiscale. Firebase Vision returnează textul și chenarele de text, grupate în blocuri, linii și elemente, în funcție de coordonatele geometrice din imagine. Această organizare pe blocuri nu este de folos în procesarea de față, dar organizarea pe linii este, din moment ce informația de pe bonurile fiscale este așezată în format cheie-valoare, pe linii. De aceea, prima etapă în extragerea informațiilor este renunțarea la structura de blocuri și organizarea în linii raportate la întregul document. Această etapă se face după algoritmul:

1. Extrage liniile din blocuri;
2. Sortează liniile de sus în jos, în funcție de punctul lor de mijloc; Consideră liniile ca fiind elementele OCR;
3. Grupează elementele OCR după distanța relativă dintre punctele lor de mijloc: elementele la o distanță mai mică de jumătate din media înălțimii tuturor elementelor se află în același grup;

Implementarea acestui algoritm se găsește în Anexa B.

Având textul din imagine organizat în grupuri de cuvinte apropiate (vechile linii retur-

nate de *Firebase Vision*) și linii raportate la întreaga imagine, ordonate de sus în jos, informațiile relevante sunt extrase după următoarele reguli:

- **Numele comerciantului**:
  1. Extrage prima linie. Dacă aceasta este formată dintr-o singură literă, continuă extragerea. Această regulă este motivată de faptul că multe bonuri pot conține la început un logo ce poate fi confundat cu o literă.
  2. Dacă linia curentă are înălțimea peste media tuturor liniilor, atunci verifică următoarea linie. Dacă și aceasta are înălțimea peste medie și mai puțin de 3 cuvinte, consideră numele comerciantului ca fiind concatenarea celor două linii. În caz contrar, consideră numele comerciantului ca fiind textul liniei curente.
- **Data achiziției**: aplică o serie de expresii regulate pentru a parsa date din întregul text. Dacă sunt găsite mai multe date, alege data cea mai apropiată de data curentă. Dacă nu este găsită nicio dată, consideră data curentă.
- **Produse și preț total**: Acestea sunt procesate parcurgând liniile de sus în jos și alcătuind o listă obiecte de tip cheie-valoare. Cheile sunt nume de produse sau cuvinte cheie care să marcheze prețul total, iar valorile sunt prețuri, numere fracționare. Produsele și prețurile aferente sunt considerate toate obiectele care sunt întâlnite deasupra primului obiect ce marchează totalul.
- **Categoria și moneda**: aceste valori sunt citite din setările predefinite și pot fi modificate de utilizator. **Categoria și moneda**: aceste valori sunt citite din setările predefinite și pot fi modificate de utilizator. **Categoria și moneda**: aceste valori sunt citite din setările predefinite și pot fi modificate de utilizator. **Categoria și moneda**: aceste valori sunt citite din setările predefinite și pot fi modificate de utilizator.

Implementarea detaliată a algoritmului de extragere a informațiilor este prezentat în Anexa C.

4.1.3 Implementare

La nivelul domeniului, algoritmul de OCR este ascuns sub interfața `Scannable`, care este implementată la nivelul infrastructurii. Aceasta expune două metode, `ocrElements()` și `image()`, ce furnizează elementele textuale și imaginea sub abstractizarea `Observable` din RxJava.

Programul 4.1: Interfețele Scannable și ExtractUseCase

```
1 interface Scannable {
2     fun ocrElements(): Observable<OcrElements>
3     fun image(): Observable<Bitmap>
4 }
5
6 interface ExtractUseCase {
7     val state: Flowable<State>
```

```
8     val preview: Flowable<OcrElements>
9     fun fetchPreview(frame: Scannable)
10    fun extract(frame: Scannable): Single<DraftId>
11 }
12
13 sealed class State {
14    object Processing : State()
15    object Idle : State()
16    data class Error(val err: Throwable) : State()
17 }
```

`ExtractUseCase` modelează și orchestrează funcționalitățile aferente ecranului de scanare:

- Valoarea `preview` expune un flux de elemente OCR care să fie afișate pe ecran, deasupra camerei, pentru a ajuta utilizatorul în capturarea imaginii;
- Funcția `fetchPreview` permite livrarea unui nou cadru surprins de cameră, care să fie procesat asincron, iar rezultatul să fie livrat către `preview`;
- Funcția `extract` declanșează procesarea imaginii bonului și salvarea informațiilor în baza de date, returnând id-ul entității salvate;
- Valoarea `state` marchează daca o imagine este procesată pentru extragerea unui bon sau nu, sau dacă a fost întâmpinată o eroare;

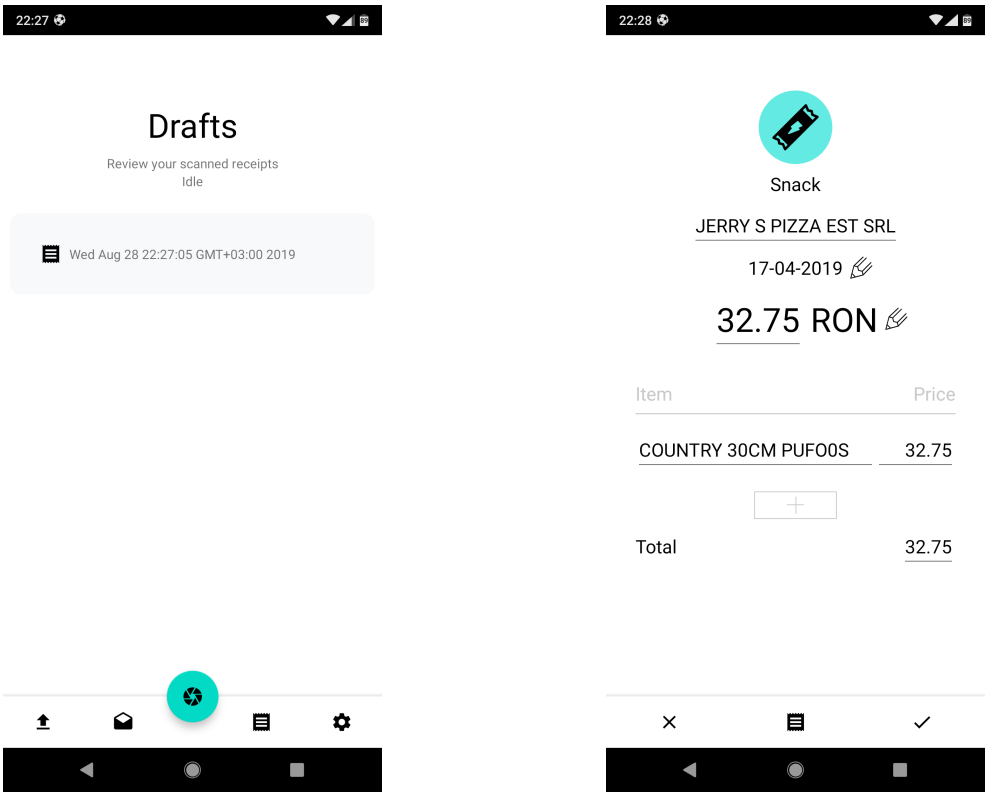
Procesarea unei imagini durează în funcție de performanțele telefonului, timp de câteva secunde. Părăsirea ecranului de scanare este permisă în acest timp deoarece obiectul `ExtractUseCase` nu este distrus odată cu obiectul vizual, ceea ce nu întrerupe procesarea.

4.2 Gestionare Drafturi

Întrucât extragerea informațiilor nu este un proces robust, datele extrase trebuie validate de utilizator. Odată ce imaginile sunt procesate, datele extrase sunt salvate în baza de date, sub categoria *drafts*. În acest moment, bonurile sunt editabile. Figura 4.1a prezintă ecranul unde utilizatorul vede toate drafturile, iar figura 4.1b ilustrează ecranul de editare.

Asupra unui draft, utilizatorul are la dispoziție următoarele opțiuni:

- modificarea categoriei, prin apăsarea pe ilustrația corespunzătoare;
- modificarea numelui comerciantului;
- modificarea datei, prin folosirea unui *date picker*;
- modificarea prețului total și a monedei;
- modificarea numelui sau prețului unui produs;
- ștergerea unui produs, prin gestul de *swipe*;
- adăugarea unui produs, prin apăsarea butonului de adăugare;
- ștergerea, validarea și vizualizarea imaginii aferente prin butoanele din bara de opțiuni;



(a) Ecranul 1 (b) Ecranul 2

Figura 4.1: Ecranele de gestionare a drafturilor

4.2.1 Implementare

La nivelul modelului, aceste opțiuni sunt reprezentate prin interfața `DraftsUseCase`. Atât funcționalitatea de listare, cât și cea de editare se folosesc de funcționalitatea *Room* prin care atunci când apare o modificare la nivelul bazei de date, o nouă valoare este emisă pentru interogările deja executate. Astfel, este ușoară o implementare reactivă pentru aceste funcționalități.

Programul 4.2: Interfața Drafts Use Case

```
interface DraftsUseCase {
    fun list(): Flowable<List<DraftListItem>>
    fun fetch(draftId: DraftId): Manage

    interface Manage {
        val value: Flowable<Draft>
        val image: Flowable<Bitmap>
        fun <T> update(newVal: T, mapper: (T, Draft) -> Draft): Completable
        fun delete(): Completable
        fun moveToValid(): Completable
        fun createProduct(): Single<Product>
        fun updateProduct(product: Product): Completable
        fun deleteProduct(product: Product): Completable
    }
}
```

Editarea câmpurilor text se face în manieră *on the fly*, ceea ce înseamnă că nu este necesar

un ecran separat drept formular și apăsarea unui buton de persistare a modificărilor. Aceasta se realizează înregistrând un *callback* pe câmpurile de text, care apelează funcția de update. Această abordare ridică problema unor fluxuri de date mult prea rapide. De aceea, asupra fluxului de modificări este aplicat operatorul RxJava `throttleLast`. Figura 4.2, din documentația RxJava [10] ilustrează modul în care acest operator funcționează.

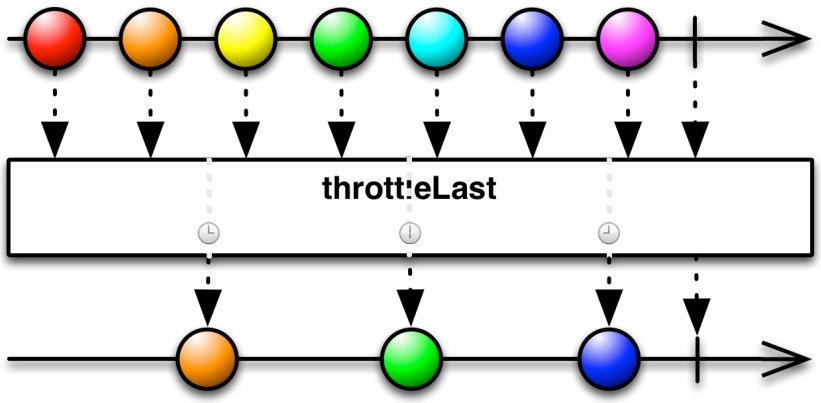


Figura 4.2: Ilustrație throttleLast

În continuare este prezentată utilizarea operatorului `throttleLast`, împreună cu un exemplu de utilizare. Funcția `throttled` primește argumentele pentru aplicarea operatorului și o funcție și returnează o nouă funcție care are aceeași signatură, același comportament, dar executată la o rată de timp specificată. În acest mod sunt exemplificate funcțiile de ordin înalt și abilitatea de a reprezenta funcțiile ca valori în limbajul Kotlin.

Programul 4.3: Funcții throttled

```
1  fun <T> throttled(
2      disposable: CompositeDisposable,
3      timeout: Long,
4      unit: TimeUnit,
5      func: ((T) -> Unit)
6  ): ((T) -> Unit) {
7      val subject = PublishSubject.create<T>()
8
9      subject
10         .throttleLast(timeout, unit)
11         .subscribe(func)
12         .addTo(disposable)
13
14     return { t -> subject.onNext(t) }
15 }
16
17 val updateMerchant =
18     throttled<String>(disposables, TIMEOUT, TIME_UNIT) {
19         useCase.update(it) { v, dwp -> dwp.copy(merchantName = v) }
20             .subscribe()
21     }
```

4.3 Setări

Ecranul de setări controlează valorile predefinite utilizate în extragerea datelor despre bonuri și indicatorul care permite sau nu colectarea datelor. Modificarea acestor valori nu este inclusă explicit în domeniul aplicației, de aceea această funcționalitate este implementată numai la nivelul prezentării și infrastructurii. Interfețele definite de model, `CollectingOption` și `ReceiptDefaults` sunt implementate de clasa `PreferencesDao`, care este folosită pentru a accesa mediul de stocare `SharedPreferences`.

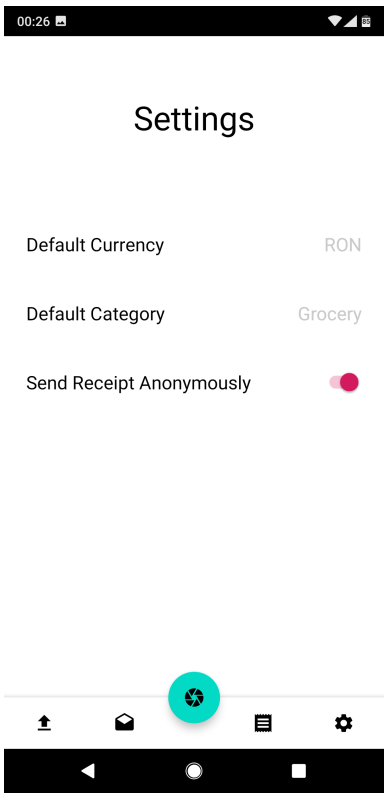


Figura 4.3: Ecranul de setări

4.4 Colectarea Datelor

Motivația colectării datelor a fost prezentată anterior. Totuși, această aplicație este bazată pe ideea de a pune utilizatorii în posesia propriilor date și de a face asta într-un mod transparent. De aceea colectarea datelor se face numai cu acordul utilizatorului și este dezactivată la instalare. În cazul în care aceasta este activată, datele sunt colectate în mod anonim. Acestea sunt trimise având un ID unic generat prima dată când funcționalitatea este folosită și nu supraviețuiește la reinstalarea aplicației.

4.5 Export

Funcționalitatea de export a datelor dă utilizatorului ocazia de a scoate datele sale din aplicație și de a le valorifica în alte moduri mai complexe. La finalul procesului de export, utilizatorul are acces la un link de descărcare a bonurilor fiscale, arhivate în format *p*. Figura 4.4 prezintă formularul pentru exportarea bonurilor. Opțiunile disponibile sunt:

- **Conținut**: doar text sau text și imagine. Exportarea atât a datelor textuale, cât și a imaginilor conduce la un consum mai mare de date, de aceea este implementată și opțiunea *doar text*. În cazul exportului imaginilor, fiecare obiect va conține un câmp cu numele imaginii aferente.
- **Format**: CSV sau JSON. Această opțiune oferă flexibilitate în disponibilitatea datelor. Formatul CSV exportă datele într-o manieră relațională, în două fișiere: *transactions.csv* și *products.csv*. Formatul JSON exportă un fișier pentru fiecare bon, ce conține datele tranzacției și o listă imbricată de produse.
- **Intervalul calendaristic**: Intervalul în care bonurile trebuie să se afle pentru a fi exportate.

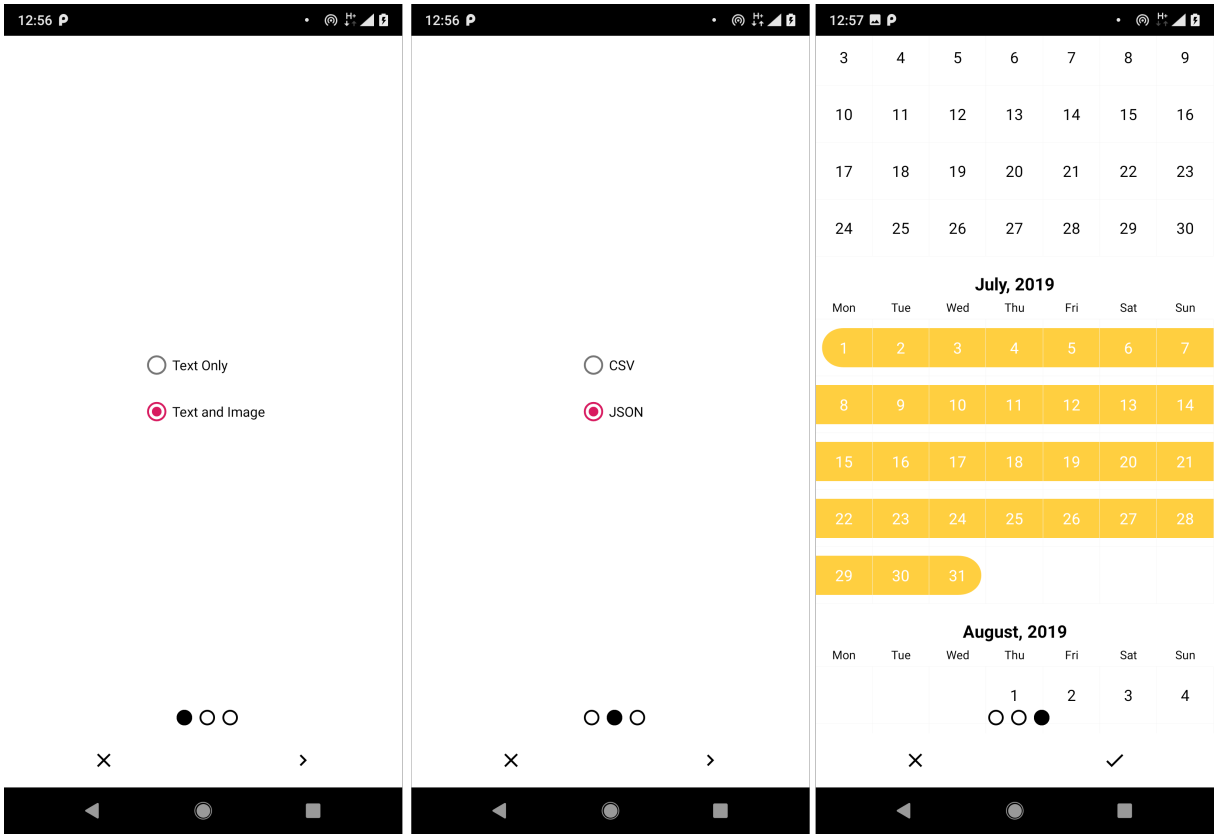


Figura 4.4: Formularul de export

Procesul de export funcționează în felul următor:

1. Utilizatorul completează formularul prezentat mai sus și datele exportului sunt sal-

vate în baza de date locală cu statusul *uploading*.

2. Aplicația afișează o notificare și trimite bonurile în cloud. La final, sesiunea de export este salvată cu statusul *waiting download link*.
3. Bonurile sunt procesate în cloud pentru a fi trnsformate (în cazul în care formatul selectat a fost CSV) și arhivate.
4. Serviciul cloud trimite o notificare către dispozitiv ce conține *link-ul* de descărcare a datelor. Acesta este salvat în baza de date locală, împreună cu statusul *complete*.

Figura 4.5 prezintă ecranul ce afișează export-urile utilizatorului. Un element conține intervalul calendaristic aferent exportului și statusul acestuia. Atunci când sesiunea de export este finalizată, două butoane sunt afișate, cu funcționalitatea de a copia link-ul de descărcare pe clipboard sau de a descărca datele arhivate pe dispozitiv.

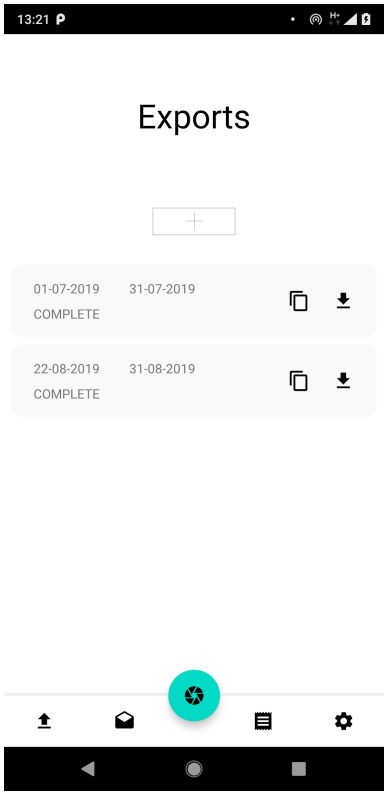


Figura 4.5: Lista de export-uri

4.5.1 Implementare

La nivelul domeniului, funcționalitatea de export este modelată de interfața `ExportUseCase`. Aceasta definește funcțiile de listare a tuturor exporturilor de pe dispozitiv, creeare a unui nou export și marcarea unui export ca finalizat la primirea unei notificări.

Programul 4.4: Interfața ExportUseCase

```
1 interface ExportUseCase {
2     fun list(): Flowable<List<Export>>
3     fun newExport(manifest: Session): Completable
```

```
4       fun markAsFinished(notification: FinishedNotification): Completable
5   }
6
7   data class Session(
8       val firstDate: Date,
9       val lastDate: Date,
10      val content: Content,
11      val format: Format,
12      val id: String
13  ) : Parcelable {
14
15      enum class Content {
16          TextOnly,
17          TextAndImage
18      }
19
20      enum class Format {
21          JSON,
22          CSV
23      }
24  }
25
26  data class FinishedNotification(
27      val exportId: String,
28      val downloadUrl: String
29  )
```

Trimiterea datelor către cloud se face printr-un serviciu de tipul *foreground*. Pe sistemul Android, *serviciile foreground* sunt servicii care interacționează cu utilizatorul prin intermediul unei notificări și au șanse foarte mici de a fi oprite de către sistem pentru a recupera resurse. Acestea sunt recomandate pentru a executa activități de lungă durată care nu blochează interfața și care sunt declanșate de o acțiune a utilizatorului. Funcția `upload` este apelată într-un astfel de serviciu cu argumentul obținut pe baza formularului prezentat mai sus. La creearea argumentului `Session` este generat un id unic ce va fi folosit pentru identificarea exportului pe durata funcționării acestuia.

Interacțiunea aplicației cu serviciile cloud Firebase pentru această funcționalitate este ilustrată în diagrama din figura 4.6.

Serviciul de *foreground* încarcă bonurile aferente exportului într-un spațiu de stocare *Firebase Cloud Storage*, sub un folder ce are numele id-ului unic generat, în format JSON (opțional și imaginile JPEG respective). La încărcarea cu succes a acestor fișiere, obiectul `Session` este trimis ca manifest în colecția *manifests* din serviciul *Firebase Firestore*.

O funcție *Firebase Cloud Functions* este configurată pentru a asculta modificări ale colecției *manifests* și a se declanșa la creearea unui nou obiect. Aceasta citește id-ul manifestului și opțiunea de format (JSON sau CSV) și procesează fișierele din folder-ul corespunzător din *Cloud Storage*. Apoi încarcă o arhivă *zip* a acestui folder în folder-ul *downloads* din Cloud Sotrage și generează un link de descărcare, pe care îl trimite către serviciul *Firebase Cloud Messaging* pentru a fi trimis mai departe ca notificare către dispozitiv.

Pentru ca notificarea să ajungă doar la dispozitivul care a creeat exportul, aplicația

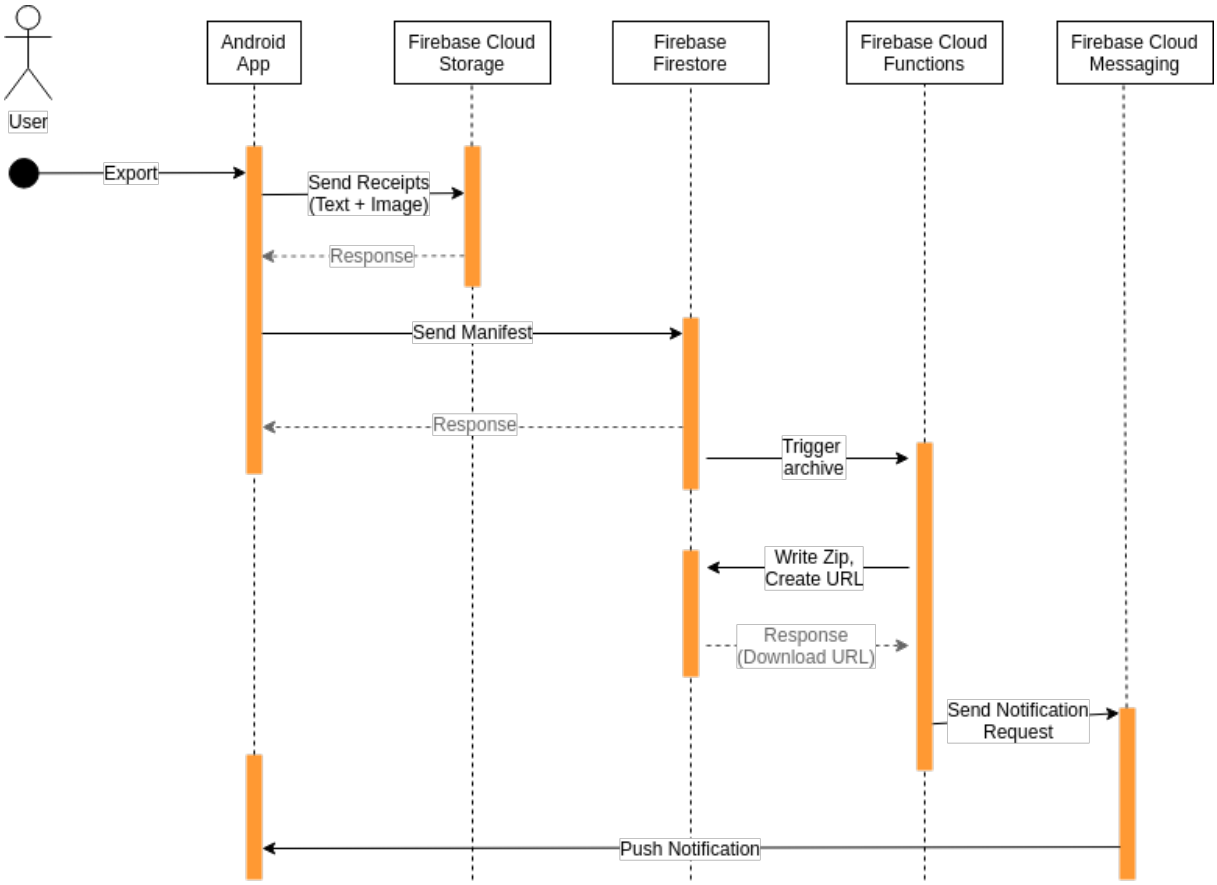


Figura 4.6: Procesul de trimitere

folosește clientul Android al *Firebase Cloud Messaging*. Acesta presupune implementarea unui serviciu ce extinde `FirebaseMessagingService`. Acest serviciu generează un *token* folosit pentru a primi notificări și îl face disponibil în metoda `onNewToken(token: String)`. Aplicația salvează acest token în *shared preferences* și îl trimite în obiectul manifest. Astfel, acest token ajunge pe cloud, de unde este transmis către *Firebase Cloud Messaging*.

Firebase Cloud Messaging suportă două tipuri de mesaje: *notification messages* și *data messages*, sau o combinație dintre cele două. (Firebase n.d.) Pentru ca notificarea să fie gestionată imediat ce a fost primită în metoda `onMessageReceived(message: RemoteMessage)` a serviciului `FirebaseMessagingService` este folosită doar funcționalitatea de *data message*. Odată ce notificarea este recepționată de către dispozitiv, metoda `markAsFinished(notification: FinishedNotification)` este apelată pentru a actualiza baza de date și o notificare este afișată.

Capitolul 5

Concluzie

5.1 Avantaje și obstacole

Am gândit ReceiptScan în jurul ideii democratizării informațiilor financiare. Utilizarea bonurilor fiscale ca date de intrare pentru aplicație aduce avantajul de a nu depinde de modul în care a fost făcută achiziția. Majoritatea utilizatorilor au mai multe carduri, dar folosesc și bani lichizi pentru unele achiziții. Soluția propusă adună toate aceste tranzacții într-un singur loc.

Un alt avantaj este flexibilitatea datelor. Soluția propusă oferă o vizualizare rapidă a tranzacțiilor în aplicație, asemenea altor soluții, dar oferă și exportul datelor, pentru ca acestea să fie folosite pentru analize detaliate în programe cum ar fi *Excel*.

Principalul obstacol al acestei aplicații este înțelegerea automată a bonurilor fiscale. Am împărțit această sarcină în două probleme: R și extragerea informațiilor relevante din textul oferit de OCR. Deși problema recunoașterii caracterelor este considerată rezolvată în condiții ideale, aceasta pune în continuare probleme sub condiții imperfecte. În cadrul acestei aplicații, condițiile pentru OCR sunt date de hartie de proastă calitate, posibil mototolită, cerneală ștearsă și poze care pot fi de proastă calitate și pun probleme soluțiilor existente. Extragerea informațiilor din textul bonurilor fiscale este și ea o problemă dificilă deoarece bonurile nu respectă un format standard, iar rezultatul OCR-ului poate să nu fie perfect. Abordarea acestor probleme va fi prezentată într-un capitol ulterior.

Anexe

Anexa A

Script-urile folosite pentru compararea soluțiilor OCR

Lorem ipsum

Anexa B

Algoritmul de unificare a liniilor

Programul B.1: LineUnification.kt

```
1  data class OcrElement (
2      val text: String,
3      val left: Int,
4      val top: Int,
5      val right: Int,
6      val bottom: Int
7  ) {
8      val mid: Float
9          get() = (bottom + top).toFloat() / 2
10
11     val height: Int
12         get() = bottom - top + 1
13 }
14
15 typealias Line = List<OcrElement>
16
17 fun firebaseTextToLines(text: FirebaseVisionText) {
18     val sorted = text
19         .textBlocks
20         .flatMap { it.lines }
21         .map {
22             OcrElement(
23                 it.text,
24                 it.boundingBox!!.left,
25                 it.boundingBox!!.top,
26                 it.boundingBox!!.right,
27                 it.boundingBox!!.bottom
28             )
29         }
30         .sortedBy { it.mid }
31
32     val unifiedLines = LinkedList<Line>()
33
34     var currentLine = ArrayList<OcrElement>()
35
36     val boxesIterator = sorted.iterator()
37
38     if (boxesIterator.hasNext()) {
39         var lastBox = boxesIterator.next()
40         currentLine.add(lastBox)
41
42         while (boxesIterator.hasNext()) {
43             val crtBox = boxesIterator.next()
44             val threshVal = THRESHOLD * (crtBox.height + lastBox.height).toFloat() / 2
45             if (crtBox.mid - lastBox.mid < threshVal) {
```

```
                currentLine.add(crtBox)
            } else {
                val sortedLine = currentLine.sortedBy { it.left }
                unifiedLines.add(
                    Line(
                        sortedLine
                    )
                )
                currentLine = ArrayList()
                currentLine.add(crtBox)
            }
            lastBox = crtBox
        }
    }

    if (currentLine.isNotEmpty()) {
        unifiedLines.add(
            Line(
                currentLine
            )
        )
    }

    return unifiedLines
}
```

Anexa C

Algoritmul de extragere a informațiilor

Programul C.1: Algoritmul de extragere

```
typealias OcrElements = Sequence<OcrElement>

class Extractor @Inject constructor(
    private val defaults: ReceiptDefaults
) {
    operator fun invoke(elements: OcrElements): Draft {
        val receipt = RawReceipt.create(elements)
        val text = receipt.text
        val merchant = extractMerchant(receipt)
        val date = extractDate(text)
        val currency = defaults.currency
        val category = defaults.category
        val (total, products) = ProductsAndTotalStrategy(
            receipt
        ).execute()
        return Draft (
            merchant,
            date,
            total,
            currency,
            category,
            products.map { Product(it.name, it.price) },
            elements.map { OcrElement(it.text, it.top, it.bottom, it.left, it.right) }.toList()
        )
    }
}

class RawReceipt(private val lines: List<Line>) : Iterable<RawReceipt.Line> {
    override fun iterator(): Iterator<Line> = lines.iterator()

    class Line(private val elements: List<OcrElement>) : Iterable<OcrElement> {
        override fun iterator() = elements.iterator()
        val text by lazy { elements.joinToString(" ") { it.text } }
        val height by lazy { elements.map { it.height }.average() }
        val top by lazy { elements.map { it.top }.min()!! }
        val bottom by lazy { elements.map { it.bottom }.max()!! }
    }

    val averageLineHeight by lazy { this.lines.map { it.height }.average() }

```

```
41      val text by lazy { lines.joinToString("\n") { it.joinToString("\t") { t -> t.text } } }
42
43      companion object {
44          private const val THRESHOLD = 0.5F
45          fun create(elements: OcrElements): RawReceipt {
46              val sorted = elements.sortedBy { t -> t.mid }
47              val unifiedLines = LinkedList<Line>()
48
49              var currentLine = ArrayList<OcrElement>()
50              val boxesIterator = sorted.iterator()
51
52              if (boxesIterator.hasNext()) {
53                  var lastBox = boxesIterator.next()
54                  currentLine.add(lastBox)
55
56                  while (boxesIterator.hasNext()) {
57                      val crtBox = boxesIterator.next()
58                      val threshVal = THRESHOLD * (crtBox.height + lastBox.height).toFloat() / 2
59                      if (crtBox.mid - lastBox.mid < threshVal) {
60                          currentLine.add(crtBox)
61                      } else {
62                          val sortedLine = currentLine.sortedBy { it.left }
63                          unifiedLines.add(
64                              Line(
65                                  sortedLine
66                              )
67                          )
68                          currentLine = ArrayList()
69                          currentLine.add(crtBox)
70                      }
71                      lastBox = crtBox
72                  }
73              }
74
75              if (currentLine.isNotEmpty()) {
76                  unifiedLines.add(
77                      Line(
78                          currentLine
79                      )
80                  )
81              }
82
83              return RawReceipt(unifiedLines)
84          }
85      }
86  }
87
88  private const val MERCHANT_MIN_LENGTH = 2
89
90  fun extractMerchant(rawReceipt: RawReceipt): String? {
91      val linesIterator = rawReceipt.iterator()
92      while (linesIterator.hasNext()) {
93          val line = linesIterator.next()
94          if (line.text.length < MERCHANT_MIN_LENGTH) continue
95
96          val nextLine = if (linesIterator.hasNext()) linesIterator.next() else null
97
98          val heightThreshold = 1.2 * rawReceipt.averageLineHeight
99
100         return if (
101             line.height > heightThreshold &&
102             nextLine != null &&
103             nextLine.text.split(" ").size < 2 &&
104             nextLine.height > heightThreshold &&
105             nextLine.top - line.bottom < rawReceipt.averageLineHeight
106         )
107             line.text + " " + nextLine.text
108         else
```

```
109             line.text
110         }
111         return null
112     }
113
114     fun extractDate(receiptText: String): Date =
115         findDatesWithPatterns(receiptText).firstOrNull() ?: Date()
116
117     fun parseNumber(string: String): Float? =
118         Regex("[+-]?([0-9]*[.,])[0-9]+")
119             .findAll(string.removeSpaceInFloat())
120             .map { it.value.replace(',', '.') }
121             .mapNotNull { it.toFloatOrNull() }
122             .sortedDescending()
123             .firstOrNull()
124
125     private val spaceBefore = "(\\d)\\s([.,])".toRegex()
126     private val spaceAfter = "([.,])\\s(\\d)".toRegex()
127
128     private fun String.removeSpaceInFloat(): String = this
129         .replace(spaceBefore, "$1$2")
130         .replace(spaceAfter, "$1$2")
131
132
133     class ProductsAndTotalStrategy(private val receipt: RawReceipt) {
134         private val horizontalBorders: HorizontalBorders
135
136         private var lastKey: Optional<OcrElement> = None
137
138         private val totalMarkRegex = "total|ammount|summe".toRegex()
139
140         private val keyPriceResults = mutableListOf<ResultObj>()
141
142         init {
143             horizontalBorders = boundaries(receipt)
144         }
145
146         fun execute(): Pair<Float?, List<Product>> {
147             walkAndProcess()
148             return makeResult()
149         }
150
151         private fun makeResult(): Pair<Float?, List<Product>> {
152             val price = keyPriceResults
153                 .mapNotNull {
154                     when (it) {
155                         is ResultObj.Total -> it
156                         else -> null
157                     }
158                 }
159                 .sortedWith(compareBy({ -it.price }, { it.top }))
160                 .firstOrNull()
161
162             val products = keyPriceResults
163                 .mapNotNull {
164                     when (it) {
165                         is ResultObj.Product -> it
166                         else -> null
167                     }
168                 }
169                 .filter { if (price != null) it.top < price.top else true }
170                 .map { Product(it.name, it.price) }
171             return price?.price to products
172         }
173
174         private fun walkAndProcess() {
175             for (line in receipt) {
176                 for (element in line) {
```

```
                val left = isAlignedToLeft(element)
                val right = isAlignedToRight(element)
                if (left && right) {
                    processKeyValue(element)
                } else if (left) {
                    processKey(element)
                } else if (right) {
                    processPrice(element)
                }
            }
        }
    }

    private fun processPrice(priceElement: OcrElement) {
        val price = parseNumber(priceElement.text)
        price?.let {
            val mLastKey = lastKey
            if (mLastKey is Just) {
                val keyElement = mLastKey.value
                if (priceElement.top - keyElement.bottom < 0.5 * priceElement.height) {
                    makeResult(keyElement, it)
                } else {
                    lastKey = None
                }
            }
        }
    }

    private fun processKey(element: OcrElement) {
        val digitCount = element.text.count { it.isDigit() }
        if (digitCount < 0.3 * element.text.length) {
            lastKey = Just(element)
        }
    }

    private fun processKeyValue(element: OcrElement) {
        val price = parseNumber(element.text)
        price?.let {
            val name = element.text.split(" ").take(3).joinToString(" ")
            if (name.length > 1) {
                makeResult(name, price, element)
            }
        }
    }

    private fun makeResult(key: String, price: Float, element: OcrElement) {
        val keyLowercase = element.text.toLowerCase()
        if (keyLowercase.contains(totalMarkRegex)) {
            keyPriceResults.add(
                ResultObj.Total(
                    price,
                    element.top
                )
            )
        } else {
            keyPriceResults.add(
                ResultObj.Product(
                    price,
                    key,
                    element.top
                )
            )
        }
    }

    private fun makeResult(keyElement: OcrElement, price: Float) {
        val keyLowercase = keyElement.text.toLowerCase()
```

```
244         if (keyLowercase.contains(totalMarkRegex)) {
245             keyPriceResults.add(
246                 ResultObj.Total(
247                     price,
248                     keyElement.top
249                 )
250             )
251         } else {
252             keyPriceResults.add(
253                 ResultObj.Product(
254                     price,
255                     keyElement.text,
256                     keyElement.top
257                 )
258             )
259         }
260     }
261
262     private fun isAlignedToLeft(element: OcrElement): Boolean =
263         (element.left - horizontalBorders.left).toFloat() / horizontalBorders.width < ALIGN_THRESH
264
265     private fun isAlignedToRight(element: OcrElement): Boolean =
266         (horizontalBorders.right - element.right).toFloat() / horizontalBorders.width < ALIGN_THRESH
267
268     private fun boundaries(receipt: RawReceipt): HorizontalBorders {
269         val elements = receipt.flatten()
270         var top = Int.MAX_VALUE
271         var bottom = -1
272         var left = Int.MAX_VALUE
273         var right = -1
274         for (a in elements) {
275             if (a.top < top) top = a.top
276             if (a.left < left) left = a.left
277             if (a.bottom > bottom) bottom = a.bottom
278             if (a.right > right) right = a.right
279         }
280         return HorizontalBorders(
281             left,
282             right
283         )
284     }
285
286     private class HorizontalBorders(val left: Int, val right: Int)
287     private val HorizontalBorders.width
288         get() = this.right - this.left + 1
289
290     private sealed class ResultObj {
291         class Total(val price: Float, val top: Int) : ResultObj()
292         class Product(val price: Float, name: String, val top: Int) : ResultObj() { val name = name.
      toUpperCase() }
293     }
294
295     companion object {
296         private const val ALIGN_THRESH = 0.1
297     }
298 }
299
300 private const val DIGIT_MISTAKES = "[\\doO]"
301
302 val formats = mapOf(
303     "$DIGIT_MISTAKES{8}" to "yyyyMMdd",
304     "$DIGIT_MISTAKES{2}/$DIGIT_MISTAKES{2}/$DIGIT_MISTAKES{4}" to "dd/MM/yyyy",
305     "$DIGIT_MISTAKES{2}-$DIGIT_MISTAKES{2}-$DIGIT_MISTAKES{4}" to "dd-MM-yyyy",
306     "$DIGIT_MISTAKES{2}\\.$DIGIT_MISTAKES{2}\\.$DIGIT_MISTAKES{4}" to "dd.MM.yyyy",
307     "$DIGIT_MISTAKES{2}/$DIGIT_MISTAKES{2}/$DIGIT_MISTAKES{2}(?!\\d)" to "dd/MM/yy",
308     "$DIGIT_MISTAKES{2}-$DIGIT_MISTAKES{2}-$DIGIT_MISTAKES{2}(?!\\d)" to "dd-MM-yy",
309     "$DIGIT_MISTAKES{2}\\.$DIGIT_MISTAKES{2}\\.$DIGIT_MISTAKES{2}(?!\\d)" to "dd.MM.yy",
310     "$DIGIT_MISTAKES{4}/$DIGIT_MISTAKES{2}/$DIGIT_MISTAKES{2}" to "yyyy/MM/dd",
```

```
    "$DIGIT_MISTAKES{4}-$DIGIT_MISTAKES{2}-$DIGIT_MISTAKES{2}" to "yyyy-MM-dd",
    "$DIGIT_MISTAKES{4}\\.$DIGIT_MISTAKES{2}\\.$DIGIT_MISTAKES{2}" to "yyyy.MM.dd"
)

fun findDatesWithPatterns(searchedString: String): Sequence<Date> {
    var results = sequenceOf<Pair<String, String>>()
    val fakeZeros = "Oo".toRegex()
    for ((regex, format) in formats) {
        val result = regex.toRegex().findAll(searchedString)
        val seq = result.map { it.value.replace(fakeZeros, "0") to format }
        results += seq
    }

    val now = Date()

    return results
        .mapNotNull {
            try {
                SimpleDateFormat(it.second, Locale.US).parse(it.first)
            } catch (e: ParseException) {
                null
            }
        }
        .sortedBy { abs(it.time - now.time) }
}
```

Bibliografie

[1] Alistair Cockburn. "Structuring use cases with goals". In: *Journal of Object-Oriented Programming* 10.5 (1997), pp. 56–62.

[2] Andrew Hunt and David Thomas. "The Pragmatic Programmer: From Journeyman to Master". In: Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. Chap. 7. Before the Project, pp. 204–207. ISBN: 0-201-61622-X.

[3] Bill Janssen et al. "Receipts2Go: The big world of small documents". In: Sept. 2012, pp. 121–124. DOI: 10.1145/2361354.2361381.

[4] Elisabeth Freeman Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, 2004. Chap. Chapter 2: the Observer Pattern, pp. 37–78.

[5] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017. Chap. Chapter 15: What Is Architecture?, pp. 141–151.

[6] *Mobile Operating System Market Share Europe — StatCounter Global Stats*. 2019. URL: https://gs.statcounter.com/os-market-share/mobile/europe (visited on 08/27/2019).

[7] *Mobile Operating System Market Share Romania — StatCounter Global Stats*. 2019. URL: https://gs.statcounter.com/os-market-share/mobile/romania (visited on 08/27/2019).

[8] *Mobile Operating System Market Share Worldwide — StatCounter Global Stats*. 2019. URL: https://gs.statcounter.com/os-market-share/mobile/worldwide (visited on 08/27/2019).

[9] Rizlène Raoui-Outach et al. "Deep Learning for automatic sale receipt understanding". In: (Dec. 2017).

[10] ReactiveX. *Observable documentation*. URL: http://reactivex.io/RxJava/javadoc/io/reactivex/Observable.html#throttleLast-long-java.util.concurrent.TimeUnit- (visited on 08/28/2019).

[11] Tesseract. *Tesseract Project*. URL: https://github.com/tesseract-ocr/tesseract (visited on 08/28/2019).

[12] Tomasz Nurkiewicz and Ben Christensen. *Reactive Programming with RxJava: Creating Asynchronous, Event-Based Applications*. O'Reilly Media, 2016. Chap. Chapter 1: Reactive Programming with RxJava, pp. 1–2.