

UNIVERSITATEA DIN BUCUREŞTI
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

Lucrare de licență

ReceiptScan

Aplicație Android pentru digitalizarea bonurilor fiscale

**Coordonator științific
prof. Ioana Leuștean**

**Absolvent
Lucian Boacă**

București, septembrie, 2019

Abstract

Urmărirea cheltuielilor este o sarcină importantă pentru a avea o viziune clară asupra situației financiare personale. Aplicațiile mobile existente rezolvă doar parțial această problemă. Acestea fie oferă prea puține informații, fie necesită un efort mult prea mare din partea utilizatorului. ReceiptScan oferă o soluție pentru urmărirea cheltuielilor, facilitând scanarea chitanțelor de cumpărături și oferind datele extrase într-un mod flexibil.

Pentru rezolvarea problemei am dezvoltat această aplicație Android ce utilizează un modul OCR avansat și un set de reguli pentru a extrage informații din bonuri fiscale. Aceasta pune la dispoziția utilizatorului un eran de vizualizare a cheltuielilor sale și opțiunea de a exporta datele în cloud pentru a le descărca apoi în afara dispozitivului.

La momentul scrierii și din căutările făcute, aceasta este o abordare unică. Aplicația este disponibilă în mod open source, pe GitHub. Aceasta oferă rezultate suficient de bune și poate fi ușor extinsă cu alte reguli pentru diferite tipuri chitanțe.

Abstract

Expenses tracking is an important task for having a clear view over personal financial situation. Existing mobile apps solve this problem only partially. They either offer way to few information or require a great involvement on the user side. ReceiptScan offers a solution for expenses tracking, making it easy to scan receipts and offering the extracted data in a flexible way.

For solving this problem, I developed this Android app which makes use of an advanced OCR module and a set of rules to extract the information out of receipts. This app displays a reports screen to show user's spending and offers the option to export the data in cloud to further be downloaded outside the device.

At the moment of writing and from the my research, this is an unique approach. The app is available open source on GitHub. It offers good enough results and it can easily be extended with other rules for different receipt types.

Cuprins

1	Introducere	1
2	Specificațiile aplicației	3
2.1	Capturarea și înțelegerea imaginilor	3
2.2	Editare draft	5
2.3	Gestionare setări	7
2.4	Colectarea bonuri fiscale	8
2.5	Export	9
2.6	Vizualizare cheltuieli	11
3	Tehnologii. Arhitectură. Persistență	13
3.1	Alegerea platformei Android	13
3.2	Tehnologii utilizate	14
3.2.1	Kotlin	14
3.2.2	RxJava	14
3.2.3	Android Architecture Components	15
3.2.4	Firebase ML Vision	15
3.2.5	Firebase Cloud Services	15
3.3	Arhitectura aplicației	16
3.4	Persistență	17
3.4.1	SQLITE	17
3.4.2	Spațiul de stocare intern	18
3.4.3	Shared Preferences	18
4	Detalii de implementare	19
4.1	Algoritmul de extragere a informației	19
4.1.1	Recunoașterea textului	19
4.1.2	Extragerea informațiilor din text	22
4.2	Capturarea și înțelegerea imaginilor	23
4.3	Editare draft	24
4.4	Gestionare setări	26

4.5	Colectarea Datelor	27
4.6	Export	27
4.7	Vizualizarea datelor	30
4.8	Testare automată	32
	Concluzie	35
	Appendices	37
A	Script-urile folosite pentru compararea soluțiilor OCR	39
B	Algoritmul de unificare a liniilor	43
C	Algoritmul de extragere a informațiilor	45

Capitolul 1

Introducere

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

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 plăștilor 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 au 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.

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 a înțelege o gamă cât mai largă de bonuri fiscale populare în România și de a necesita un efort minim din partea utilizatorului;
- 2. Respectarea 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 astfel 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 au 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 de implementat.

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* [5] 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* [11] o abordare de procesare bazată pe *deep learning*.

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 4.8 prin prezentarea concluziilor și a observațiilor de final.

Capitolul 2

Specificațiile aplicăției

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), să cum acestea sunt definite în lucrarea *Structuring use cases with goals* [2] și inspirate din exemplul prezentat în *The Pragmatic Programmer* [3]. 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 fiecărei funcționalități însotită de *screenshot-uri* din aplicație, 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, iar **extensiile** sunt scenarii secundare, ce servesc scopului 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 și

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.

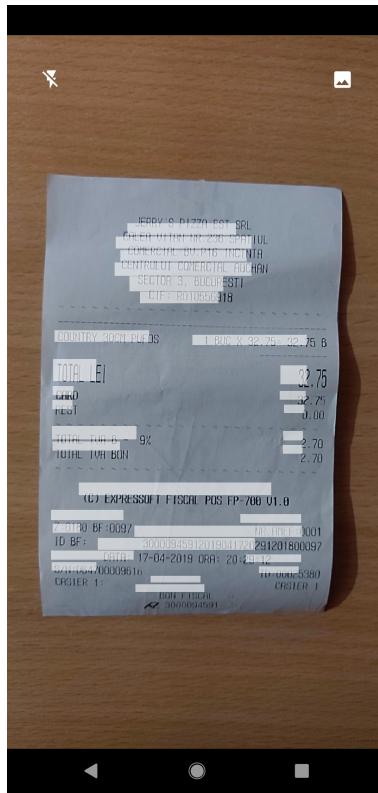


Figura 2.1: Ecranul de Scanare

- **Scop:** Capturarea unei imagini și extragerea informațiilor relevante din aceasta;
- **Condiție de succes:** Prezența unei înregistrări în baza de date ce modeleză 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;

Imaginiile 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)

Variatii

- 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 camera 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 2.2a prezintă ecranul unde utilizatorul vede toate drafturile, iar figura 2.2b 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;

Mentiri

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:

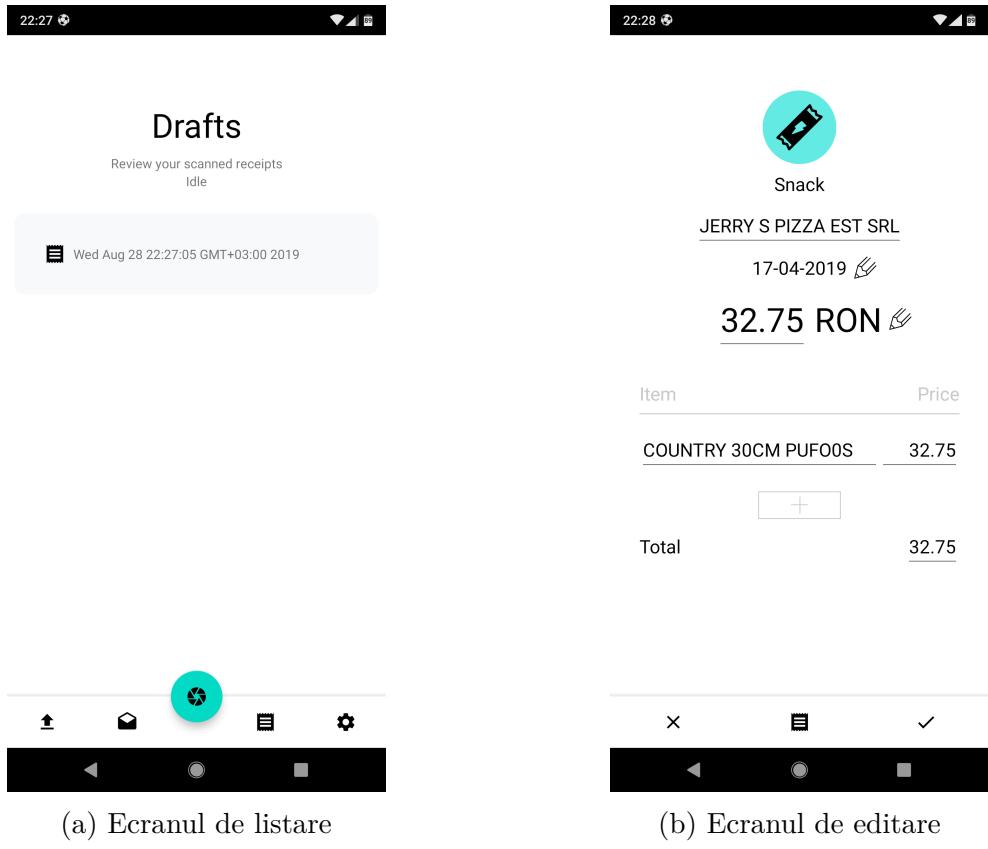


Figura 2.2: Ecranele de gestionare a drafturilor

- 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;

Variatii

- 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

Aplicația permite setarea de valori predefinite pentru monedă și categorie, care să fie folosite în interpretarea imaginilor. De asemenea, aplicația permite activarea sau dezactivarea colectării de date în mod anonim (Specificația 2.4). Figura 2.3 prezintă ecranul de modificare a setărilor.

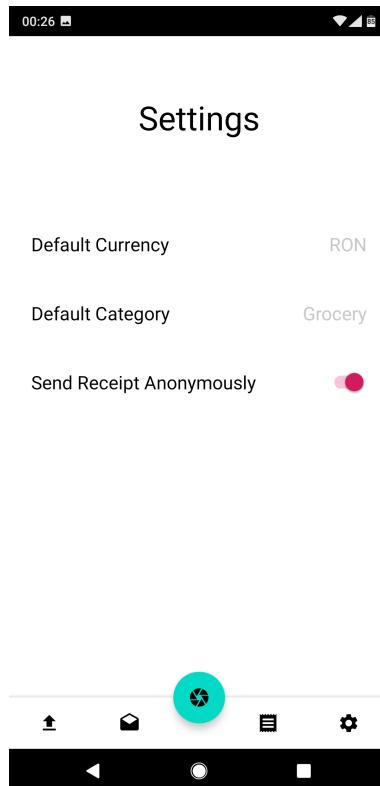


Figura 2.3: Ecranul de setări

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

- **Scenarii de eșec:** Modificările nu pot fi persistate; Valorile nu pot fi accesate;

Mențiuni

Setările considerate sunt:

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

Principalul scenariu

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

2.4 Colectarea bonuri fiscale

Așa cum am menționat mai sus, această aplicație este construită având în vedere siguranța și confidențialitatea datelor. Totuși, lipsa de date care să facă legătura între imagini ale chitanțelor comerciale și conținutul acestora este un impediment în a rezolva problema în cauză prin metode avansate, care să ofere o performanță sporită. De exemplu, construirea unui astfel de set de date ar conduce la întocmirea unui *benchmark* asupra căruia să fie testate noi metode.

Astfel este motivată implementarea acestei funcționalități. Colectarea trebuie să se facă în *mod anonim*, numai cu *acordul utilizatorului* și să aibă un *impact minim asupra experienței utilizatorului*.

- **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

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.

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

Conform cu motivația aplicației, datele înregistrate de aceasta trebuie să fie disponibile independent de aceasta. Exportul datelor se face prin încărcarea acestora pe un server din cloud și primirea unui link către datele respective, arhivate în format *zip*. Opțiunile pentru exportul datelor (ilustrate în figura 2.4) sunt următoarele:

- **Continut:** 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.

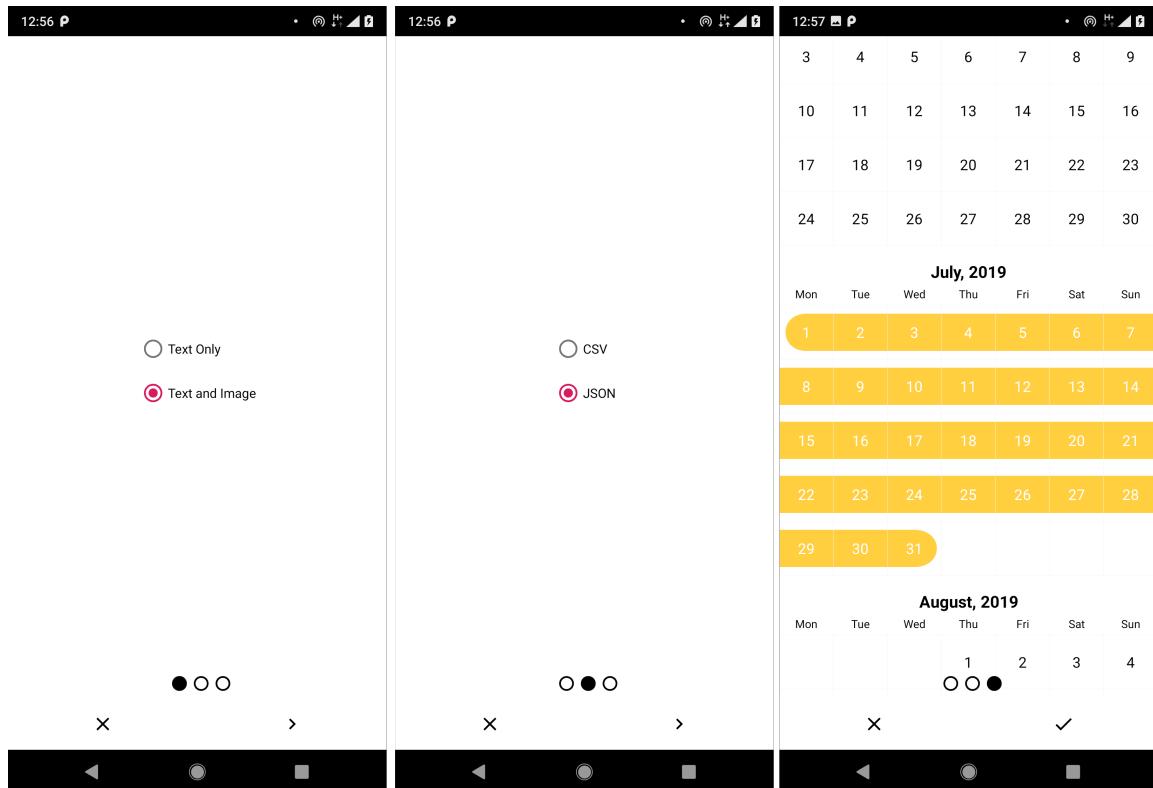


Figura 2.4: Formularul de export

Odată finalizat exportul datelor, dispozitivul primește o notificare ce conține *link-ul* de descărcare a acestora. Figura 2.5 prezintă ecranul ce afișează lista tuturor *export-urilor* făcute de pe dispozitiv, iar cele complete pun la dispoziție două butoane pentru copierea *link-ului* pe *clipboard*, respectiv descărcarea arhivei.



Exports

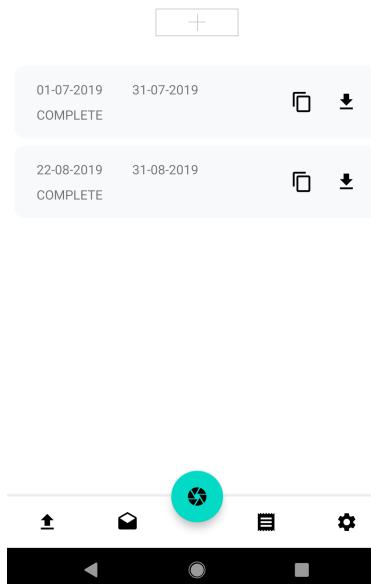


Figura 2.5: Lista de export-uri

- **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;

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;

Variatii

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

Extensii

- În cazul lipsei de conectivitate, acțiunea de export este programată pentru o dată ulterioră, 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;

2.6 Vizualizare cheltuieli

Ecranul de vizualizare a cheltuielilor (prezentat în figura 2.6a) este punctul de intrare în aplicație. Utilizatorul își poate vedea cheltuielile organizate pe monede, luni și categorii. În a doua jumătate a ecranului este afișată lista tuturor cheltuielilor din selecția curentă. Prin selectarea uneia, se deschide ecranul de vizualizare a unui bon fiscal (figura 2.6b)

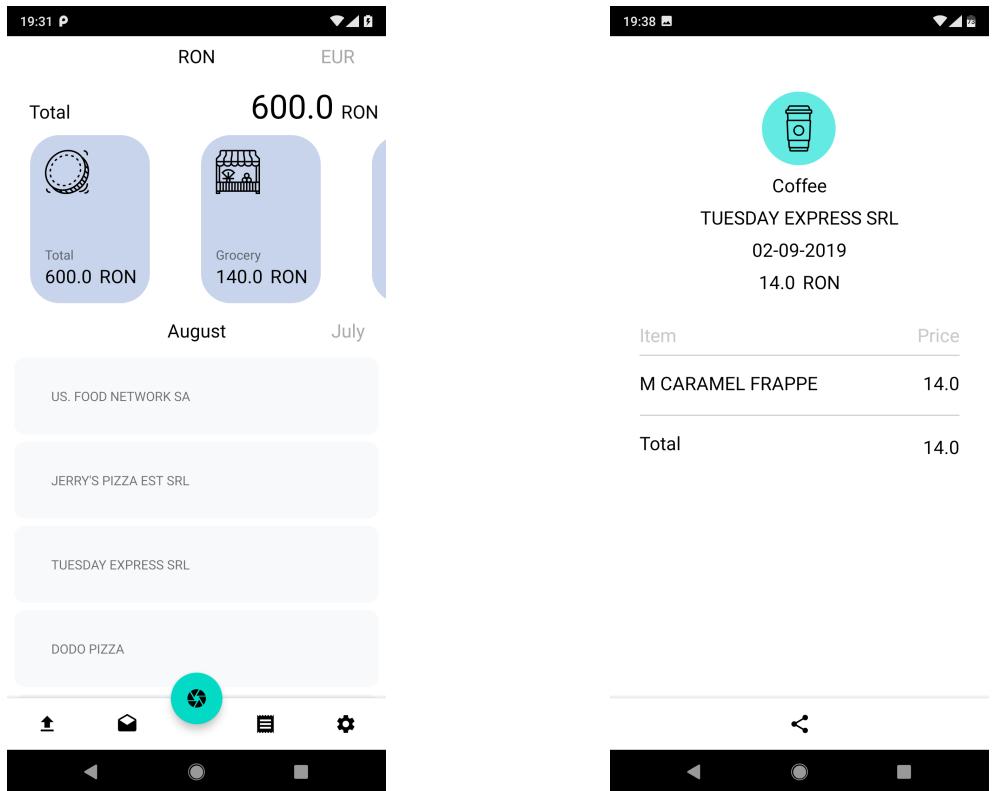
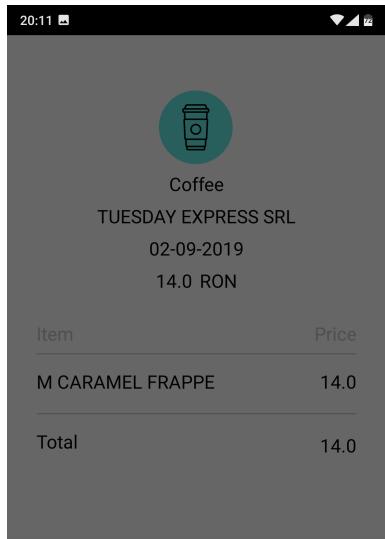


Figura 2.6: Ecranele de gestionare a cheltuielilor

Din ecranul de vizualizare a unui bon, utilizatorul poate trimite acel bon printr-o aplicație externă, cum ar fi clientul de *e-mail* sau o aplicație de mesagerie. Motivația acestei funcționalități sunt cazurile în care cheltuielile sunt împărtășite între mai multe persoane sau decontate și trebuie atașate prin *e-mail*. Opțiunile de trimis externă sunt: *doar text*, *doar imagine*, *imagine și text*. Figura 2.7 ilustrează procesul de distribuire externă.

- **Scop:** Vizualizarea datelor în legătură cu cheltuielile; Trimiterea informațiilor aferente unei tranzacții pe canale externe;
- **Condiții de succes:** Afisarea cu succes a datelor; Trimiterea cu succes către aplicații externe;
- **Condiții de eșec:** Neafisarea datelor când acestea există; Eșecul în a face legătura cu alte aplicații pentru a trimite un bon;

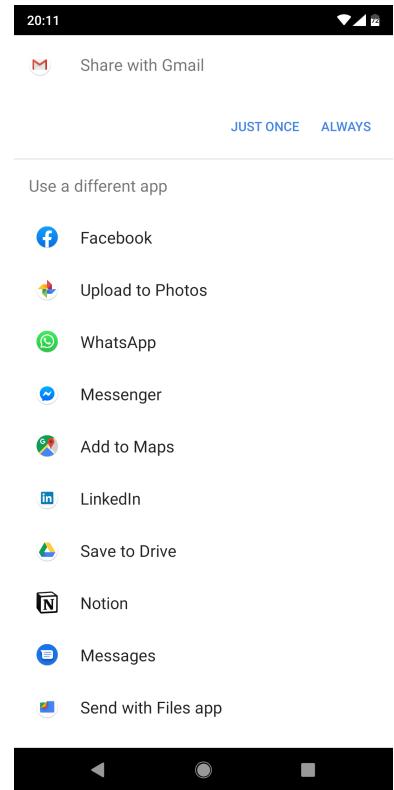


Chose what to share

Text Only

Image Only

Image and Text



(a) Opțiunile de trimitere externă

(b) Ecranul de vizualizare a unui bon

Figura 2.7: Aplicațiile externe prin care poate fi trimis un bon

Mențiuni

- Datele sunt afișate organizate pe monedă, lună și categorie;
- La nivelul afișării pe categorii, există și o opțiune de a afișa toate tranzacțiile, indiferent de categorie;
- Datele despre o tranzacție sunt afișate într-un ecran separat;

Scenariu principal

1. Utilizatorul deschide aplicația;
2. Sunt interogate toate monedele și lunile disponibile și afișate datele pentru luna curentă și ultima monedă utilizată;
3. Utilizatorul accesează o tranzacție din lista returnată;
4. Utilizatorul distribuie această tranzacție;

Extensii

- Utilizatorul selectează o nouă monedă, categorie sau lună, iar datele sunt actualizate;

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 detine 76.08% cotă de piată la nivel global [10], 73.71% la nivelul Europei [8] și 81.3% la nivelul României [9]. 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 [14] 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*[6] 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 [7].

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*.

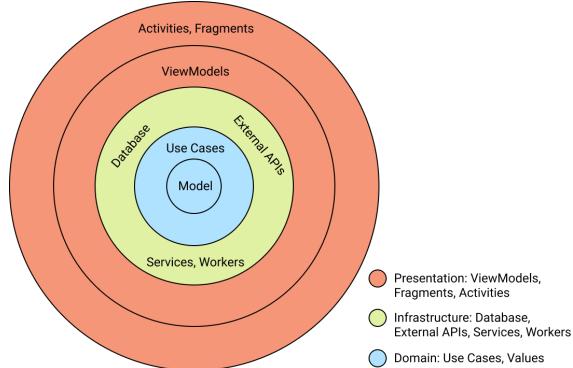


Figura 3.1: Nivelurile conceptuale ale arhitecturii aplicatiei

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 depend 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 necesită un serviciu implementat la un nivel inferior. Injectarea dependințelor este exemplificată în programul 3.1.

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 *runtime*, 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:

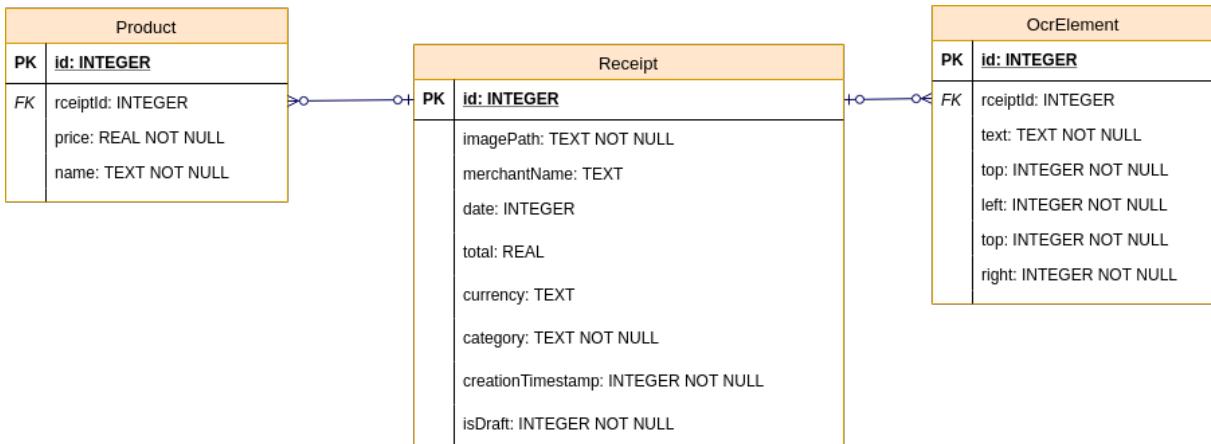


Figura 3.2: Modelul de date SQL

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 de implementare

Secțiunile precedente au prezentat specificațiile aplicației, arhitectura și principalele tehnologii care facilitează implementarea. În continuare vor fi discutate detaliile de implementare ale fiecărei funcționalități.

4.1 Algoritmul de extragere a informației

Înțelegerea conținutului bonurilor fiscale din imagini reprezintă principala provocare a acestui proiect și funcționalitatea în jurul căreia este construită întreaga aplicație. În mod natural, aceasta 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 două sarcină și pot folosi metode mai avansate, de *machine learning*, pe măsură ce sunt colectate mai multe date. Aceste opțiuni sunt subiectul unor cercetări viitoare.

4.1.1 Recunoașterea textului

O constrângere 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* [13]. Pentru dezvoltatorii de aplicații mobile, Google oferă librăria *Firebase Vision*, cu suport gratuit pentru OCR pe dispozitiv. Compararea dintre cele două soluții a fost făcută astfel:

- Firebase Vision a fost rulat folosind un test de instrumentare, întrucât această librărie nu poate rula decât pe un dispozitiv mobil;

- Tesseract a fost rulat pe un computerul personal, folosind *Python*;
- Imaginea a fost pre-procesată doar pentru Tesseract, întrucât această librărie nu oferă o performanță satisfăcătoare pe imagini neprocesate;
- Preprocesarea a constat în aplicarea unui algoritm care să elimine fundalul, să transforme imaginea în alb-negru și să uniformizeze luminozitatea;
- Asupra ambelor rezultate a fost aplicat un algoritm care să grupeze chenarele de text pe linii;
- Metrica după care au fost comparate cele două soluții este cea a acurateții textului recunoscut;
- Cele două script-uri folosite pentru a rula cele două soluții se găsesc în anexa A.

În urma a mai multor teste, am observat că *Tesseract* este mai susceptibil la zgomot și nu oferă rezultate de aceeași calitate ca *Firebase Vision*. Un exemplu ușor este imaginea 4.1. Textele extrase de cele două soluții sunt prezentate în figura 4.2, unde se vede cantitatea de zgomot mai ridicată în rezultatul obținut cu *Tesseract*. De asemenea, *Tesseract* a emis alte câteva caractere non-utf8 de zgomot, ce nu au putut fi redate în acest text. 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*.



Figura 4.1: Exemplu bon

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*.

SC AUCHAN ROMANIA SA
HYPERMARCHE AUCHAN
MUN.BUCURESTI,CALEA VITAN,
236 , SECT.3 , BUC
C. I . F. RO17233051
BON FISCAL
ZILNIC 07.00 – 23.00
BINE ATI VENIT!
SCHWEPPES KINLEY TON
2,17 B
1 BUC X 2,17
2,17
TOTAL:
2,17
CARD

—
00033 009 83473 0317
VALOARE
0,18
0,18
TOTAL
TVA
B-TVA 9%
TOTAL TAXE:
2,17

—
VA MULTUMIM CA NE-ATI ALES!
AUCHAN.. SI VIATA SE SCHIMBA
CASIER
CASA
NUMAR BON:
07.06.2019
SOLO1397
9
0217-00060
16:15:05
3000148209
BON FISCAL

SC AUCHAN ROMANIA SA
HYPERMARCHE AUCHAN
—MUN.BUCURESTI CALEA VITAN,
236 , SECT.3 , BUC
C.1.F.: RO17233051

iS neon BON FISCAL = wee
oe TILNIC 07.00 – 23.00
BINE ATI VENIT !

j SCHWEPPES KINLEY TON
1 BUC X 2,17 2,17 B

PO PDO ED PE OD PB PD OD PD PE
OD Od PD OM PD PD OD OD
PD PG ED PD OD LL OE OD LP
AD ODO OD ODED OD OD OE EOE
OD OS OO PO

VA MULTUMIM CA NE-ATI ALES!
AUCHAN... SI VIATA SE SCHIMBA

wie anes CASA: : 9

ecm mgd NUMAR BON: 0217-00060

=–4 07.06.2019 16:15:05
2.3000148209

– BON FISCAL –

TE Sencggeeme

(b) Tesseract OCR Result

(a) Firebase Vision OCR result

Figura 4.2: Textul extras de cele două soluții OCR

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 fată, dar organizarea pe linii este, din moment ce informația din 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 distanță 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. Acest proces este ilustrat și în figura 4.3. Organizarea la nivel de **bloc** este reprezentată în albastru, cea de **linie** în verde, iar cea de **element** în roșu. Se observă că în organizarea dorită se renunță la structurile de blocuri, liniile se extind pe toată lățimea bonului, iar elementele devin fostele liniii.

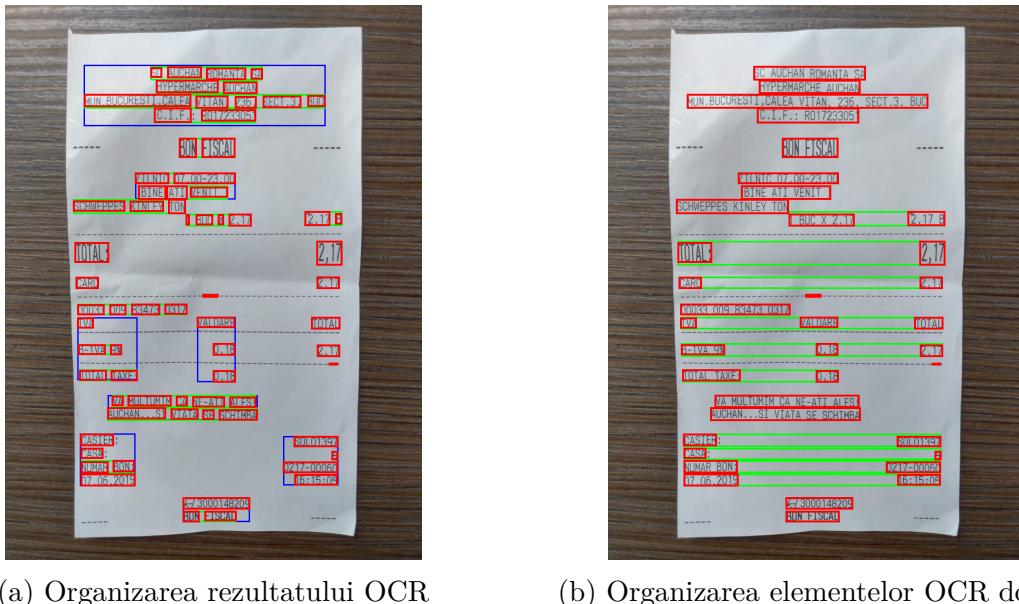


Figura 4.3: Procesul de organizare a rezultatului OCR

Având textul din imagine organizat în grupuri de cuvinte apropiate (vechile linii returnate 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.

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

4.2 Capturarea și înțelegerea imaginilor

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 feedPreview(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 dacă 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 îintrerupe procesarea.

Pentru implementarea vizorului am folosit librăria *CameraView*[4]. O constrângere a acesteia este aceea că imaginea surprinsă este pusă automat pe un *thread* secundar și este accesată printr-un *callback*. Această abordare intră în conflict utilizarea *RxJava*. La o inspecție a codului acestei librării am observat existența unei metode *package private* de a obține imaginea capturată în mod *sincron*. Așadar, într-un modul nou am implementat decoratorul `RxPictureResult` în pachetul librăriei, care să expună imaginea capturată într-un `Observable`.

Procesarea cadrelor pentru a afișa în timp real textul recunoscut de modului *OCR* este limitată la maxim 15 cadre pe secundă pentru a nu suprasolicita dispozitivul. *CameraView* livrează cadre la o rezoluție scăzută pentru procesarea în timp real, ceea ce scade timpul de procesare al acestora, dar și calitatea textului extras. Aceasta nu este o problemă deoarece afișarea în timp real are doar rolul de a ghida utilizatorul. Dacă o porțiune de text este recunoscută la o rezoluție scăzută, atunci aceasta va fi recunoscută și în imaginea capturată la rezoluție întreagă. Pentru afișarea chenarelor am implementat obiectul vizual `OcrOverlay`, care este așezat deasupra vizorului.

4.3 Editare draft

Operațiunile permise asupra unui draft sunt modelate la nivelul domeniului prin interfața `DraftsUseCase`, după cum este prezentată în programul 4.2. 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 acestea.

Programul 4.2: Interfața Drafts Use Case

```

1 interface DraftsUseCase {
2     fun list(): Flowable<List<DraftListItem>>
3     fun fetch(draftId: DraftId): Manage
4
5     interface Manage {
6         val value: Flowable<Draft>
7         val image: Flowable<Bitmap>
8         fun <T> update(newVal: T, mapper: (T, Draft) -> Draft): Completable
9         fun delete(): Completable
10        fun moveToValid(): Completable
11        fun createProduct(): Single<Product>
12        fun updateProduct(product: Product): Completable
13        fun deleteProduct(product: Product): Completable
14    }
15 }
```

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. La nivelul interfeței grafice, câmpurile textuale sunt reprezentate prin câmpuri `EditText`, care permit tastarea în interior. De fiecare dată când utilizatorul face modificări asupra acestor câmpuri, o funcție *callback* este declanșată, care apelează funcția de actualizare a valorii în baza de date.

Pentru ca actualizarea să nu se producă mult prea des, funcțiilor de actualizare le este aplicat operatorul `throttleLast`. Funcționarea acestuia este ilustrată în figura 4.4[12]. Acest operator emite obiecte numai după ce un anumit timp a trecut. Aplicarea acestui operator unei funcții se face prin crearea unui obiect `PublishSubject` și este ilustrată în programul 4.3.

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     }
```

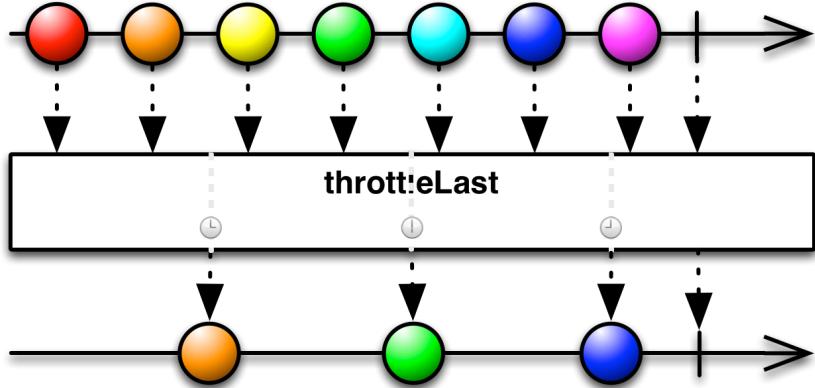


Figura 4.4: Operatorul throttleLast

Baza de date este actualizată cu valori compuse de tipul **Draft**. De aceea funcția `update` primește o nouă valoare singulară și o funcție care are ca argumente noua valoare unitară și vechea valoare **Draft** și emite valoarea compusă nouă, ce este persistată. Implementarea acestei funcții se face folosind funcția `copy` ce este definită automat pentru clase de date în *Kotlin*.

Obiectul vizual din *Android* predefinit pentru afișarea imaginilor nu permite gestul de *zoom*. Pentru a oferi această funcționalitate pentru imaginea draftului curent, am folosit librăria *PhotoView* [1].

4.4 Gestionare setări

Modificarea setărilor nu face parte din domeniul problemei acestei aplicații, de aceea nu are o reprezentare la acest nivel. În schimb, domeniul definește interfețele `ReceiptDefaults` și `CollectingOption`, care conțin valori ce sunt modificate prin intermediul setărilor.

Programul 4.4: Interfețele ReceiptsDefaults și CollectingOption

```

1 interface ReceiptDefaults {
2     val currency: Currency
3     val category: Category
4 }
5
6 interface CollectingOption {
7     val enabled: Boolean
8     val appId: String
9 }
```

Valorile din setări sunt persistate în *shared preferences*, aşa cum este arătat în secțiunea 3.4.3. La nivelul infrastructurii este definită clasa `PreferencesDao` cu scopul de a accesa și persista valorile din *shared preferences*. Tot această clasă implementează cele două interfețe din programul 4.4 și este injectată unde interfețele sunt cerute.

4.5 Colectarea Datelor

La nivelul domeniului, funcționalitatea de colectare de date are o reprezentare simplă, dată de interfața `ReceiptCollector`. Atunci când `DraftsUseCase.Manage`:`:moveToValid` se execută cu succes, metoda `send` este apelată și aceasta declanșează procesul de colectare a bonului curent.

Programul 4.5: Interfața `ReceiptCollector`

```
1 interface ReceiptCollector {  
2     fun send(id: Long)  
3 }
```

Constrângerea acestei funcționalități de a avea un impact minim asupra experienței utilizatorului este rezolvată prin implementarea acesteia ca `Worker`, ceea ce permite executarea întârziată, în *background* și sub anumite condiții. Din moment ce această acțiune nu este critică pentru utilizator, ea se execută numai atunci când dispozitivul este conectat la o rețea *UNMETERED* (Wi-Fi).

Worker-ul primește ca parametru id-ul bonului care trebuie sincronizat în cloud. La instanțiere, îi sunt injectate un obiect de acces la baza de date și opțiunea de colectare, aşa cum este prezentată în programul 4.4. Dacă această opțiune este activată, bonul este extras din baza de date și trimis în cloud, împreună cu imaginea aferentă.

Așa cum este specificat, colectarea se face în mod anonim. Totuși, un identificator care să arate dacă anumite date provin de la același dispozitiv poate fi util. De aceea bonurile sunt trimise împreună cu un *UUID*. Acest identificator este generat prima dată când este cerut și salvat în *shared preferences*. De menționat este că acest identificator nu supraviețuiește la reinstalarea aplicației.

Această funcționalitate, la fel ca cea de export, utilizează serviciile cloud *Firebase*. Autentificarea se face pe baza cheii de aplicație generată din consola *Firebase* atunci când aplicația este atașată unui proiect. Această cheie este salvată în fișierul *google-services.json* și distribuită împreună cu aplicația.

Stocarea datelor textuale se face în colecția *collected* din *Firebase Firestore*, iar imaginile se stochează în *Firebase Cloud Storage*, în directorul *collected*.

4.6 Export

Funcționalitatea de export este modelată prin interfața `ExportUseCase`. Aceasta definește funcțiile de listare a tuturor exporturilor de pe dispozitiv, creare a unui nou export și marcarea unui export ca finalizat la primirea unei notificări.

Programul 4.6: 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 opriate 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 crearea 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 4.5.

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 (optional 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 instanță *Firebase Cloud Functions* este configurată pentru a asculta schimbări ale colecției *manifests* și a se declanșa la crearea 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 folderul *downloads* din Cloud Storage și generează un link de descărcare, pe care îl trimit

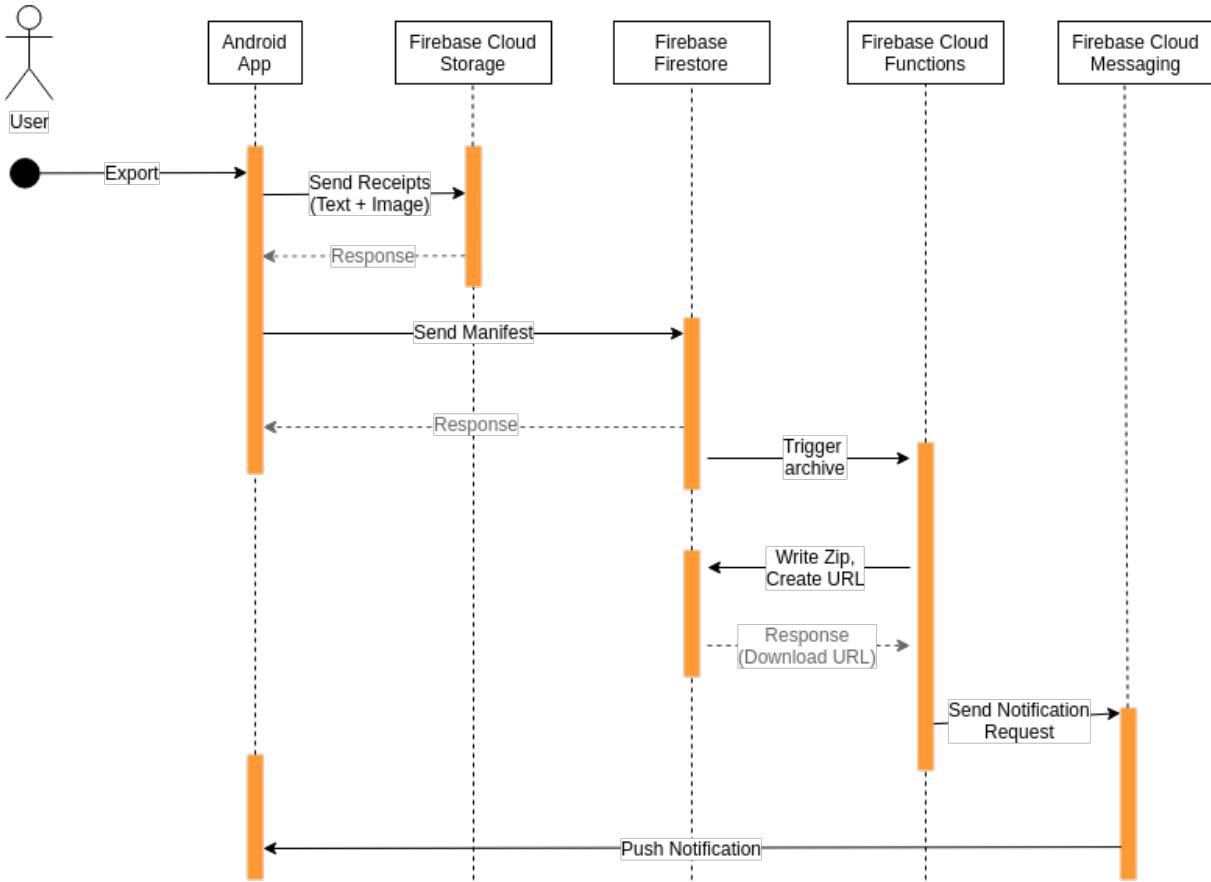


Figura 4.5: Procesul de export

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 folosește clientul Android al *Firebase Cloud Messaging*. Aceasta 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 trimită 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. 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 ajunge pe dispozitiv, metoda `markAsFinished(notification: FinishedNotification)` este apelată pentru a actualiza baza de date și o notificare este afișată.

4.7 Vizualizarea datelor

Funcționalitatea de vizualizare a datelor este modelată de `ReceiptsUseCase`. Pentru a ușura gestionarea și testarea stării, am definit interfața `ISourceManager`, care este implementată separat, iar implementarea interfeței `ReceiptsUseCase` folosește *design pattern-ul* delegării pentru a implementa interfața de gestionare a stării.

Programul 4.7: ReceiptsUseCase

```
1 interface ReceiptsUseCase : ISourcesManager {
2
3     fun fetch(receiptId: ReceiptId): Manage
4
5     interface Manage {
6         val receipt: Flowable<Receipt>
7         fun exportReceipt(): Single<String>
8         fun exportPath(): Single<ImagePath>
9         fun exportBoth(): Single<Pair<String, ImagePath>>
10    }
11 }
12
13 interface ISourcesManager {
14     val availableCurrencies: Flowable<List<Currency>>
15     val availableMonths: Flowable<List<Date>>
16     val categories: Flowable<List<SpendingGroup>>
17     val currentSpending: Flowable<SpendingGroup>
18     val transactions: Flowable<List<ReceiptListItem>>
19
20     fun fetchForCurrency(currency: Currency)
21     fun fetchForMonth(month: Date)
22     fun fetchForCategory(spendingGroup: SpendingGroup)
23 }
24
25 data class SpendingGroup (
26     val group: Group,
27     val total: Float,
28     val currency: Currency
29 )
30
31 sealed class Group {
32     data class Categorized(val value: Category): Group()
33     object Total: Group()
34 }
```

Gestionarea stării folosește o serie de operatori *RxJava* pentru a obține comportamentul dorit. Aceasta trebuie să emită valori noi fie atunci când sursa de date (baza de date *sqlite*) suferă modificări (de exemplu, atunci când o nouă chitanță este inserată), fie atunci când una dintre funcțiile `fetchFor*` este apelată. De asemenea, sursele de date sunt dependente unele de altele, astfel:

1. `availableCurrencies` nu depinde de alte surse de date; această interogare se execută prima;
2. `availableMonths` depinde de monedă; prima monedă din rezultatul interogării precedente este folosită pentru a executa această interogare;
3. `categories` depinde de interogările precedente. Categoriile sunt interogate numai pentru luna și moneda selectată;

4. `currentSpending` depinde de interogarea precedentă; valoarea din interogarea precedentă cu cea mai mare sumă a cheltuielilor dă valoare acestei interogări;
5. `transactions` depinde de monedă, luna selectată și categoria selectată;

Pentru ca starea să fie actualizată corespunzător, următoarele tehnici sunt folosite:

1. Fiecarei funcții `fetchFor*` îi corespunde un *BehaviorProcessor*, capabil să păstreze ca stare ultima valoare emisă. Atunci când funcția este apelată, emite argumentul în acest procesor.
2. Fiecare procesor este combinat folosind operatorul `mergeWith` cu interogarea din baza de date aferentă pentru a obține actualizări atunci când sursa de date se modifică; combinației îi sunt aplicăți operatorii `.replay(1).autoConnect()` pentru a reține ca stare ultima valoare emisă; Aceste combinații devin surse de parametrii;
3. Pentru interogările dependente, sursele de parametrii sunt combinate folosind operatorul `combineLatest` pentru a emite noi valori de fiecare dată când una dintre sursele unitare este actualizată; apoi aceste combinații sunt redirecționate către interogările din baza de date;

Din punct de vedere vizual, acest ecran folosește 3 *RecyclerViews* orizontale pentru a selecta moneda, categoria și luna dintr-un carusel și un *RecyclerView* vertical pentru lista de tranzacții. Selectorii pentru lună și monedă au ca element activ pe cel din mijloc și pentru a discretiza scroll-ul pentru aceștia este folosită clasa predefinită `LinearSnapHelper`. În schimb, selectorul de categorie are ca element activ pe cel din stânga. Pentru a obține efectul de *snap* în acest caz, am implementat clasa `StartSnapHelper`.

Din moment ce acest ecran reutilizează același comportament de carusel vertical pentru selectorii menționați mai sus, am implementat clasa abstractă `HorizontalCarousel` și diferite implementări pentru interfața `PositioningStrategy`, ce are ca scop modificarea comportamentului la poziționare a clasei `HorizontalCarousel`.

Funcționalitățile asociate unui singur bon validat sunt modelate în subinterfața `Manage`, unde sunt definite cele trei tipuri de export din specificații. Pentru ca datele să fie preluate de aplicații externe, rezultatele acestor funcții sunt atașate unor obiecte `Intent`. Aplicațiile care acceptă un intent configurat cu acele date sunt vizibile în ecranul de selectare a aplicației sănătoase.

Funcția ce lansează intent-ul pentru a trimite atât imaginea, cât și textul, este prezentată în programul 4.8. Câmpul de acțiune face ca acest intent să fie consumat de aplicații externe care acceptă tipul setat. Acestea pot consuma oricare dintre datele *extra* sau *parcelable*.

Programul 4.8: Lansare intent

```
1 | private val bothExporter = { text: String, imageUri: Uri ->
```

```

2     Intent().apply {
3         action = Intent.ACTION_SEND_MULTIPLE
4         type = "image/jpeg"
5         putParcelableArrayListExtra(Intent.EXTRA_STREAM, arrayListOf(imageUri))
6         addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)
7         putExtra(Intent.EXTRA_TEXT, text)
8         putExtra(Intent.EXTRA_SUBJECT, "Receipt from ${viewModel.merchant.value}")
9     }.let(::startActivity)
10 }
```

4.8 Testare automată

Pentru implementarea de teste automate, structura codului are o mare importanță. Arhitectura implementată valorifică decuplarea elementelor, ceea ce asigură și ușurința în a le testa. De asemenea, injectarea dependințelor facilitează înlocuirea acestora cu obiecte *mock*, ce au un comportament definit în cadrul testului, pentru a testa modul în care funcționează o componentă în orice caz.

În cadrul acestui proiect am implementat o suită de teste unitare pentru a testa logica de la nivelul domeniului. De asemenea, am structurat codul în aşa fel încât marea parte a computațiilor ce pot fi greșite să se întâpte la nivelul domeniului, pentru a reduce nevoia de a testa celelalte nivele.

Pentru rularea testelor am folosit librăria *JUnit 4*, iar pentru a modela dependințele componentelor testate am folosit *Mockito* și extensia pentru limbajul *Kotlin - Mockito-Kotlin*. De asemenea, o mare importanță a avut-o suportul din *RxJava* pentru teste. Obiectele din *RxJava* oferă metoda `.testObserver()`, care permite testarea codului asincron într-un mod clar și intuitiv.

Structurarea specificațiilor în modul semi-formal permite, pe lângă modelarea codului, și definirea testelor într-un mod ușor de înțeles. De exemplu, specificația 2.6 *"Utilizatorul selectează o nouă monedă, categorie sau lună, iar datele sunt actualizate;"* se traduce în testul 4.9.

Programul 4.9: Test Emisie

```
1 @Test
2     fun 'when fetching new currency, values are emmited'() {
3         val repo: ReceiptsRepository = mock {
4             on { getAvailableCurrencies() } doReturn Flowable.just(currencies).subscribeOn(Schedulers.io())
5         }
6         val subject = SourcesManager(repo)
7         subject.availableCurrencies.test()
8         val currencyObs = subject.currentCurrency.test()
9
10        subject.fetchForCurrency(otherCurrency)
11
12        currencyObs.awaitCount(2).assertValues(currencies[0], otherCurrency)
13
14        subject.fetchForCurrency(otherCurrency)
15
16        currencyObs.awaitCount(1).assertValues(currencies[0], otherCurrency, otherCurrency)
17    }
```


Concluzii

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 întâmpinat în dezvoltarea aplicației a fost întelegerea automată a bonurilor fiscale. Abordarea aleasă are două mari dezavantaje. Primul este acela că performanța ei este limitată de performanța modulului *OCR* ales. Al doilea dezavantaj este lipsa flexibilității și mai ales imposibilitatea ca extragerea informațiilor să se îmbunătățească fără a face modificări în codul aplicației.

Funcționalitatea de export o găsesc foarte utilă pentru urmărirea cheltuielilor personale. Câteva întrebări la care se poate răspunde având la îndemână nu doar lista tranzacțiilor, ci și produsele de pe fiecare chitanță sunt: *Ce sumă am cheltuit în această lună pe apă, pâine etc.?, Cu cât sunt mai scumpe produsele la supermarket-ul X comparativ cu Y?*. De asemenea, aceste date sunt și o înregistrare a evoluției prețurilor de-a lungul timpului. Adunarea acestor date manual este o sarcină laborioasă, pe care puțini o fac. Receipt-Scan este o unealtă care să faciliteze această sarcină.

Din punct de vedere al implementării, limbajul *Kotlin* aduce o experiență de dezvoltare mult mai plăcută comparativ cu *Java 7*. Librăriile din cadrul *Android Architecture Components* încearcă să ofere soluții moderne pentru structurarea și implementarea aplicațiilor. Totuși, menținerea compatibilității cu versiunile vechi ale *API-urilor Android* limitează gradul de dezvoltare al acestora. Un *bug* rezolvat recent care evidențiază această problemă este păstrarea subiecției unui fragment la un *viewModel* după ce acest fragment și-a încheiat ciclul de viață, ceea ce conduce la *memory leaks*. Privind retrospectiv, poate că *Flutter*, un *framework* mult mai nou, dezvoltat în limbajul *Dart* ar fi fost o alegere mai

bună pentru implementarea acestei aplicații.

O experiență plăcută am avut lucrând cu *RxJava*. Această librărie are o complexitate ridicată și necesită un efort considerabil din partea programatorului pentru a o înțelege. Avantajul primit în schimb este ușurința cu care se poate executa cod asincron. De asemenea, abordarea funcțională a acestei librării face testarea mai ușoară și reduce din posibilele *bug-uri*.

Dezvoltarea ulterioară a acestei aplicații vizează publicarea acesteia și promovarea pe medii online, cum ar fi *Reddit* sau *ProductHunt*. De asemenea, această aplicație este în mod *open source* și poate aduna contribuții de la mai mulți programatori, care pot îmbunătăți metoda de extragere a informațiilor din chitanțe.

Anexe

Anexa A

Script-urile folosite pentru compararea soluțiilor OCR

Codul testului rulat pentru a obține textul din 4.2a:

Programul A.1: LineUnification.kt

```
1 @RunWith(AndroidJUnit4::class)
2 class OcrCompareTest {
3
4     @Test
5     fun runOcr() {
6         val image = readImage()
7         val recognizer = FirebaseVision.getInstance().onDeviceTextRecognizer
8         val firebaselImage = FirebaseVisionImage.fromBitmap(image)
9         val result = recognizer.processImage(firebaselImage).let { Tasks.await(it) }
10        println(result)
11    }
12
13
14    private fun readImage(): Bitmap {
15        val context = InstrumentationRegistry.getInstrumentation().context
16        val imageStream = context.assets.open("resizedReceipt.jpg")
17        return BitmapFactory.decodeStream(imageStream)
18    }
19 }
```

Script-ul rulat pentru a obține textul din 4.2b:

Programul A.2: LineUnification.kt

```
1 import cv2 as cv
2 import numpy as np
3 import pytesseract as tes
4
5
6 text = get_text_from_image("resizedReceipt.jpg")
7 print(text)
8
9
10 def get_text_from_image(imageName):
11     img = preprocess(imageName)
12     result = tes.image_to_string(img)
```

```

13     return result
14
15
16 def preprocess(image_name):
17     image = cv.imread(image_name)
18     gray = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
19     receiptBox = find_receipt_box(gray)
20     M, w, h = perspective_transform(receiptBox)
21     receiptImg = apply_perspective_correction(gray, M, w, h)
22     receiptImg = cv.adaptiveThreshold(receiptImg, 255, cv.ADAPTIVE_THRESH_GAUSSIAN_C, cv.
23         THRESH_BINARY, 71, 10)
24     return receiptImg
25
26 def find_receipt_box(image):
27     """
28         Finds a contour around the receipt in the given image.
29         Returns the bounding box and the binary image
30     """
31     # gray = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
32     gray = cv.medianBlur(image, 15, 0)
33     _, thresh = cv.threshold(gray, 255, 125, cv.THRESH_BINARY | cv.THRESH_OTSU)
34     k = np.ones((25, 25))
35     thresh = cv.erode(thresh, k, iterations=1)
36     thresh = cv.dilate(thresh, k, iterations=1)
37     contours = cv.findContours(thresh, cv.RETR_LIST, cv.CHAIN_APPROX_SIMPLE)
38     contours = sorted(contours[0], key=cv.contourArea, reverse=True)
39     contour = contours[0]
40     rect = cv.minAreaRect(contour)
41     box = cv.boxPoints(rect)
42     box = np.int0(box)
43     return box
44
45
46 def perspective_transform(contour):
47     """
48         Produces the transformation matrix and the new size for perspective correction
49     """
50     ord_rect = np.float32(order_rect(contour))
51     (tl, tr, br, bl) = ord_rect
52
53     dist_top = np.linalg.norm(tl - tr)
54     dist_btm = np.linalg.norm(bl - br)
55     width = max(dist_btm, dist_top)
56
57     dist_left = np.linalg.norm(tl - br)
58     dist_right = np.linalg.norm(tr - bl)
59     height = max(dist_left, dist_right)
60
61     dest_corners = np.array([
62         [0, 0],
63         [width - 1, 0],
64         [width - 1, height - 1],
65         [0, height - 1]
66     ], dtype=ord_rect.dtype)
67
68     M = cv.getPerspectiveTransform(ord_rect, dest_corners)
69     return M, width, height
70
71
72 def order_rect(pts):
73     """
74         orders a rectangle in the order top-left, top-right,
75         bottom-right, bottom-left
76     """
77     new = np.zeros((4, 2), dtype="int64")
78     s = pts.sum(axis=1)
79     new[0] = pts[np.argmin(s)]
    new[2] = pts[np.argmax(s)]

```

```
80     diff = np.diff(pts, axis=1)
81     new[1] = pts[np.argmin(diff)]
82     new[3] = pts[np.argmax(diff)]
83
84     return new
85
86
87 def apply_perspective_correction(image, M, width, height):
88     """Crops the contour and applies perspective correction"""
89     warped = cv.warpPerspective(image, M, (width, height))
90
91     return warped
```


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) {
```

```
46     currentLine.add(crtBox)
47 } else {
48     val sortedLine = currentLine.sortedBy { it.left }
49     unifiedLines.add(
50         Line(
51             sortedLine
52         )
53     )
54     currentLine = ArrayList()
55     currentLine.add(crtBox)
56 }
57 lastBox = crtBox
58 }
59 }
60 if (currentLine.isNotEmpty()) {
61     unifiedLines.add(
62         Line(
63             currentLine
64         )
65     )
66 }
67 }
68 return unifiedLines
70 }
```

Anexa C

Algoritmul de extragere a informațiilor

Programul C.1: Algoritmul de extragere

```
1 typealias OcrElements = Sequence<OcrElement>
2
3 class Extractor @Inject constructor(
4     private val defaults: ReceiptDefaults
5 ) {
6     operator fun invoke(elements: OcrElements): Draft {
7         val receipt = RawReceipt.create(elements)
8         val text = receipt.text
9         val merchant = extractMerchant(receipt)
10        val date = extractDate(text)
11        val currency = defaults.currency
12        val category = defaults.category
13        val (total, products) = ProductsAndTotalStrategy(
14            receipt
15        ).execute()
16        return Draft (
17             merchant,
18             date,
19             total,
20             currency,
21             category,
22             products.map { Product(it.name, it.price) },
23             elements.map { OcrElement(it.text, it.top, it.bottom, it.left, it.right) }.toList()
24         )
25     }
26 }
27
28 class RawReceipt(private val lines: List<Line>) : Iterable<RawReceipt.Line> {
29     override fun iterator(): Iterator<Line> = lines.iterator()
30
31     class Line(private val elements: List<OcrElement>) : Iterable<OcrElement> {
32         override fun iterator() = elements.iterator()
33         val text by lazy { elements.joinToString(" ") { it.text } }
34         val height by lazy { elements.map { it.height }.average() }
35         val top by lazy { elements.map { it.top }.min()!! }
36         val bottom by lazy { elements.map { it.bottom }.max()!! }
37     }
38
39     val averageLineHeight by lazy { this.lines.map { it.height }.average() }
40 }
```

```

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
private const val MERCHANT_MIN_LENGTH = 2
88
89 fun extractMerchant(rawReceipt: RawReceipt): String? {
90    val linesIterator = rawReceipt.iterator()
91    while (linesIterator.hasNext()) {
92        val line = linesIterator.next()
93        if (line.text.length < MERCHANT_MIN_LENGTH) continue
94
95        val nextLine = if (linesIterator.hasNext()) linesIterator.next() else null
96
97        val heightThreshold = 1.2 * rawReceipt.averageLineHeight
98
99        return if (
100            line.height > heightThreshold &&
101            nextLine != null &&
102            nextLine.text.split(" ").size < 2 &&
103            nextLine.height > heightThreshold &&
104            nextLine.top - line.bottom < rawReceipt.averageLineHeight
105        ) {
106            line.text + " " + nextLine.text
107        } else
108    }

```

```

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) {

```

```

177     val left = isAlignedToLeft(element)
178     val right = isAlignedToRight(element)
179     if (left && right) {
180         processKeyValue(element)
181     } else if (left) {
182         processKey(element)
183     } else if (right) {
184         processPrice(element)
185     }
186 }
187 }
188 }
189
190 private fun processPrice(priceElement: OcrElement) {
191     val price = parseNumber(priceElement.text)
192     price?.let {
193         val mLastKey = lastKey
194         if (mLastKey is Just) {
195             val keyElement = mLastKey.value
196             if (priceElement.top - keyElement.bottom < 0.5 * priceElement.height) {
197                 makeResult(keyElement, it)
198             } else {
199                 lastKey = None
200             }
201         }
202     }
203 }
204
205 private fun processKey(element: OcrElement) {
206     val digitCount = element.text.count { it.isDigit() }
207     if (digitCount < 0.3 * element.text.length) {
208         lastKey = Just(element)
209     }
210 }
211
212 private fun processKeyValue(element: OcrElement) {
213     val price = parseNumber(element.text)
214     price?.let {
215         val name = element.text.split(" ").take(3).joinToString(" ")
216         if (name.length > 1) {
217             makeResult(name, price, element)
218         }
219     }
220 }
221
222 private fun makeResult(key: String, price: Float, element: OcrElement) {
223     val keyLowercase = element.text.toLowerCase()
224     if (keyLowercase.contains(totalMarkRegex)) {
225         keyPriceResults.add(
226             ResultObj.Total(
227                 price,
228                 element.top
229             )
230         )
231     } else {
232         keyPriceResults.add(
233             ResultObj.Product(
234                 price,
235                 key,
236                 element.top
237             )
238         )
239     }
240 }
241
242 private fun makeResult(keyElement: OcrElement, price: Float) {
243     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",

```

```

311     "$DIGIT_MISTAKES{4}--$DIGIT_MISTAKES{2}--$DIGIT_MISTAKES{2}" to "yyyy-MM-dd",
312     "$DIGIT_MISTAKES{4}\\.$DIGIT_MISTAKES{2}\\.$DIGIT_MISTAKES{2}" to "yyyy.MM.dd"
313 )
314
315 fun findDatesWithPatterns(searchedString: String): Sequence<Date> {
316     var results = sequenceOf<Pair<String, String>>()
317     val fakeZeros = "Oo".toRegex()
318     for ((regex, format) in formats) {
319         val result = regex.toRegex().findAll(searchedString)
320         val seq = result.map { it.value.replace(fakeZeros, "0") to format }
321         results += seq
322     }
323
324     val now = Date()
325
326     return results
327         .mapNotNull {
328             try {
329                 SimpleDateFormat(it.second, Locale.US).parse(it.first)
330             } catch (e: ParseException) {
331                 null
332             }
333         }
334         .sortedBy { abs(it.time - now.time) }
335 }
```

Bibliografie

- [1] Chris Banes. *Photo View*. URL: <https://github.com/chrisbanes/PhotoView/> (visited on 09/07/2019).
- [2] Alistair Cockburn. “Structuring use cases with goals”. In: *Journal of Object-Oriented Programming* 10.5 (1997), pp. 56–62.
- [3] 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.
- [4] Mattia Iavarone. *CameraView Documentation*. URL: <https://natario1.github.io/CameraView/> (visited on 09/07/2019).
- [5] Bill Janssen et al. “Receipts2Go: The big world of small documents”. In: Sept. 2012, pp. 121–124. DOI: 10.1145/2361354.2361381.
- [6] Elisabeth Freeman Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, 2004. Chap. Chapter 2: the Observer Pattern, pp. 37–78.
- [7] 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.
- [8] *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).
- [9] *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).
- [10] *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).
- [11] Rizlène Raoui-Outach et al. “Deep Learning for automatic sale receipt understanding”. In: (Dec. 2017).
- [12] ReactiveX. *Observable documentation*. URL: <http://reactivex.io/RxJava/javadoc/io/reactivex/Observable.html#throttleLast-long-java.util.concurrent.TimeUnit-> (visited on 08/28/2019).

- [13] Tesseract. *Tesseract Project*. URL: <https://github.com/tesseract-ocr/tesseract> (visited on 08/28/2019).
- [14] 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.