FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Mobilní aplikace pro rozpoznávání obsahu účtenek
Student:	Patrik Mokriš
Vedoucí:	Ing. Tadeáš Sosín
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Cílem práce je tvorba prototypu aplikace pro chytré skenování účtenek s využitím Google ML Kit SDK.

Proveďte analýzu případných konkurenčních řešení. Proveďte analýzu funkčních i systémových požadavků a na jejich základě vytvořte low-fidelity prototyp. Navrhněte nástroje a jiné produkty třetích stran potřebné pro realizaci aplikace a vnitřní architekturu aplikace. Navrhněte způsob testování a následného nasazení do produkce. Navrhněte prostředí pro podporu provozu aplikace, které umožní její další rozvoj a podporu stávajících uživatelů. Po dohodě s vedoucím práce realizujte prototyp mobilní aplikace určený pro zařízení s operačním systémem Android.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 5. května 2022 v Praze.

Bakalářská práce

MOBILNÍ APLIKACE PRO ROZPOZNÁVÁNÍ OBSAHU ÚČTENEK

Patrik Mokriš

Fakulta informačních technologií
Katedra softwarového inženýřství
Vedoucí: Ing. Tadeáš Sosín
7. února 2023

Odkaz na tuto práci: Mokriš Patrik. *Mobilní aplikace pro rozpoznávání obsahu účtenek.* Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Obsah

Seznam obrázků

Seznam výpisů kódu

Chtěl bych poděkovat především Ing. Tadeáši Sosínovi za vedení této práce, pravidelné konzultace a mnoho přínosných poznatků. Také bych rád poděkoval rodině a přátelům za podporu během studia a náročných chvílích.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 7. února 2023 .....

Abstrakt

Tato bakalářská práce se zabývá návrhem mobilní aplikace pro skenování účtenek, rozpoznání jejich obsahu a jejich správou. Součástí aplikace je vyexportování těchto záznamů do účetního reportu ve formátu CSV či PDF. Výstupem práce je implementace prototypu pro operační systém Android. Aplikace je navržena semi-multiplatformním přístupem pomocí KMM a stojí na principech Clean architektury. Pro rozpoznání textu z obrazu je použita knihovna ML Kit od společnosti Google. Rozpoznávání obsahu účtenek v prototypu je implementováno vyhledáváním shod v analyzovaném textu pomocí regulárních výrazů. Přínosem bakalářské práce je zjednodušení procesu měsíčního vyúčtování výdajů firmy Matee Devs s.r.o. a správy účtenek s ním související.

Klíčová slova mobilní aplikace, Android, skenování účtenek, správa účtenek, finanční report, textová analýza, Clean architektura, Google ML Kit, KMM, Jetpack Compose

Abstract

This bachelor's thesis is about design of a mobile application for scanning receipts, recognizing their content and managing them. Exporting these records to an accounting report in CSV or PDF format is part of the application. Output this thesis is implementation of a prototype for the Android operating system. The application is designed with a semi-multiplatform approach using KMM and is based on the principles of Clean architecture. To recognize text from an image, the ML Kit library from Google is used. Receipt content recognition in the prototype is implemented by searching for matches in the analyzed text using regular expressions. This bachelor's thesis brings a simplification to a process of monthly accounting of Matee Devs s.r.o. expenses and receipts management related to it.

Keywords mobile application, Android, receipts scan, receipts management, finance report, text analysis, Clean architecture, Google ML Kit, KMM, Jetpack Compose

Seznam zkratek

API	Application programming interface
CSV	Comma-separated values
DI	Dependency injection
FP	Funkční požadavek
IDE	Integrated Development Environment
KMM	Kotlin Multiplatform Mobile
ML	Machine learning
NP	Nefunkční požadavek
PDF	Portable Document Format
SQL	Structured Query Language
UC	Use case
UI	User interface
UX	User experience

Úvod

V roce 2022 je běžné, že vývojáři s dobrým nápadem se místo jeho sdílení rozhodnou ho uchopit do vlastních rukou a realizovat se založením vlastní firmy. V opačném případě by mohli příležitostně dosáhnout pouze částečného podílu z potenciálně úspěšného projektu. To ale sebou nese nadbytečné procesy a povinnosti nad rámec vývoje vysněné idey – například účetnictví a řízení měsíčních nákladů. Tuto obtíž je možné vyřešit investováním do existujícího systému, který využívají větší firmy, ale jde o risk, zda se investice v počátku vyplatí. A jelikož jsou vývojáři vynalézaví, dokáží si navrhnout vlastní proces řešící daný problém s minimálním úsilím.

Takto samostatně si poradili vývojáři firmy Matee Devs s. r. o. Od vzniku jejich firmy až do nedávna bylo řešení dostačující, dokud se výdaje, napříč rostoucím počtem zaměstnanců, ve formě papírových účtenek nezačaly střádat. Dočasně tento problém vyřešil jejich sběr na jedno určité místo a na konci měsíce pomocí skeneru a cloudového úložiště je převést do virtuální podoby. To ale opět stálo drahocenný čas. A tak vznikla myšlenka si pomoct mobilní aplikací, jelikož smartphone mají zaměstnanci vždy u sebe a účtenku by šlo odbavit ihned při převzetí. Tímto by se zbavili nutnosti je fyzicky uchovávat.

Problém správy účtenek řeší podnikatelé už řadu let a řešení přes mobilní aplikace existují, ale žádné volné na trhu neuspokojovalo plně požadavky zaměstnanců. Hlavním požadavkem bylo, aby načtení účtenky bylo rychlé a snadné. Bylo třeba, aby aplikace rozpoznala obsah dané účtenky (zejména ty podstatné údaje potřebné pro účetní) s co nejmenší nutností vyplňování formulářů zaměstnancem. Dalším podstatným kritériem byla možnost účtenku zpracovat i v režimu bez připojení k internetu. Tím byla většina mobilních aplikací nevyhovujících právě z důvodů zdlouhavého procesu při skenování účtenky či zpracování údajů na bázi zasílání na server.

Z těchto důvodů se začala firma zajímat o vlastní řešení. Ke čtení textu z obrazu (v tomto případě účtenek) už měl vedoucí práce zvolenou volně dostupnou knihovnu. Konkrétně knihovnu ML Kit od společnosti Google s lehkou integrací do mobilních zařízení podávající dobré výsledky. Bylo tedy potřeba vyřešit zpracování výsledku čtení a navrhnout aplikaci řešící zbytek procesu od vyfocení fyzické účtenky přes vytváření výkazu (reportu) pro dané období až po jeho exportování, které dostane firemní účetní.

Cíle práce

Cílem bakalářské práce je tvorba prototypu aplikace pro chytré skenování účtenek s využitím knihovny Google ML Kit SDK pro rozpoznání textu v obrazu.

Prvním z postupových cílů je provedení analýzy případných konkurenčních řešení. Na základě toho a potřeb zaměstnanců firmy Matee Devs s.r.o. provést analýzu funkčních a systémových požadavků aplikace a zadefinovat je. Dle nich vytvořit low-fidelity prototyp, který zaměstnanci otestují, a provést změny podle jejich zpětné vazby. Dále je třeba navrhnout nástroje a jiné produkty třetích stran potřebné pro realizaci aplikace a vnitřní architekturu aplikace. Pro zajištění stability a ověření funkčnosti navrhnout způsob testování. Následně zvolit prostředí pro nasazení do produkce a podporu provozu aplikace, které umožní její další rozvoj a podporu stávajících uživatelů. Finálně realizovat prototyp mobilní aplikace určený pro zařízení s operačním systémem Android a otestovat jej.

Přínosem bakalářské práce je zjednodušení procesu měsíčního vyúčtování výdajů firmy Matee Devs s.r.o. a správy účtenek s ním související.

Kapitola 1
Analýza

Tato kapitola se zaobírá fází analýzy. V softwarovém inženýrství je to proces, který má za cíl úplné porozumění projektu či problému. V této fázi se analyzují potřeby zákazníka, požadavky, specifikace, překážky a rizika. Získané poznatky usnadňují průběh dalších fází.

1.1 Analýza domény

1.1.1 Definice účtenky

Jelikož se náš projekt zaměřuje na extrahování informací z účtenek a jejich správou pro ulehčení starostí s účetnictvím, je důležité znát co vlastně taková účtenka je a jaké informace dle zákona musí obsahovat. Účtenka je formálně definovaná jako daňový doklad, což je účet za provedenou práci nebo dodané zboží, který musí mít náležitosti stanovené § 26-35 zákona o DPH.

1.1.2 Analýza obsahu účtenek

Podle způsobu platby a výše zdanitelného plnění existují dva typy daňových dokladů:

1.1.2.1 zjednodušený daňový doklad

Lze ho použít pouze při nákupu za hotovost nebo při placení kartou či šekem, a to do 10 000 Kč včetně DPH. Vzorovou účtenku podle zákona o evidenci tržeb je možné vidět na obrázku 1.1. Doklad musí obsahovat následující položky:

1. fiskální identifikační kód (FIK),
2. daňové identifikační číslo poplatníka (DIČ, popřípadě IČ),
3. označení provozovny,
4. označení pokladního zařízení,
5. pořadové číslo účtenky,
6. datum a čas přijetí tržby, nebo čas vystavení účtenky (pokud je vystavena dříve),
7. celkovou částku tržby,
8. bezpečnostní kód poplatníka,
9. údaj o způsobu evidence – běžný, nebo zjednodušený režim.[1]

1.1.2.2 běžný daňový doklad

Používá se ve všech ostatních případech. Běžný daňový doklad obsahuje všechny náležitosti zjednodušeného daňového dokladu a navíc obsahuje:

1. identifikační údaje o příjemci zdanitelného plnění (kupujícím),
2. základ DPH a vyčíslenou DPH. [2]

1.1.2.3 Význam jednotlivých údajů

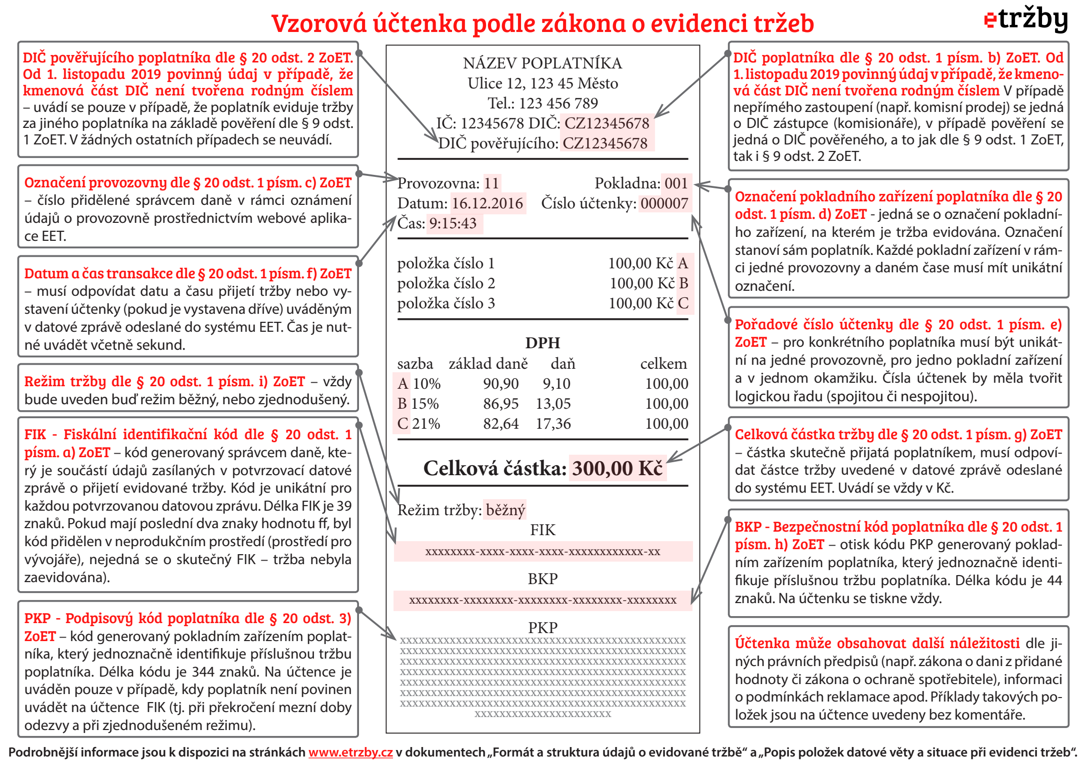
1. Fiskální identifikační kód – FIK je unikátní označení každé účtenky, které generuje Finanční správa v okamžiku, kdy odešlete datovou zprávou údaje o tržbě. Tento kód obdržíte zpravidla do několika sekund. Fiskální identifikační kód se poté zobrazí také na účtence, kterou dostane zákazník. Celkem má 39 znaků. Pokud by se stalo, že poslední dva mají hodnotu „ff“, pak to znamená, že tržba nebyla zaevidována, ale že vznikla ve vývojářském prostředí. Nejedná se tak o skutečný FIK.
2. Daňové identifikační číslo – DIČ naleznete na svém daňovém přiznání. U fyzických osob se používá jako DIČ rodné číslo, u právnických osob pro tyto účely poslouží IČ (identifikační číslo).
3. Označení provozovny – Jedná se o identifikační číslo, které váš e-shop coby provozovna dostane po zaevidování na portálu Daňové evidence.
4. Označení pokladního zařízení – Jedná se o číslo zařízení, na kterém probíhá evidence tržeb. Číslo pokladního zařízení získáte po zaevidování pokladny. Pokud má provozovatel více pokladních zařízení na jedné prodejně, pak každé musí mít své unikátní číslo.
5. Pořadové číslo účtenky – Číslo, které se uvádí v účetnictví. Pro každého provozovatele a systém je to unikátní číslo.
6. Datum a čas přijetí tržby – Na účtenku je nutno uvést úplné a přesné datum a čas (včetně sekund) přijetí tržby. Pokud je účtenka vystavena dřív, zapisuje se na ni datum a čas vystavení.
7. Celková částka tržby – Pro potřeby Finanční správy není nutné na účtenku uvádět celý výpis zakoupených položek. Postačí celková výše tržby, která musí být uvedena v Kč.
8. Bezpečnostní kód poplatníka – BKP je kód, který vygeneruje váš systém. Udává vazbu mezi vámi coby poplatníkem a vystavenou účtenkou. Identifikuje tak evidovanou tržbu. Jedná se vlastně o otisk hodnoty Podpisového kódu poplatníka.
9. Údaj o způsobu evidence – Zde provozovatel uvede, v jakém režimu své tržby eviduje. U e-shopů bude na tomto místě zapsán režim běžný. [1]

1.1.3 Analýza formátu účtenek

Oproti obsahu, jehož položky jsou jasně dané zákonem, tak formát je volně na prodejci. Klasická účtenka je typicky bílý termopapír, který oproti běžnému papíru zajišťuje delší uchování čitelných informací, s černým písmem. Jejich obsah je členěn do 3 částí – hlavička, výčet položek a spodní část, kde jsou všechny ostatní informace. Hlavička obsahuje informace o obchodní firmě a provozovně, pořadové číslo účtenky a datum. Ve výčtu nalezneme jednotlivé položky, jejich počet a cenu. To je ale víceméně vše, co v kontextu formátu mezi sebou účtenky sdílí v důsledku absence nařízeného či respektovaného standardu. Některé mají bohatou hlavičku, kde nalezneme mnoho informací i s jejich označením, některé naopak velmi zkratkovitou jen aby dané informace

obsahovaly. Ani celkovou sumu nenalezneme z pravidla pod výčtem položek. Když se podíváme do vzorové účtenky, tak některé firmy preferují mít zbytek povinných údajů chybějících v hlavičce právě pod výčtem a celkovou částku s údaji o platbě až pod nimi. Tím se pro aplikaci čitelnost těchto informací velmi různí, jelikož se nedá na kostru dat spoléhat.

Obrázek 1.1 Vzorová účtenka podle Zákona o evidenci tržeb [3]



1.2 Analýza existujících řešení

Problém uchovávání a správy účtenek trápí mnoho firem již dlouhou dobu, tudíž pro něj existuje již celá řada řešení. Na základě popularity a doporučení přímo od Google Play je pár těchto řešení představeno a zhodnoceno. Popis těchto aplikací vychází z vlastní analýzy a dostupných informací na jejich stránce v Google Play, která je u každé aplikace citována. Dále poslouží jako inspirace pro návrh a sestavení požadavků cílové aplikace. Výstup analýzy je aktuální k datu 28. 5. 2022. Pořadí řešení odpovídá počtu stažení ze zmíněného obchodu.

Představení a zhodnocení každé aplikace rozčlením do těchto odstavců:

I úvod,

II spuštení, vzhled a přehlednost,

III průběh skenování účtenky,

IV sada dalších funkcionalit,

V cena,

VI zhodnocení.

1.2.1 Expensify

Expensify je americká firma založena roku 2008 snažící se o komplexní a robustní řešení celkových financí zákazníka s hlavním zaměřením na desktopové a webové rozhraní. Její mobilní aplikace je jen odnož celého systému pro skenování a zpracování dokladů. Na Google Play má přes 1 milion stažení. [4]

Při spuštění je uživatel vyzván k přihlášení do Expensify účtu – bez toho aplikaci nelze používat. Na první pohled působí aplikace uhlazeně s využitím tří barev unifikující jejich značku a produkty. Dobře se v ní orientuje, leč pro navigaci využívá *hamburger menu*[1], což v kontextu mobilních aplikací působí zastarale.

Záznam účtenky lze vytvořit v sekci Expenses, kde si uživatel může vybrat ze tří přístupů. Vyfocením, nahráním z galerie či manuálním zadáním. Při focení aplikace nijak neupozorňuje uživatele, jestli je v obraze nějaká účtenka rozeznána. Nahraný obrázek není možné už nijak editovat. Následně se fotografie účtenky nahraje na server, kde probíhá zpracování nazývané *SmartScanning*, tedy dostupné pouze v režimu online. Aplikace oznamuje uživateli, že se výsledek dozví „in short while“ (v krátkém okamžiku), nicméně trvá v průměru 10-15 minut. Po doběhnutí se v detailu účtenky objeví zpracované údaje a je možné je ručně doplnit. Záznamy účtenek obsahují pouze pár parametrů bez možnosti si dodatečně nějaký přidat.

Mezi další funkcionality patří vytvoření reportu ze záznamů účtenek, vygenerování linku do systému Expensify nebo ve formátu PDF s možností zaslání přes email nebo SMS. Neobsahuje v sobě žádné statistiky, grafy, přehledy, filtrování a záznamy účtenek či reportů nelze spravovat (záložkovat, oštítkovat, mazat). Toto je dostupné v rámci webového prostředí.

Stažení a používání aplikace je zdarma. Nový uživatel má k dispozici 30 krát zpracování pomocí *SmartScanning*, dále je zpoplatněna tarifem 5$ měsíčně.

Pokud uživatel využívá běžně celý systém Expensify pro správu svých financí a primárně se pohybuje v desktopovém nebo webovém prostředí, aplikace plní dobře svůj účel jakožto způsob nahrání účtenek. Podařilo se zpracovat pomačkanou i špatně čitelnou účtenkou a sdílení reportů v rámci systému Expensify je velmi snadné. Nicméně krom zaměření na design neobsahuje takovou sadu funkcionalit a uživatelé mají na výběr lepší řešení.

1.2.2 Smart Receipts

Mimo stránky ohledně ceníku toho o firmě či týmu Smart Receipts moc není dohledatelné, nicméně v žebříčcích aplikací, se svými 500 tisíci staženími, se řadí na druhé místo mezi nejstahovanějšími. [5]

Při spuštení aplikace se uživateli zobrazí rovnou založení nového reportu bez nutnosti přihlašování. Uživatelské rozhraní je sestaveno z výchozích Android komponent bez dalšího stylizování. Při navigaci mezi obrazovkami vyskakují v dolní části reklamy. Na přehlednosti je kámen úrazu nastavení aplikace, které je velice vyčerpávající, jelikož je vše soustředěné nestrukturovaně v jednom seznamu. Narozdíl od ostatních aplikací unikátně nemá rozdělené sekce Účtenky a Reporty, ale účtenky se spravují v rámci jednotlivých reportů.

Záznam účtenky lze vytvořit pomocí 4 způsobů – načíst z PDF, zadat ručně, vyfotit nebo načíst z galerie. Při fotografování neindikuje rozpoznání účtenky, ale bylo možné dále fotografii editovat. Po editaci je ve výchozím nastavení zobrazen prázdný formulář bez zpracovaných hodnot. Nelze zároveň přidat volitelné parametry. Pokud uživatel požaduje nechat účtenku aplikací zpracovat, je nutné funkcionalitu aktivovat přes speciální volbu v menu z toolbaru s názvem „Automatic Scans (OCR)“, která je zpoplatněná.

Co si ale pravděpodobně zasloužilo pozornost uživatelů, je bohatá možnost exportování reportů – ve formátu PDF s různými úrovněmi detailu, CSV nebo například ZIP, opět s nastavitelným obsahem. Navíc umožňuje export i import jednotlivých účtenek ve vlastním „.SMR“ formátu. Dále v rámci reportu, je možné si zobrazit statistiky účtenek a nechat si je zálohovat.

[1] https://en.wikipedia.org/wiki/Hamburger_button

Bez rozpoznání obsahu účtenek je aplikace zcela zdarma, za 10 „Automatic Scans (OCR)“ si účtuje 28 Kč bez možnosti tarifního poplatku.

Aplikace po stránce UX/UI není tolik uživatelsky přívětivá oproti některým ostatním řešení, ale pokud jsou důležité pouze funkcionality, tak splní požadavky i náročnějších uživatelů. Nicméně při větším objemu účtenek je netarifní sazba za zpracování obsahu nákladná.

1.2.3 MrReceipt

Projekt MrReceipt (v originálu PanParagon) vyvíjen polskými vývojáři, se nachází ve stejné kategorii stažení přes 500 tisíc. Tým se aktivně snaží svojí aplikaci propagovat na sociálních sítích, kde se zároveň podílí na dobročinných akcí. [6]

Při spuštění aplikace je uživateli zobrazena obrazovka s podmínkami používání aplikace, ve kterých může například najít souhlas se zpracováním osobních dat i partnery vývojové firmy, což může být lehce znepokojující. Po odsouhlasení je představeno, co bylo přidáno v nynější aktualizaci. Vzhled aplikace je průměrný – nic neobvyklého, ale zároveň nijak výrazně zajímavé. Stejně jako u Expensify využívá přehuštěné *hamburger menu*.

Záznam účtenky lze vytvořit pouze vyfocením nebo načtením z PDF. Při fotografování není účtenka detekována, ale následně v editaci správně vytvaruje ořezání podél hran účtenky. Unikátně nabízí 3 barevné profily uložení fotografie – barevně, stupně šedi nebo černobíle. Editační formulář po zpracování je podobný jako u ostatních se dvěma extra parametry – záruka a doba na vrácení, podle kterých aplikace notifikací uživatele upozorní před vypršením. Rozeznává pouze datum a sumu, zbytek je na doplnění uživatelem.

Pokud se uživatel zaregistruje, otevře se mu funkce *Sharing center*, přes kterou je možné posílat jednotlivé záznamy účtenek jiným uživatelům a přijímat od ostatních uživatelů. Dále zde je možné nalézt obsáhlé filtrování a statistiky. Oproti ostatním aplikacím zde chybí sekce reportů. Obsahuje pouze premium (placenou) funkci, která přes email zašle odkaz na uživatelem vyfiltrované záznamy, což pro měsíční účetnictví není úplně vhodné, jelikož si filter nelze uložit a pokaždé se musí ručně nastavit.

Aplikace MrReceipt má pár unikátních funkcionalit, ale těmi podstatnými, zejména rozpoznáním obsahu účtenek, nepřevyšuje ostatní řešení. Je především zaměřena se na polské publikum, kterému přizpůsobuje svůj obsah, což pro firmu Matee Devs není nijak benefitující.

1.2.4 Easy expense

Aplikace vyvíjená týmem Easy Expense Tracker byla vydaná v lednu roku 2022 jako výsledný produkt jejich předešlých experimentů podobného zaměření. Na Google Play má přes 100 tisíc stažení. [7]

Při otevření aplikace se spustí tutoriál, který uživateli pomáhá aplikaci přizpůsobit svým potřebám a ukázat mu proces a výsledek skenování účtenky. Během něho je uživatel vyzván k přihlášení nebo vytvoření účtu přes email či Google účet. Co se týče UI, využívá hodně ikon a emoji, což zanechává méně profesionální ale přívětivý dojem. Na hlavní obrazovce je toho mnoho zobrazeno najednou, což dělá aplikaci mírně nepřehlednou.

Záznam účtenky lze vytvořit pomocí 3 způsobů – načtení z PDF, galerie nebo vyfocením. U vyfocení aplikace automaticky detekuje, pokud je nějaká účtenka v záběru a pokud uživatel zůstane dostatečně dlouho zaměřený, promptně naviguje do editace rozpoznaných informací, což je rychlé a velice příjemné. Bohužel občas se jí nedaří v účtence rozpoznat parametr obchodník.

Co je třeba zmínit, je možnost si přímo v aplikaci přidat nebo si vytvořit takzvaný *Workspace*, ve kterém může více uživatelů najednou nahrávat a spravovat své účtenky a reporty, ulehčující správu společných výdajů. Mezi další funkce patří synchronizace s kreditní kartou nebo bankovním účtem, podrobné statistiky, chat mezi spolupracovníky v rámci *Workspace*, vyhledávání a filtrování záznamů účtenek či reportů, export do PDF a další.

Aplikace má velké množství funkcionalit, ale též nemá možnost bezplatného používání. Lze si ji vyzkoušet na 7 dní zdarma, poté je třeba upgradovat na premium plán, který vychází na 1300 Kč ročně pro jednoho uživatele.

Nový uživatel se díky tutoriálu rychle zorientuje a je uživatelsky velmi přívětivá. Odrazujícím faktorem je, že v nabídce není bezplatné používání (to vysvětluje nižší počet stažení). Je vhodnější pro menší tým, který by potřeboval rychlé řešení a tato investice by se mu mohla vyplatit.

1.2.5 Foreceipt

Doporučená aplikace od Google Play vyvíjená firmou Foreceipt Inc. snažící se ulehčit správu výdajů a daní podnikatelům a malým firmám s počtem stažení něco přes 10 tisíc. [8]

Při spuštění se uživateli v 6 posuvných obrázcích aplikace představí a vyzývá ho k zaregistrování či přihlášení pomocí emailu nebo Google účtu. Po přihlášení do aplikace vyskočí dialog s výběrem státu, kde v nabídce jsou 4 možnosti – US, Canada, Australia a Other. Po dalším prozkoumání lze vyčíst, že zaměření účetnictví a daní je právě pro tyto 3 státy. Vizuálně je charakterizována světle zelenou barvou v kontrastu s bílou. Jiné barvy se krom záporných čísel v aplikaci nevyskytují, má dobře sjednocený design.

Záznam účtenky lze vytvořit pomocí 3 způsobů – načtení z PDF, galerie nebo vyfocením. Při focení nedetekuje účtenku, ani v následném editu nijak nerozpoznává a po potvrzení ořezávání je uživatel navigován na seznam všech účtenek, kde musí čekat kolem 15 sekund, než doběhne zpracování. Pokud došlo k chybě, tak je záznam zablokován a krom vymazání s ním nelze pracovat. V úspěšném případě si ale aplikace s účtenkami poradila poměrně dobře a všechny základní údaje správně rozeznala.

U představení byly zmíněné dvě funkcionality – cloud zálohování a sdílení v rámci týmů, bohužel jsou nefunkční. Dále zde uživatel může najít komplexnější vytváření reportů s několika různými parametry a možnostmi vyexportování do csv či PDF, měsíční statistiky a celkový přehled výdajů.

Při zaregistrování se automaticky zapne 15 denní zkušební doba premium účtu, který vyjde na 4$ měsíčně. Po vypršení se přepne do free plánu a deaktivují se premium funkcionality, které nejsou přímo specifikované.

Možná mohlo jít o nějaký bug v aktuální verzi, ale Foreceipt neobsahovalo týmy, o jejichž spuštění informovalo uživatele v představení aplikace. Dalším nepříznivým faktorem byla orientace na zmíněné tři státy projevující se přebytečnými parametry, které působily jen jako balast pro evropského uživatele.

1.2.6 Shrnutí analýzy existujících řešení

Každé řešení je něčím inspirativní, ale zároveň má své nedostatky. Tyto poznatky budou přímo či nepřímo využity v návrhu aplikace. Ze stávajících řešení stojí za vyzdvihnutí Expensify a Easy Expense. Oboje tyto aplikace měly dobré rozpoznání obsahu účtenek. U Expensify je problém, že všechna správa účtenek a reportů je řízena ve webovém rozhraní a tedy se nejedná o plnohodnotnou aplikaci v tomto ohledu. Navíc funkce SmartScanning probíhala dlouho a omezovala další akce. Easy Expense byla uživatelsky nejvíce přívětivá ze všech řešení, ale má vysokou tarifní sazbu na jednoho uživatele. Přehled všech zhodnocených řešení je v tabulce 1.1.

1.3 Analýza funkčních a nefunkčních požadavků

V následující sekci jsou dle předešlé analýzy domény a existujících řešení definovány funkční a nefunkční požadavky, které pomohou vytyčit rozsah a sjednotit očekávané funkcionality a omezení výsledné aplikace.

Tabulka 1.1 Srovnání existujících řešení

	I	II	III	IV	V	VI	VII	VIII
Expensify	✓	5/5	**X**	✓	**X**	30krát zdarma, 5 $ měsíčně	X	X
Smart Receipts	**X**	2/5	Při ořezávání	✓	✓	10x 1 €	✓	✓
Mr Receipt	**X**	3/5	Při ořezávání	✓	✓	zdarma	✓	✓
Easy Expense	✓	4/5	✓	✓	✓	1300 Kč ročně	✓	✓
Foreceipt	**X**	2/5	**X**	✓	**X**	4 $ měsíčně	✓	✓

I Vyžaduje přihlášení
II Design
III Detekce účtenky při skenování
IV Rozpoznání obsahu účtenky
V Možnost úpravy naskenované účtenky
VI Cena rozpoznání obsahu
VII Obsahuje statistiky
VIII Umožňuje správu účtenek

1.3.1 Uživatelé

Uživatelé budou majitelé zařízení s operačním systémem Android. Lze tedy předpokládat, že uživatelé budou tento operační systém znát a budou očekávat ovládací prvky, které znají. Proto bude aplikace využívat základních ovládacích prvků systému Android a bude se držet Android *design guidelines* (návrhových směrnicí).

1.3.2 Funkční požadavky

Funkční požadavky definují chování systému za konkrétních podmínek a jaké jsou jeho vlastnosti a funkce(popisují, co má aplikace dělat). Mezi tyto požadavky patří byznys požadavky, administrativní funkce, autentizace uživatelů apod. [9]

U každého požadavku uvedu jeho krátký popis a prioritu na stupnici VYSOKÁ, STŘEDNÍ, NÍZKÁ, odpovídající jeho důležitosti, pořadí při implementaci či zahrnutí do budoucího vývoje.

FP1: Vyfocení účtenky

Priotita: VYSOKÁ
Po odsouhlasení s využíváním fotoaparátu na zařízení bude moct uživatel pořídit fotografii účtenky, která se lokálně uloží a bude možné s ní dále pracovat.

FP2: Oříznutí pořízené fotografie

Priotita: VYSOKÁ
Po vyfocení bude možné vhodným způsobem pořízenou fotografii oříznout či upravit pro oddělení účtenky od zbytku fotografie pro přesnější zpracování a ušetření úložného prostoru na zařízení.

FP3: Rozpoznání obsahu účtenky

Priotita: VYSOKÁ
Aplikace zpracuje pořízenou fotografii účtenky, zanalyzuje její strukturu a obsah, z čehož vyextrahuje potřebná data (údaje na účtence) a uloží si je. Podstatnými údaji účtenky jsou: obchodník, datum a celková suma položek.

FP4: Vytvoření a persistence záznamu účtenky

Priotita: VYSOKÁ
Po vyfocení, či jiném způsobu zadání účtenky, a zpracování, aplikace vytvoří o účtence záznam spojující všechny tyto informace dohromady a uloží si jej do lokální databáze. Každý záznam účtenky musí vždy obsahovat údaje potřebné pro identifikaci výdaje v rámci účetnictví(výkazu zisků a ztrát), konkrétně:

- obchodník,
- datum,
- celková částka.

FP5: Správa záznamů účtenek

Priotita: VYSOKÁ
Aplikace dokáže vhodným způsobem zobrazit seznam náhledů všech účtenek, zobrazit si jejich podrobný detail, přidávat a mazat je.

FP6: Editační formulář záznamu

Priotita: VYSOKÁ
V aplikaci bude možné si zobrazit pro jednotlivé záznamy účtenek editační formulář, který umožňuje upravit údaje a tyto změny uložit.

FP7: Přehled rozpoznaného obsahu

Priotita: VYSOKÁ
Po zpracování fotografie budou uživateli, například přes editační formulář, zobrazeny rozpoznané údaje dané účtenky. U jednotlivých údajů bude uživateli mimo hodnoty zobrazena i oblast, odkud byla informace vyextrahována. Díky tomu uživatel může ověřit správnost přečtené hodnoty a případně opravit chyby vzniklé nepřesností knihovny zpracovávající obsah fotografie.

FP8: Reporty

Priorita: VYSOKÁ
Aplikace umožní vytvářet takzvané reporty, do kterých může uživatel libovolně přidávat nebo odebírat účtenky. Report si bude tento seznam udržovat a informovat uživatele o celkové sumě. Záznam účtenky může být přidán nejvýše jedenkrát v rámci jednoho reportu, ale v libovolném počtu reportů.

FP9: Vygenerování reportu

Priorita: VYSOKÁ
Z obsahu reportu bude možné vygenerovat a sdílet soubor ve formátu PDF nebo csv, který obsahuje všechny údaje jednotlivých záznamů krom fotografie, ta je nepovinná.

FP10: Zálohování fotek záznamů

Priorita: STŘEDNÍ
Po vytvoření záznamu účtenky se fotografie automaticky zálohuje ve vzdáleném repositáři zajišťující zálohu v případě ztráty dat, poškození či ztrátě zařízení. Při odstranění záznamu účtenky by se zároveň měla smazat její záloha.

FP11: Detekce účtenky

Priorita: STŘEDNÍ
Při pořizování fotografie účtenky bude uživatele aplikace vhodným způsobem informovat, že v záběru rozpoznala účtenku.

FP12: Načtení z galerie

Priorita: STŘEDNÍ
Záznamy účtenek bude možné nahrát přímo z galerie zařízení.

FP13: Načtení obsahu z oblasti

Priorita: STŘEDNÍ
Hodnota parametru účtenky může být aplikací rozpoznána načtením z oblasti vybranou uživatelem.

FP14: Zálohování pomocí Google účtu

Priorita: NÍZKÁ
Aplikace umožní uživateli se přihlásit ke svému Google účtu a zálohovat všechny záznamy účtenek a reportů v cloudovém úložišti. S tímto se pojí i automatická synchronizace, která umožní sdílet uživatelská data na více zařízeních pod jedním Google účtem.

FP15: Načtení záznamu účtenky z PDF

Priorita: NÍZKÁ
Záznamy účtenek bude možné načíst ze souboru PDF.

FP16: Možnost přidávání vlastních parametrů

Priorita: NÍZKÁ
V rámci editačního formuláře bude možné přidávat vlastní dodatečné parametry a informace o záznamu účtenky.

FP17: Filtrování

Priorita: NÍZKÁ
V seznamech záznamů účtenek napříč aplikací je bude možné filtrovat dle povinných údajů – obchodníka, data, celkové částky.

FP18: Spolupráce na reportech

Priorita: NÍZKÁ
Aplikace umožní vytvářet sdílené reporty, do kterých bude možné zvát ostatní uživatele. Přizvaní uživatelé mohou do reportu přidávat či odebírat vlastní záznamy účtenek.

FP19: Statistiky

Priorita: NÍZKÁ
Aplikace bude obsahovat sekci statistik, kde uživateli pomocí grafů a statistických hodnot zobrazí analytický přehled dosud přidaných záznamů účtenek.

1.3.3 Nefunkční požadavky

Nefunkční požadavky slouží jako omezení návrhu systému a specifikují kvalitativní atributy systému. [10]

NP1: Platforma Android

Aplikace bude primárně vytvořena pro operační systém Android. Rozšíření na ostatní platformy není omezením, ale vítaným bonusem.

NP2: Stabilita

Aplikace musí být stabilní a nesmí docházet k neočekávaným pádům a nestandardnímu chování aplikace.

NP3: Rozšířitelnost

Aplikace bude vyvíjená tak, aby umožnila rozšíření funkcionalit v budoucnosti.

NP4: Google ML Kit

Ke zpracování obrazu a rozpoznání obsahu účtenek bude využita knihovna Google ML Kit.

NP5: Offline rozpoznání obsahu

Focení a rozpoznání obsahu účtenky bude nezávislé na dostupnosti připojení k internetu, tedy bude probíhat na hostujícím zařízení aplikace.

1.4 Případy užití

Případy užití specifikují funkcionality aplikace z pohledu aktérů, v naší aplikaci z pohledu uživatele. Níže uvedený seznam popisuje případy užití z diagramu 1.2.

UC1: Navigace mezi sekcemi aplikace

Uživatel se chce navigovat mezi sekcemi aplikace (Účtenky, Reporty, Statistiky, Nastavení).

1. Uživatel klikne na ikonu odpovídající sekci, kam chce být navigován
2. Aplikace nad lištou zobrazí uživateli danou sekci

UC2: Vyfotografování účtenky

Uživatel chce vyfotografovat účtenku.

1. Uživatel klikne na tlačítko pro vyfocení
2. Aplikace zobrazí pohled zadní kamery
3. Uživatel vyfotí účtenku kliknutím na tlačítko pro vyfocení
4. Aplikace zobrazí vyfocenou fotografii

UC3: Nahrání fotografie účtenky z lokální galerie

Uživatel chce do aplikace nahrát fotografii účtenky uloženou na zařízení v galerii.

1. Uživatel klikne na tlačítko přidání fotografie z galerie
2. Aplikace zobrazí dostupné fotografie v galerii na zařízení
3. Uživatel vybere fotografii s účtenkou, jejíž záznam chce přidat
4. Aplikace zobrazí vybranou fotografii

UC4: Ořezání obrázku

Uživatel chce oříznout nahraný či vyfocený obrázek s účtenkou pro oddělení účtenky od zbytku obsahu fotografie.

1. Aplikace zobrazí vyfocenou fotografii s pohyblivým rámečkem pro ořezání
2. Uživatel pomocí rámečku vybere oblast s účtenkou pro ořezání
3. Uživatel potvrdí kliknutím na tlačítko
4. Aplikace vymezený prostor ohraničený rámečkem vyřízne a nahradí původní obrázek oříznutým

UC5: Vytvoření záznamu účtenky

Uživatel chce vytvořit záznam účtenky.

1. Uživatel se naviguje do sekce účtenek dle UC1
2. Uživatel klikne na tlačítko pro přidání nového záznamu účtenky
3. Aplikace zobrazí dialog s možnými způsoby, jak záznam přidat
4. a. Uživatel klikne na možnost pomocí vyfocení a pokračuje dle UC2
   b. Uživatel klikne na možnost přidání fotografie z galerie a pokračuje dle UC3
5. Aplikace zobrazí vyfocenou fotografii a pokračuje se dle UC4
6. Aplikace na pozadí fotografii zanalyzuje, pokusí se rozpoznat klíčové informace a zobrazí uživateli editační formulář, kde budou předvyplněné rozpoznané parametry. V těsné blízkosti rozpoznaných parametrů budou zobrazeny výřezy z fotografie, odkud byly extrahovány
7. Uživatel vyplní zbývající parametry, ověří správnost automaticky vyplněných díky výřezům a potvrdí kliknutím na potvrzovací tlačítko
8. Aplikace zobrazí uživateli detail vytvořeného záznamu účtenky

UC6: Zobrazení záznamů účtenek

Uživatel si chce zobrazit všechny dostupné záznamy a detail jednoho z nich.

1. Uživatel se naviguje do sekce účtenek dle UC1
2. Aplikace zobrazí seznam náhledů všech aktuálně dostupných záznamů účtenek. Tyto náhledy budou obsahovat klíčové informace – obchodník, datum, celková částka – a další doplňující prvky zlepšující přehled o záznamech
3. Uživatel klikne na náhled záznamu účtenky, pro který chce zobrazit detail
4. Aplikace zobrazí úplný popis daného záznamu – DETAIL. Zde budou zobrazena data o dané účtence včetně náhledu fotografie

UC7: Zobrazení náhledu fotografie účtenky

Uživatel si chce zobrazit náhled fotografie daného záznamu účtenky.

1. Uživatel si zobrazí detail dle postupu v UC6
2. Uživatel na tento náhled klikne
3. Aplikace zobrazí fotografii dané účtenky přes celý displej
4. Uživatel klikne na zavírací tlačítko
5. Aplikace zavře náhled dané fotografie a zobrazí uživateli stejný detail jako v kroku 1

UC8: Odstranění záznamu účtenky

Uživatel chce odstranit záznam účtenky.

1. Uživatel se naviguje do sekce účtenek dle UC1
2. Uživatel potáhne se záznamem účtenky pomocí odstraňovacího gesta
3. Aplikace zobrazí dialog dotazující se, jestli uživatel doopravdy chce daný záznam odstranit
4. a. Uživatel klikne na potvrzovací tlačítko
   b. Uživatel klikne na zamítací tlačítko
5. a. Aplikace smaže záznam se všemi jeho souvisejícími daty a odstraní jeho náhled z listu
   b. Záznam se vrátí do původní polohy před vychýlením

UC9: Přeuspořádání záznamů účtenek

Uživatel chce přeuspořádat seznam zmíněný v UC6.

1. Uživatel se naviguje do sekce účtenek dle UC1
2. Uživatel klikne na filtrovací tlačítko
3. Aplikace přeuspořádá pořadí aktuálních záznamů účtenek dle následujícího filtrovacího pravidla

UC10: Úprava záznamu účtenky

Uživatel chce upravit jeden ze záznamů účtenek.

1. Uživatel se naviguje do detailu daného záznamu dle UC6
2. Uživatel klikne na editační tlačítko
3. Aplikace zobrazí editační formulář, kde je obdobným způsobem, jako při procesu vytváření záznamu (popsaným v UC2), možné doplnit chybějící údaje o účtence, upravit ty stávající či přidat vlastní dodatečné údaje.
4. Uživatel provede žádané změny přepsáním hodnot parametrů ve formuláři
5. a. Uživatel klikne na potvrzovací tlačítko
   b. Uživatel klikne na zavírací tlačítko
6. a. Aplikace přepíše daný záznam účtenky, zavře editační formulář a zobrazí uživateli stejný detail z kroku 1 s aktualizovanými hodnotami
   b. Aplikace zavře editační formulář bez aplikování změn a zobrazí uživateli stejný detail z kroku 1

UC11: Vytvoření reportu

Uživatel si chce vytvořit report shrnující informace o záznamech účtenek dle jeho výběru.

1. Uživatel se naviguje do sekce reportů dle UC1
2. Uživatel klikne na tlačítko pro vytvoření nového reportu
3. Aplikace zobrazí dialog s textovým polem pro vyplnění názvu a možnostmi, jak report vytvořit
4. Uživatel vyplní název reportu, vybere jednu z možností a klikne na tlačítko „vytvořit“
5. a. Pokud si uživatel vybral vytvoření ručním výběrem, aplikace zobrazí seznam záznamů účtenek
   b. Pokud si uživatel vybral naplněním dle data, aplikace zobrazí dialog s kalendářem
   c. Pokud si uživatel vybral prázdný report, pokračuje se v kroku 7
6. a. Uživatel zvolí účtenky, které v reportu chce zahrnout, a klikne na potvrzovací tlačítko
   b. Uživatel vybere od-do rozsah dat účtenek, které chce v reportu zahrnout, a klikne na potvrzovací tlačítko
7. Aplikace uživateli zobrazí PŘEHLED vytvořeného reportu shrnující informace o zahrnutých záznamech účtenek

UC12: Zobrazení reportů

Uživatel si chce zobrazit seznam všech reportů a přehled jednoho z nich.

1. Uživatel se naviguje do sekce reportů dle UC1
2. Aplikace zobrazí seznam náhledů všech aktuálně dostupných reportů. Tyto náhledy budou obsahovat klíčové informace – název reportu, počet účtenek a celková suma – a další doplňující prvky zlepšující přehled o reportech.
3. Uživatel klikne na náhled reportu, který si chce zobrazit
4. Aplikace zobrazí úplný popis daného reportu – PŘEHLED. Přehled bude obsahovat hlavičku s klíčovými informacemi a seznam účtenek zahrnutých v tomto reportu.

UC13: Odstranění reportu

Uživatel chce ze seznamu reportu odstranit zvolený report.

1. Uživatel se naviguje do sekce reportů dle UC1
2. Uživatel potáhne náhled reportu pomocí odstraňovacího gesta
3. Aplikace zobrazí dialog dotazující se, jestli uživatel doopravdy chce daný report odstranit
4. a. Uživatel klikne na potvrzovací tlačítko
   b. Uživatel klikne na zamítací tlačítko
5. a. Aplikace odebere report ze seznamu bez újmy na záznamech účtenek v něm zahrnutých
   b. Report se vrátí do původní polohy před vychýlením

UC14: Správa záznamů účtenek v reportu

Uživatel chce přidat či odebrat záznamy účtenek z reportu.

1. Uživatel se naviguje do vybraného přehledu reportu dle UC12
2. Uživatel klikne na tlačítko „spravovat“ pod seznamem zahrnutých účtenek
3. Aplikace zobrazí seznam všech záznamů účtenek, kde vybrané jsou zvýrazněné
4. Uživatel zahrne ve výběru ty, které chce zahrnout, či odebere z výběru ty, které chce odebrat
5. Uživatel potvrdí výběr
6. Aplikace zobrazí přehled daného reportu s aktualizovanými informacemi a seznamem zahrnutých účtenek

UC15: Rychlé odebrání záznamu účtenky do reportu

Uživatel chce rychle odebrat záznam účtenky z reportu bez nutnosti postupu v UC13.

1. Uživatel se naviguje do vybraného přehledu reportu dle UC12
2. Uživatel přidrží záznam účtenky a stejným gestem jako v UC8 ho vychýlí zprava doleva
3. Aplikace odebere tento záznam z reportu a aktualizuje informace o reportu

UC16: Vyexportování reportu

Uživatel chce report o zahrnutých záznamech účtenek vyexportovat mimo aplikaci pro své další zájmy.

1. Uživatel se naviguje do vybraného přehledu reportu dle UC12.
2. Uživatel klikne na tlačítko sdílet
3. Aplikace zobrazí nabídku s výběrem formátů, ve které chce report vyexportovat
4. Uživatel si vybere jeden z formátů kliknutím na něj
5. Aplikace zobrazí rozhraní systému pro sdílení souborů mezi aplikacemi s předaným souborem
6. Pomocí preferované aplikace uživatel nasdílí report

UC17: Zobrazení statistik

Uživatel si chce zobrazit statistiky o přidaných záznamech.

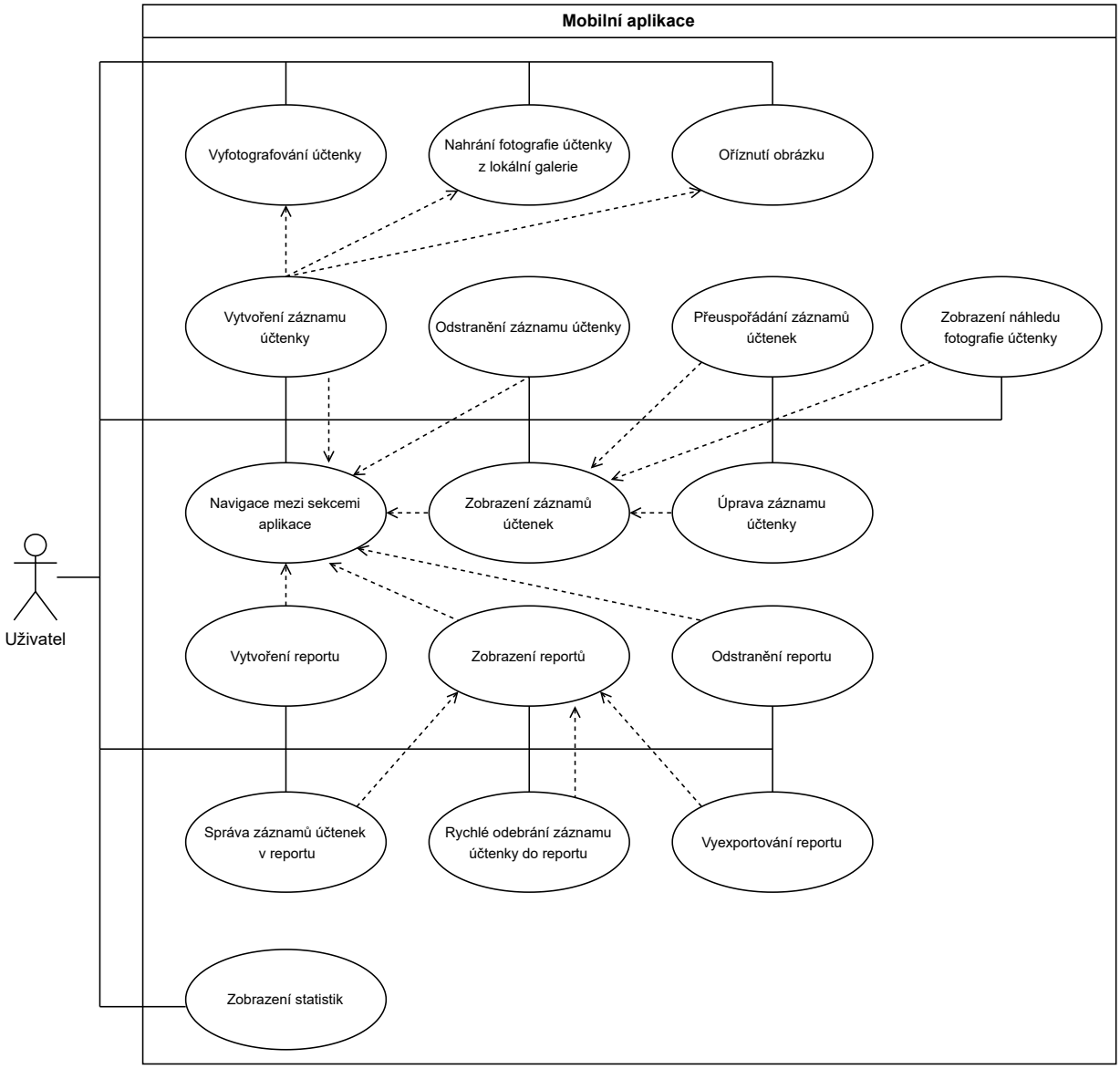
1. Uživatel se naviguje do sekce účtenek dle UC1
2. Aplikace zobrazí různé grafy a statistická data o přidaných záznamech účtenek

1.5 Diagram aktivit

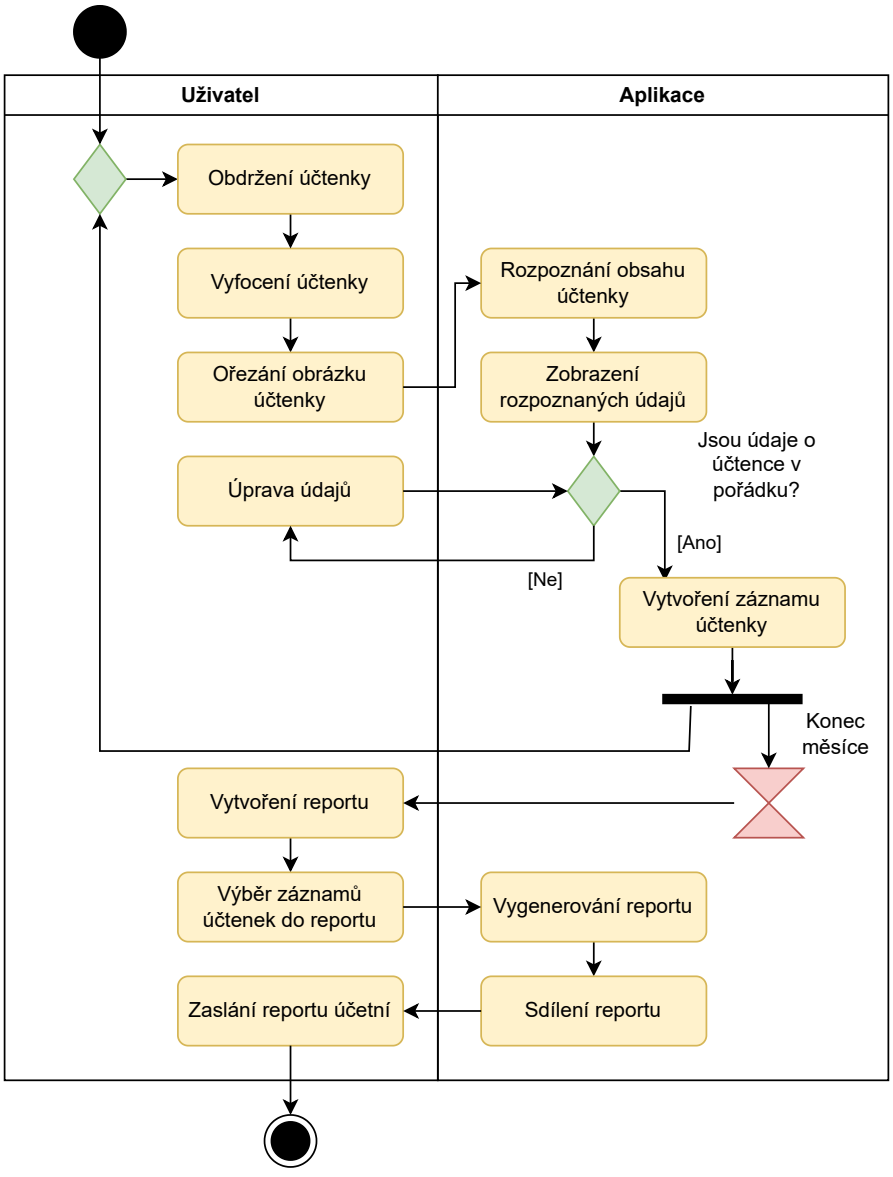
Diagram aktivit je grafické zobrazení procesu, který se skládá z různých činností a operací. Cílem tohoto diagramu je poskytnout strukturovaný přehled o procesu a umožnit rychlé a jednoduché pochopení souvisejících činností.

Zjednodušené znázornění procesu využití aplikace zaměstnanci Matee Devs při správě účtenek lze vidět v diagramu 1.3. Zobrazuje období jednoho měsíce, kdy uživatel postupně sbírá jednotlivé účtenky, nahrává je do aplikace pomocí vyfocení a na konci měsíce z nich vytvoří report, který zašle své účetní či účetnímu.

Obrázek 1.2 Use case diagram



Obrázek 1.3 Proces správy účtenek v rámci jednoho měsíce



Kapitola 2
Návrh

Tato kapitola se zaobírá fází návrhu. V softwarovém inženýrství je to proces, který se snaží najít nejlepší řešení pro problém s využitím získaných informací z fáze analýzy. Diskutuje o možných postupech, metodách a možnostech, zhodnocuje je a porovnává. Na základě těchto poznatků jsou ty nejvhodnější vybrány a pokládají základní kameny pro implementaci.

2.1 Platforma aplikace

Operačních systémů (platforem) pro mobilní zařízení je hned několik, z toho dva silně dominují – otevřený Android vyvíjený společností Google se zastoupením na trhu 70,96% a uzavřený iOS vyvíjený společností Apple se zastoupením na trhu 28,43%[1]. [11] Tyto statistiky jsou z celosvětového pohledu a na každém kontinentu je situace trochu jiná, často odpovídá ekonomické situaci, jelikož mobilní Android zařízení jsou dlouhodobě dostupnější.

V Evropě je tento podíl 61.05% ku 38.47% [11], což je mnohem vyváženější poměr a zcela běžně se setkáváme s oběma typy zařízení. Pro vývojáře to znamená, že pro vyšší pokrytí uživatelů je třeba vyvíjet pro obě tyto platformy. To se sebou přináší náklady, ale zároveň možnosti či přístupy, jak mobilní aplikace vyvíjet, které dle projektu stojí za uvážení.

Tyto přístupy se dělí na nativní vývoj, multiplatformní vývoj (kde navíc je na výběr z různých frameworků) a poměrně nově semi-multiplatformní (hybridní) přístup. V následujících sekcích bude možné nalézt jejich popisy zmiňující výhody a nevýhody.

V zadání je specifikován pouze operační systém Android, ale jedná se zatím pouze o prototyp. V dalším rozvoji této aplikace se počítá i s rozšířením na platformu iOS, která je ve firmě Matee Devs taktéž hojně zastoupena. Tudíž je vhodné provést analýzu těchto možností a zvolit tu, která se z hlediska poměru rychlosti vývoje a rozšiřitelnosti vyplatí.

2.1.1 Nativní vývoj

Nativní (od anglického *native*) vývoj spočívá ve vývoji aplikací pro jednu konkrétní platformu, na které budou spuštěny. Díky tomuto zaměření obecně vede k rychlejším a lépe responzivním aplikacím, což může být klíčové pro komplexní aplikace, kde je výkon důležitý. Další výhodou je větší kontrola nad vzhledem a pocitem z celé aplikace, jelikož je možné využívat nejaktuálnější grafické komponenty bez omezení. V neposlední řadě stojí za zmínku možnost využití funkcionalit pro specifická zařízení, které by v multiplatformním přístupu nebyly dostupné.

[1] Zbylých 0,61% není známo nebo se jedná o systémy starších zařízení. Z tohoto důvodu nebudou dále zmiňované.

V kontextu iOS a Android mobilních aplikací to tedy znamená, že se vyvíjí 2 zcela oddělené aplikace. Každá je psaná v jiném programovacím jazyku, může mít různou architekturu a zcela odlišné metody vývoje. V důsledku je tato možnost obecně finančně i časově náročnější.

2.1.2 Multiplatformní vývoj

Multiplatformní mobilní vývoj je přístup, který vám umožňuje vytvořit jedinou mobilní aplikaci, která poběží na několika operačních systémech. V aplikacích pro různé platformy lze sdílet část, nebo dokonce celý zdrojový kód. To znamená, že vývojáři mohou vytvářet a nasazovat mobilní prostředky, které fungují na Androidu i iOS, aniž by je museli překódovat pro každou jednotlivou platformu. [12]

Přestože idea multiplatformního vývoje není nijak nová, dříve oproti nativním aplikacím podstatně s výkonem zaostávala a složitě se tyto aplikace optimalizovaly. S moderními frameworky[2] – zejména Flutter a React Native – a vzrůstajícím výkonem hardwaru napříč celou škálou mobilních zařízení *native* dohání. Aplikace se testuje jednou, celkový čas vývoje a dodání na trh je kratší a tým menší se stejně zaměřenými vývojáři, čímž je předávání znalostí mezi nimi jednodušší. Tímto je tato možnost pro poptávku velice zajímavá.

Nicméně multiplatformní přístup má i svá úskalí.

- Ve složitých operacích a robustních aplikacích výkon stále nativní vývoj nedohnal.
- Vývojáři jsou odkázáni na daný framework – rozhraní jeho komponent, jak drží krok s nativní platformou, integrace knihoven třetích stran (často jsou vyvíjeny v nativních programovacích jazycích a chybí implementace) nebo například možnosti programovacího jazyka frameworku.
- Aplikační balíčky a celková aplikace zabírají více místa v úložišti.
- Možnou nevýhodou jsou stejně vypadající aplikace bez nativního dojmu, na který jsou uživatelé zvyklí.

2.1.3 Semi-multiplatformní vývoj – KMM

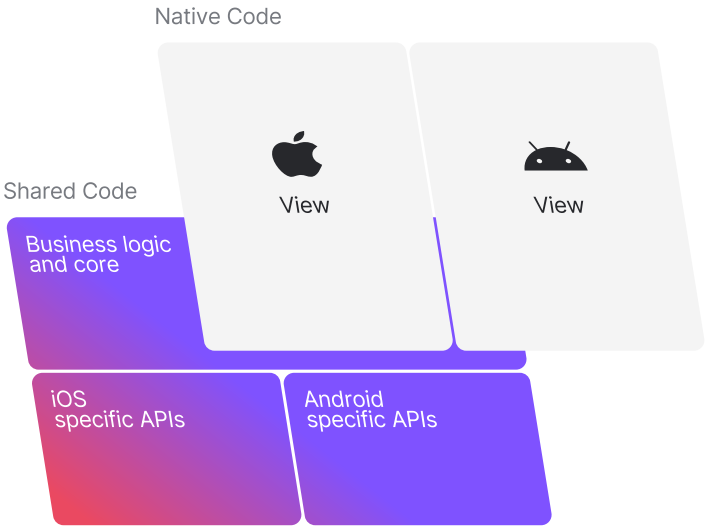
Kotlin Multiplatform Mobile (dále KMM) je technologie vyvíjená společností JetBrains a přináší zajímavou semi-platformní (hybridní) možnost. Staví na kombinaci multiplatformní logiky na pozadí a nativního UI jednotlivých platforem. Tím je možné dosáhnout výhod obou předchozích přístupů:

- využití maximálního potenciálu nativních UI komponent a responzivity,
- výkon aplikace,
- urychlení vývoje a zjednodušeného testování díky sdílení business a datové logiky,
- integrace knihoven třetích stran implementované pro nativní vývoj,
- využití funkcionalit pro specifická zařízení.

Princip této technologie je hezky vizualizovaný na obrázku 2.1 z oficiálního webu. V něm jsou znázorněny 2 složky vývoje. Spodní složkou je společný modul označený *Shared Code*, ve kterém nalezneme business logiku s datovou vrstvu označenou *Business logic and core* a rozhraní označené *iOS/Android specific APIs*, jejichž implementace je specifická dle platformy. Tato složka je sdílená pro obě platformy. Horní složka označená *Native Code* obsahuje nativní kód, konkrétně prezentační vrstvu. Tu si každá platforma implementuje odděleně a logiku na pozadí aplikace využívá napojením na sdílený modul. [13]

[2] „něco jako softwarová knihovna, je to kostra, na které může programátor vystavět své programy“ (https://it-slovnik.cz/pojem/framework/)

Obrázek 2.1 Kotlin multiplatform pro Android a iOS vývoj [13]



KMM je především atraktivní pro Android vývojáře, jelikož je psaný (jak název napovídá) v programovacím jazyku Kotlin, ve kterém se vyvíjí i nativní Android aplikace. Nemusí se tedy učit nový jazyk a mohou dále prohlubovat své znalosti. Je sice třeba dbát na strukturu projektu, sdružovat společnou logiku ve sdíleném modulu a mít vhodně zvolenou architekturu, ale jinak Android vývojář vyvíjí aplikaci stejně jako by vyvíjel nativně.

Následně se na společnou logiku ve sdíleném modulu iOS vývojář napojí pomocí vystavených rozhraní od KMM (přeložením z Kotlinu a vygenerováním pro Swift) a pouze doimplementuje specifické části rozhraní pro danou platformu a UI vrstvu aplikace. Tím se ušetří podstatná část vývoje iOS části a při změnách není třeba upravovat obě platformy.

2.2 Architektura aplikace

„*The architecture of a software system is the shape given to that system by those who build it. The form of that shape is in the division of that system into components, the arrangement of those components, and the ways in which those components communicate with each other.*

The purpose of that shape is to facilitate the development, deployment, operation, and maintenance of the software system contained within it. The strategy behind that facilitation is to leave as many options open as possible, for as long as possible.“ [14]

Zadefinovat si v krátkosti, co vlastně architektura aplikace znamená, není vůbec lehký úkol a s každým přiblížením či konkretizací narůstá o další odstavce. Zde je vybrána citace od „Uncle Boba“, jelikož architektury aplikací pro Android se od svého vzniku postupně měnily a vyvíjely, až se jejich evoluce zastavila u té, kterou právě Robert C. Martin popsal.

První odstavec přibližuje, co by architektura softwarového systému měla být – nějaký tvar daný těmi, kdo ho vytvořili, formovaný komponentami, uspořádáním těchto komponent a způsoby, jak mezi sebou komunikují. Druhý pojednává, jaký je důvod za takovým tvarem – ulehčení vývoje, nasazení, údržby systému a hlavně použití takové strategie umožňující mít co nejvíce otevřených možností, po co nejdelší dobu.

V této definici je zajímavý důraz nejen na samotnou strukturu aplikace, ale zároveň myšlenka, že je třeba brát ohled na její celoživotní cyklus a aplikovat zmíněnou strategii pro co největší ušetření času i peněz z dlouhodobého hlediska.

V anglických zdrojích jsou architektury také nazývány *design patterns* (návrhové vzory). Z tohoto důvodu se v citovaných částech objevuje i toto označení.

2.2.1 Architektura MVC

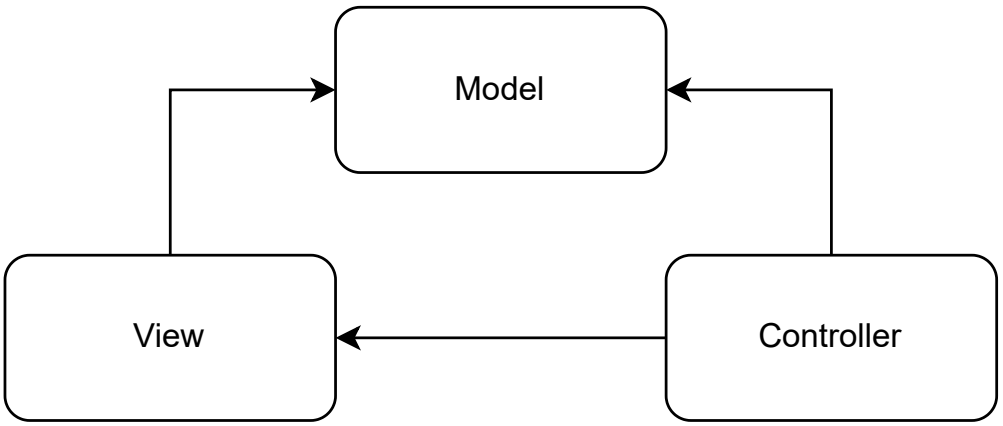
MVC (Model-View-Controller) je vzor v návrhu softwaru běžně používaný k implementaci uživatelských rozhraní, dat a řídicí logiky. Zdůrazňuje oddělení mezi business logikou softwaru a zobrazením. Toto „oddělení starostí“ zajišťuje lepší dělbu práce a lepší údržbu. [15]

Architekturu MVC popsal v roce 1979 norský vědec Trygve Reenskaug. Rozděluje aplikaci do tří komponent:

- Model (model), což je doménově specifická reprezentace informací, s nimiž aplikace pracuje.
- View (pohled), který převádí data reprezentovaná modelem do podoby vhodné k interaktivní prezentaci uživateli.
- Controller (řadič), který reaguje na události (typicky pocházející od uživatele) a zajišťuje změny v modelu. Na základě těchto změn se aktualizuje samotný pohled.

V počátcích aplikací pro operační systém Android se s touto architekturou experimentovalo a vznikaly knihovny snažící se ji implementovat, ale postupně se sjednotil názor, že není vhodná. Hlavním argumentem je, že implementace Android aplikací spočívá v definování takzvaných „aktivit“, které už *by design* (dle návrhu) zahrnují funkce View i Controlleru a jsou mezi sebou provázané – například životním cyklem. Tedy použití návrhového vzoru MVC vede k umělému rozdělování kódu, který se sebou silně souvisí, což jde proti návrhu Android komponent.

Obrázek 2.2 Model-View-Controller diagram [16]



2.2.2 Architektura MVP

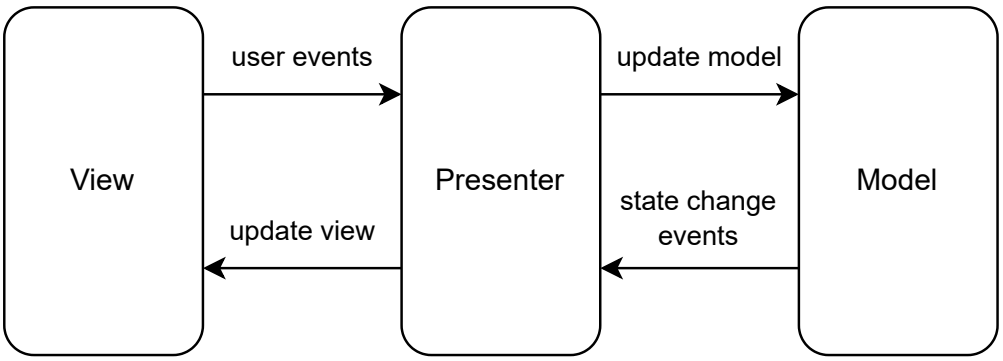
MVP (Model-View-Presenter) je návrhový vzor založený na konceptech vzoru MVC. Nespecifikuje však, jak celý systém strukturovat. Pouze určuje, jak strukturovat UI. Tento vzor obecně rozděluje odpovědnosti mezi čtyři komponenty. Za prvé, View je zodpovědné za vykreslování prvků uživatelského rozhraní. Za druhé, View rozhraní, které se využívá pro oslabení vazby mezi ním a Presenterem. Nakonec, Presenter interaguje s View a Modelem. Model je zodpovědný za business logiku a řízení stavu. [17]

Takto je citován jeden z výkladů MVP v kontextu vývoje pro Android. View rozhraním je myšleno rozhraní, které vystavuje Presenter, přes které může View aktualizovat na základě *callbacks* (zpětných volání). Díky tomu Presenter není na View závislý, ale pouze na specifikovaném rozhraní. V diagramu 2.3 je možné vidět jeho schéma (pro jednoduchost zdroj view rozhraní neuvádí). Změny oproti předchůdci MVC:

- Odstranění závislosti View na Modelu – tuto závislost nyní zprostředkovává Presenter.
- Splynutí Controlleru s View – View v kontextu MVP implementuje rozhraní, přes které je ovladatelné Presenterem. Nyní se View stará i o zpracování uživatelských interakcí a informuje o nich Presenter.
- Presenter – nová komponenta reagující na signály od View, načítá data z Modelu, zpracovává je a předává View.

Díky těmto změnám, zejména té druhé, byla architektura MVP přijatá komunitou. Je mnohem více použitelnější pro vývoj Android aplikací oproti MVC, protože je kompatibilní s návrhem Android komponent. Navíc kód prezentační logiky bylo možné testovat zvlášť a zvýšila se jeho znovupoužitelnost, jelikož Presenter lze použít pro více View.

Obrázek 2.3 Model-View-Presenter diagram [17]



2.2.3 Architektura MVVM

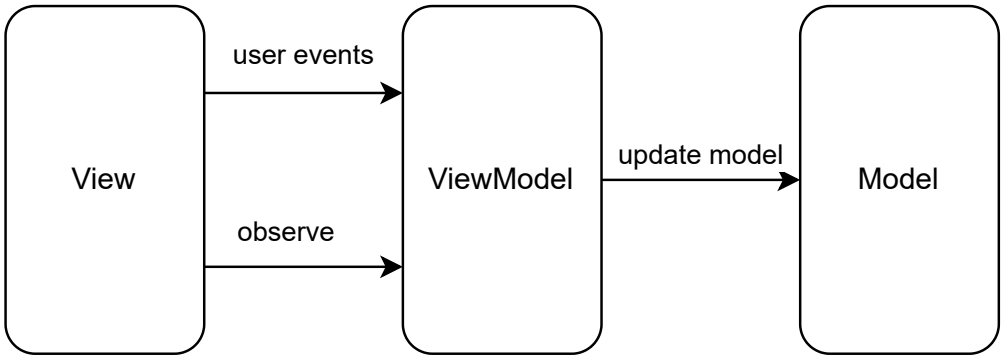
Model-View-ViewModel (MVVM) je průmyslově uznávaný vzor softwarové architektury, který překonává všechny nevýhody návrhových vzorů MVP a MVC. MVVM navrhuje oddělit logiku prezentace dat (zobrazení nebo uživatelské rozhraní) od hlavní části aplikace s business logikou. [18]

Z této definice není hned zřejmé, jaká konkrétní změna je oproti MVP, ale je to patrné z diagramu 2.4. Díky uplatnění observable patternu a reaktivního programování komponenta ViewModel aktuální data pouze „vystavuje“ (kdežto Presenter na přímo data předává) a View je „pozoruje“. Zároveň s daty vystavuje ViewModel i metody, přes které může View informovat o uživatelských interakcích. Pokud by došlo ke změně ve zdroji dat z Modelu či na základě vyslaného signálu z View, ViewModel dané změny zpracuje, aktualizuje vystavená data a View se automaticky překreslí bez nutnosti být o tom upozorněno jinou komponentou.

Mezi další velký bonus patří opět větší znovupoužitelnost a zvýšená modularita, jelikož View nemusí implementovat rozhraní Presenteru. Zároveň ViewModel nemusí být závislý na rozhraní, které poskytuje Presenter. Se zvýšenou modularitou a rozpadem „cyklické závislosti“ se úzce váže i lepší testovatelnost.

Co ale zde není přímo určené a je ponecháno na vývojářích, je místo nebo komponenta koncentrující business logiku aplikace. Ta je v kontextu MVVM rozmělněna mezi ViewModel a implementace rozhraní získávající data z datových zdrojů spadající pod Model, což v rámci jedno-platformních aplikací není tak stěžejní, ale může být problematické v kontextu multiplatformních.

Obrázek 2.4 Model-View-ViewModel diagram [18]



2.2.4 Architektura Clean

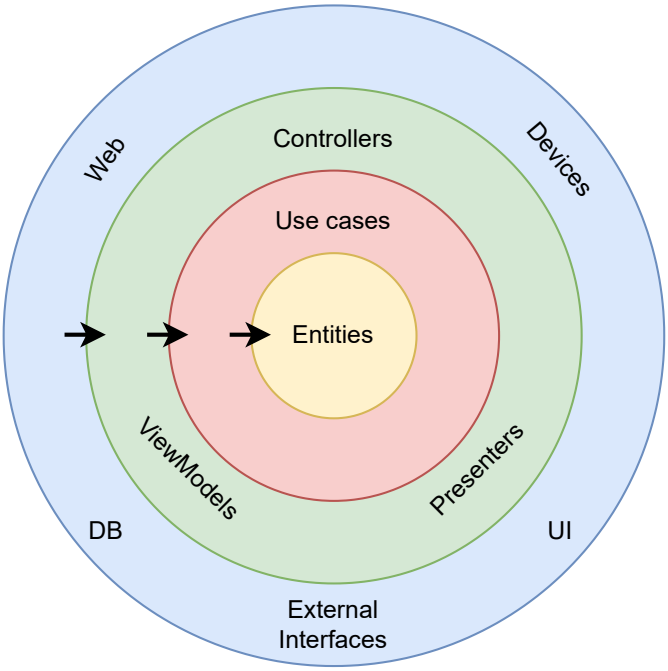
Doménově orientovaná architektura Clean byla navržena a popsána Robertem C. Martinem („Uncle Bobem“) ve stejnojmenné knize „Clean Architecture: A Craftsman's Guide to Software Structure and Design“ v roce 2017. Na rozdíl od přechozích návrhových vzorů považuje za jádro aplikace (zmiňovaný Model) doménovou vrstvu či business logiku nikoliv data. Uplatnění tohoto návrhového vzoru přinese aplikaci následující hodnoty:

- *Independent of frameworks* – Architektura nezávisí na existenci nějaké knihovny softwaru nabitého funkcemi. To vám umožňuje používat takové frameworky jako nástroje, spíše než aby vás nutily napěchovat váš systém do jejich omezení.
- *Testable* – Business logiku lze testovat bez uživatelského rozhraní, databáze, webového serveru, nebo jakéhokoliv jiného vnějšího prvku.
- *Independent of the UI* – Uživatelské rozhraní lze snadno změnit, aniž by bylo nutné měnit zbytek systému. Webové uživatelské rozhraní lze nahradit uživatelským rozhraním konzole, např. beze změny business logiky.
- *Independent of the database* – Nezávislé na databázi. Můžete vyměnit Oracle nebo SQL Server za Mongo, BigTable, CouchDB nebo něco jiného. Vaše business logika není vázaná na databázi.
- *Independent of any external agency* – Ve skutečnosti vaše business logika neví vůbec nic o rozhraních pro vnější svět. [14]

Pokud zapátráme ve zdrojích pojednávajících o této architektuře, všimneme si jejího netradičního kruhového diagramu 2.5 oproti běžným obdélníkům. Šipky míří od kraje směrem ke středu vyjadřující závislosti mezi jednotlivými vrstvami. Čím blíž jsme ke středu, tím jsou komponenty z dané vrstvy kritičtější pro daný systém a naopak čím dál jsme od středu tím méně je daná implementace méně podstatná a lehčeji nahraditelná.

Za doménovou vrstvu jsou brané *Entities* (Entity) a *Use Cases* (Případy užití). Entity zapouzdřují celopodnikovou kritickou business logiku. Entita může být objekt s metodami, nebo to může být soubor datových struktur a funkcí. Na tom nezáleží, dokud entity mohou být používány mnoha různými aplikacemi v systému. Software ve vrstvě případů užití obsahuje business logiku specifickou pro aplikaci. Zapouzdřuje a implementuje všechny případy použití systému. Tyto případy užití orchestrují tok dat z i do entit a řídí tyto entity k využití své kritické business logiky k dosažení cílů případu užití. [14]

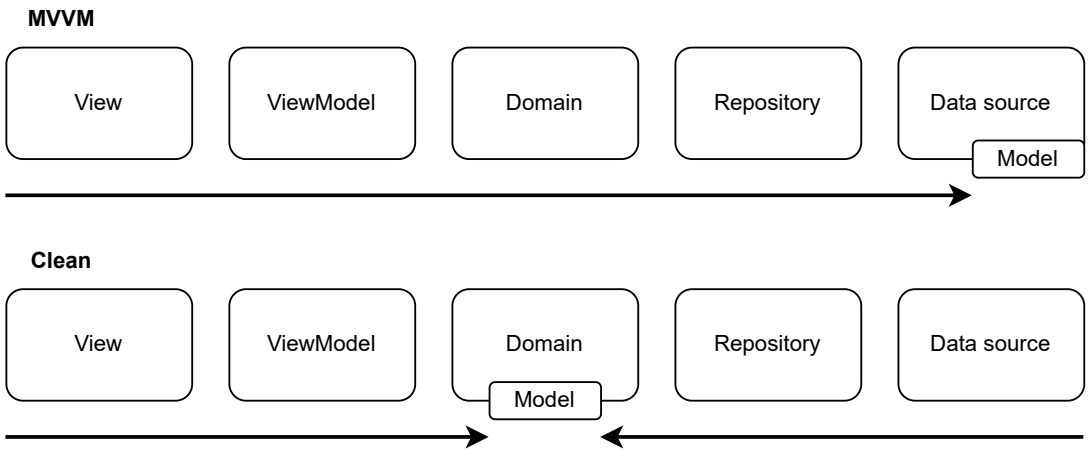
Obrázek 2.5 Clean architektura diagram [14]



Díky těmto principům je Clean architektura optimální pro multiplatformní aplikace využívající stejnou kritickou business logiku. Z dlouhodobého hlediska je systém velmi dobře škálovatelný a udržitelný.

Implementace tohoto návrhového vzoru v kontextu vývoje pro Android získala podobu aplikováním *Dependency Inversion Principle* (principu inverze závislostí) na MVVM znázorněno v diagramu 2.6. Šipky vyjadřují směr závislostí v dané architektuře. Pomocí zmíněného principu dosáhneme změny směru a nezávislosti doménové vrstvy.

Obrázek 2.6 MVVM a Clean porovnání

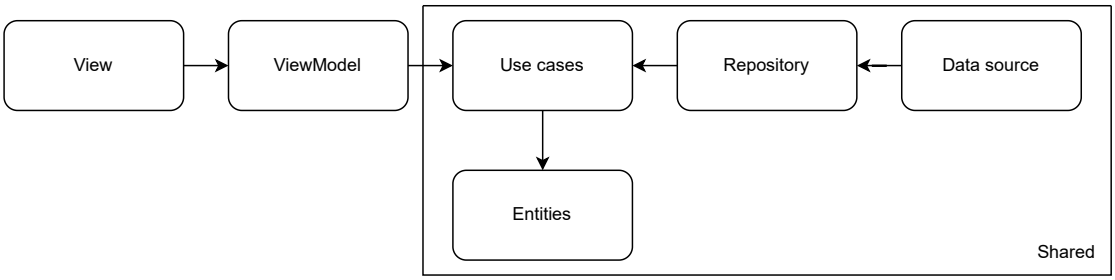


2.2.5 Zvolená architektura

Implementace aplikace bude postavena na Clean architektuře z několika důvodů. Oproti MVVM je sice v počátcích vývoje nutné psát více kódu kvůli vrstvě navíc – případy užití, ale díky ní je aplikace více modulární, lze jí lépe testovat, škálovat a nebude docházet k duplikaci kódu. Další podstatný důvod vychází z volby platformy aplikace. Výsledný prototyp dle zadání bude vyvíjen pro operační systém Android, ale s pomocí Kotlin Multiplatform psaný takovým způsobem, aby rozšíření o další platformu nebylo pracné a využilo se maximum již napsaného (otestovaného) kódu. Pro takový účel je Clean architektura optimální volbou oproti ostatním výše popsaným.

Její konkrétní podobu lze vidět v diagramu 2.7. Vrstvy ohraničené v bloku *Shared* budou sdílené napříč platformami. Napojení další platformy, které KMM podporuje, znamená pouze doimplementovat vrstvy View a ViewModel.

Obrázek 2.7 Zvolená architektura



2.3 Návrh rozhraní aplikace

2.3.1 Nástroje pro tvorbu prototypů

Tato sekce se bude zabývat volbou nástroje pro tvorbu UI/UX prototypu aplikace. Je důležité rozumět pojmu *fidelity*, který se v technickém kontextu překládá jako přesnost či věrnost. Přesnost prototypu je měřítkem toho, jak dobře se prototyp podobá zamýšlenému konečnému produktu. Prototyp s vysokou přesností vypadá a chová se velmi podobně jako zamýšlený produkt, zatímco prototyp s nízkou přesností může být velmi jednoduchý model sloužící jako „náčrtek“. Pojem *fidelity* je ve spojení s prototypem ustálený, proto dále nebude překládán. Bude uváděn na stupnici low–medium–high, což odpovídá nízké, střední či vysoké přesnosti.

Srovnání jednotlivých nástrojů je možné nalézt v příloze A, kde jsou představeny a porovnány. Ze srovnání, charakteristik nástrojů a splnění požadovaných kritérií byl zvolen nástroj Uizard pro návrh uživatelského rozhraní prototypu aplikace.

2.3.2 Low fidelity prototyp

Na základě funkčních požadavků a postřehů z konkurenčních řešení bylo navrhnuto uživatelské rozhraní aplikace. Zahrnuje navigaci mezi moduly, modul účtenek, proces skenování a úpravy účtenky, modul nastavení, modul reportů, proces vytváření reportu, správy účtenek v rámci reportu a proces exportování. V ukázkách 2.9, 2.8, B.2, B.3 a B.4 jsou možné vidět části navrhnutého low fidelity prototypu z nástroje Uizard. Některé z ukázek jsou zachyceny v módu vyšší fidelity pro větší názornost. Ukázky již zahrnují změny, které vyplynuly z uživatelského testování zaměstnanci. Kompletní statickou sadu lze najít v přiloženém médiu společně s odkazem na interaktivní klikatelnou verzi, na které byly testy prováděny.

2.3.3 Uživatelské testování low-fidelity prototypu

Jelikož je aplikace především určena zaměstnancům firmy Matee Devs, po navržení uživatelského rozhraní bylo důležité získat od nich zpětnou vazbu, aby se jim s aplikací dobře pracovalo. Pro účely uživatelského testování byly sepsány testovací scénáře rozdělené do dvou sekcí. Zaměstnanci se pokoušeli plnit jednotlivé kroky a na konci sekcí byly pokládány dodatečné dotazy ohledně průběhu a dojmu. Na základě jejich reakcí byly navrhnuty změny v uživatelském rozhraní sepsané v 2.3.4.

2.3.3.1 Sekce účtenek

1. Navigace na seznamu všech účtenek
   a. Najdi možnost, jak přidat nový záznam účtenky
   b. Najdi možnost, jak vymazat záznam účtenky
   c. Najdi možnost pro filtrování účtenek
   d. Najdi způsob, jak si zobrazit detail účtenky
   e. Najdi způsob, jak se navigovat do Reportu
   f. Najdi způsob, jak se dostat do nastavení
2. Navigace v detailu účtenky
   a. Naviguj se do libovolného detailu účtenky
   b. Zobraz si fotografii účtenky
   c. Naviguj se zpět na detail
   d. Najdi způsob, jak si zobrazit dodatečné informace o účtence
   e. Zobraz si dodatečné informace
   f. Schovej dodatečné informace o účtence
   g. Najdi možnost editace záznamu účtenky
   h. Přejdi do editace účtenky
   i. Vrať se zpět bez provedení změn
3. Navigace v detailu účtenky
   a. V seznamu všech účtenek vytvoř nový záznam (stisknutím tlačítka +)
   b. Vyber způsob přidání – vyfocením
   c. Vyfoť účtenku
   d. Potvrď ořezání účtenky (jakým způsobem by si ořezávanou plochu změnil?)
   e. Vyplň jméno obchodní firmy (Merchant)
   f. Potvrď finalizaci editace účtenky
   g. Vrať se zpět na seznam účtenek
4. Navigace v detailu účtenky
   a. V seznamu všech účtenek vyber první záznam
   b. Přepni se do editačního módu
   c. Změň kategorii účtenky na libovolnou jinou kategorii
   d. Přidej *tag* (označení) „s klientem“
   e. Potvrď změny
   f. Vrať se zpět na seznam účtenek

5. Přidání vlastního parametru
   a. V seznamu všech účtenek vyber první záznam
   b. Přepni se do editačního módu
   c. Zvol možnost přidání dalšího vlastního parametru
   d. Parametr pojmenuj „Approval Code“ a jeho hodnotu naskenuj
   e. Zvol oblast, odkud má být parametr naskenován a potvrď

2.3.3.2 Sekce reportů

1. Navigace v reportech
   a. Najdi možnost, jak přidat nový report
   b. Najdi možnost, jak vymazat report
   c. Najdi možnost pro filtrování reportů
   d. Najdi způsob, jak si zobrazit detail reportu
2. Navigace v detailu reportu
   a. Naviguj se do libovolného detailu reportu
   b. Najdi možnost přidání záznamu účtenky do reportu
   c. Najdi možnost odebrání záznamu účtenky z reportu
   d. Přejmenuj daný report
   e. Najdi možnost exportování reportu
   f. Vrať se na seznam všech reportů
3. Vytvoření reportu za poslední měsíc
   a. Zvol možnost vytvoření nového reportu
   b. Vyber možnost vytvoření dle data
   c. Jak bys vybral 15. březen 2021?
   d. Přejdi na další krok
   e. Vytvoř report
   f. Vyexportuj tento report do CSV

2.3.4 Změny na základě uživatelského testování

Interaktivní low fidelity prototyp prošlo 5 zaměstnanců Matee Devs. Na základě zpětné vazby a průběhu testování dle scénářů byly v návrhu uživatelského rozhraní udělány tyto změny:

1. místo samotné ikony v tlačítku pro přidání nového vlastního parametru přidán text „Add parameter“,
2. u nezdařeného automatického načtení údaje účtenky přidána hláška „Paramater unsuccessfully parsed“ místo ikony otazníčku,
3. v nastavení přidána sekce „App“ s volbou „About“ a „Color mode“,
4. upravený proces vytváření reportu, přidána možnost přidání dle tagu,
5. změna ikony pro exportování pro lepší návodnost akce,
6. přidána obrazovka pro editaci detailu reportu pro sjednocení s editací detailu záznamu účtenky,
7. rozšíření *tap* zóny u dialogů na celý řádek,

8. možnost pro úpravu fotografie účtenky přejmenována z „Edit“ na „Modify“ kvůli duplikaci textace s „Edit“ nadpisem obrazovky,
9. u dialogu vybírání datumu přidána možnost vybrat rok pomocí rozbalovacího listu.

2.4 Zvolené technologie a nástroje pro realizaci aplikace

V této sekci budou zmíněny technologie a nástroje použité v implementaci prototypu aplikace. U každé bude zmíněn její krátký popis, využití a proč byla zvolena.

2.4.1 Jetpack Compose

Jetpack Compose je moderní doporučovaný *toolkit* (sada nástrojů) pro tvorbu nativního UI Android aplikací. Jeho první stabilní verze byla vydána 28.7.2021 za účelem nahradit původní způsob tvoření prezentační vrstvy imperativním způsobem pomocí *Views* zapisovaných v XML souborech. Tento způsob byl zdlouhavý a komplexnější obrazovky byly náročné na návrh a implementaci. Nyní díky Jetpack Compose a jeho deklarativnímu přístupu je možné tvořit UI přímo v kódu v programovacím jazyku Kotlin. Slibuje méně celkového kódu, intuitivnost, zrychlení vývoje a větší možnosti při tvoření UI. [23]

Idea stojí na tvoření UI pomocí „Composable“ funkcí. Tyto funkce přijímají vstupy jako argumenty a vrací komponenty UI jako výstup. Odtud vychází deklarativní přístup řízený daty (zmíněných argumentů). V kódu je lze rozeznat díky anotaci @Composable. Lze je mezi sebou kombinovat a zanořovat, což vytváří modulárnější přístup a zároveň možnosti tvořit komplexní obrazovky. Rychlost překreslování vychází z takzvané fáze „rekompozice“. V této fázi je efektivním způsobem identifikováno, ve kterých Composables došlo k nějaké změně a je třeba je překreslit. Zbytek je přeskočen pro maximální efektivnost.

I přesto, že se jedná o mladou technologii, rychle se na trhu adaptovala a osvojila, jelikož lze zavádět postupně. Obrazovky psané v Jetpack Compose lze napojit na ty původní či je dokonce mezi sebou kombinovat. Díky tomu je možné využívat knihovny třetích stran, které nemají Compose implementaci. Aktuální projekty tedy není nutné přepisovat, pouze zaktualizovat naimportované knihovny a novou logiku psát přímo v Jetpack Compose.

Prezentační vrstva bude implementována pomocí tohoto *toolkitu*, jelikož je doporučovaný přímo od Googlu (lze hovořit o standardu). Z pohledu dnešních trendů se nativní vývoj bude ubírat právě tímto směrem. Je to tedy rozhodnutí i z hlediska budoucího rozvoje aplikace.

2.4.2 SQLDelight

SQLite je nejpoužívanější databázový stroj na světě. SQLite je zabudován do všech mobilních telefonů a většiny počítačů a je součástí bezpočtu dalších aplikací, které lidé používají každý den. Formát souboru SQLite je stabilní, multiplatformní a zpětně kompatibilní a vývojáři se zavazují, že jej takto udrží do roku 2050. [24]

SQLDelight je knihovna, která umožňuje vytvářet, spravovat a dotazovat SQLite databáze z aplikací pro Android, iOS, tvOS a macOS. Generuje typově bezpečná kotlin API z příkazů SQL. Ověřuje schéma, příkazy a migrace v době kompilace a poskytuje funkce IDE, jako je automatické dokončování a nápovědy k vylepšení, které zjednodušují psaní a údržbu SQL. Umožňuje aplikacím získat zabezpečené přístupy k datům v databázi pomocí jediného zápisu kódu a zároveň umožňuje uživatelům snadno pracovat s daty prostřednictvím SQL. [25]

Tato knihovna byla zvolena díky možnosti generování kódu dle jednoho jednoduchého zápisu, který je kontrolován a snadno se píše díky pluginu v IDE Android Studio. Na základě něho je zkonstruována samotná databáze, modelové třídy a metody pro práci s nimi v kódu. Zároveň je postavená nad SQLite databází, která je rychlá, zabírá málo úložného prostoru a je dostatečná

pro potřeby této aplikace – podporuje multiplatformní vývoj (lze ji zahrnout ve společné logice) a dle závazku vývojářů udržovaná do roku 2050.

2.4.3 Firebase Storage

Firebase Storage je jeden z modulů nabízený platformou Firebase, která vyvíjí a spravuje podpůrné systémy pro mobilní a webové aplikace, spadající pod Google. Jedná se o cloudové úložiště s velmi jednoduchou integrací pro obě platformy Android i iOS. Nabízí robustní řešení pro nahrávání a stahování souborů, jelikož bere v potaz, že uživatelé nemají vždy aktivní připojení k internetu. Při výpadku tyto akce pozastaví a automaticky obnoví při získání připojení. [26] Lze jej snadno spravovat (soubory, nastavení, přístupy) v rámci Firebase console pro daný projekt, na který je aplikace napojena. Možným benefitem zavedení je snadné napojení dalších Firebase modulů, například Google Analytics.

Přes tento modul budou zálohovány záznamy účtenek. Podporuje přihlášení přes Google účet, přes který bude mít uživatel k nim přístup. Dnes je velmi běžné, že uživatel již takový účet má a nebude třeba se již nijak registrovat.

Byla zvažována i alternativa – Google Drive, využívaná miliardou uživatelů pro správu souborů v cloudovém řešení. V aktuálním stavu je ale její integrace silně náročnější oproti zvolené variantě. Dříve fungovala ve formě SDK zastřešující procesy napojení, ale tato varianta od 6. prosince 2018 přestala být podporována. Dokumentace nynějšího řešení pro Android ve formě API je rozprostřena mezi několik zdrojů (není aktuálně jednotná, zahrnující všechny části implementace). Důvodem bylo opakující se předávání mezi týmy Firebase a Google tam a zpět. Z časových důvodů a komplexity aktuální verze byla tato možnost zavrhnuta.

2.4.4 Google ML Kit SDK

ML Kit přináší expertýzu Google ve strojovém učení mobilním vývojářům ve výkonném a snadno použitelném balíčku. Zpracování ML Kitu probíhá na zařízení – lze jej použít pro zpracování obrázků a textu, které musí zůstat v zařízení. Díky tomu je rychlý a odemyká možnost využití v reálném čase, jako je zpracování vstupu z kamery.[27]

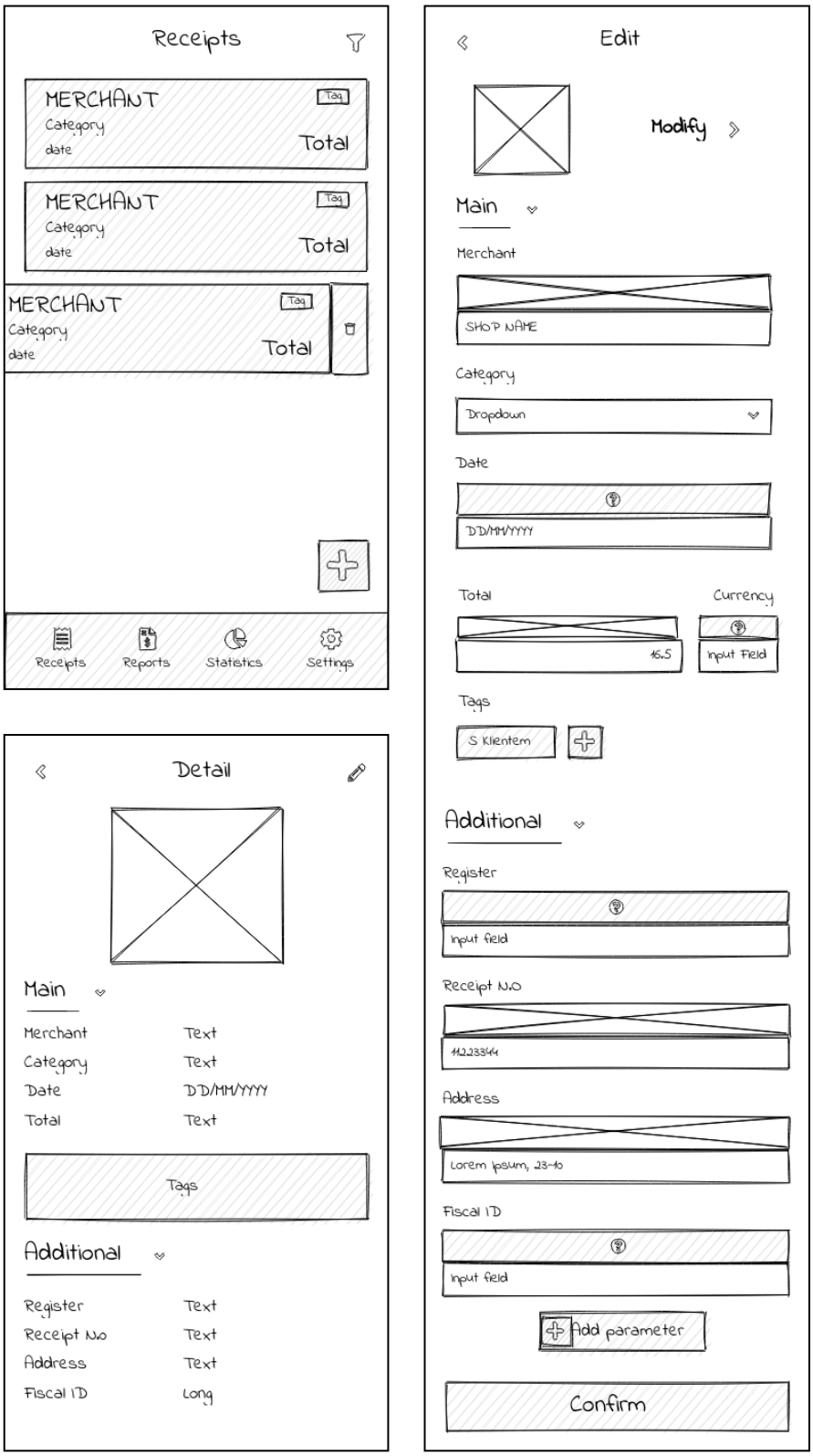
Z této knihovny budou využity 2 moduly – *Text Recogniction* pro rozpoznání textu z obrazu a *Image labeling* pro rozpoznání účtenek v obraze. Pomocí těchto modulů aplikace zpracuje importované obrázky či výstup kamery pro získání dat z účtenek, které následně budou zanalyzované a zpracované. Technické přiblížení je možné nalézt v odpovídající sekci v kapitole Implementace.

Na zpracování textu z obrázku je mnoho veřejných placených i volně dostupných API, k jejichž využití je třeba obrázek online zaslat. To by nesplňovalo NP5, tedy že zpracování obrazu má být dostupné i v offline režimu. Navíc v této variantě nemůže být vždy zaručeno, aby třetí strana tyto data nezpracovávala. Je tedy třeba vybrat importovatelnou knihovnu nezávislou na připojení k internetu. To výběr zužuje na komerční řešení a zadavatelem zvolený ML Kit od Google. A jelikož je volně dostupný pro obě platformy Android i iOS, souhlasím s touto volbou.

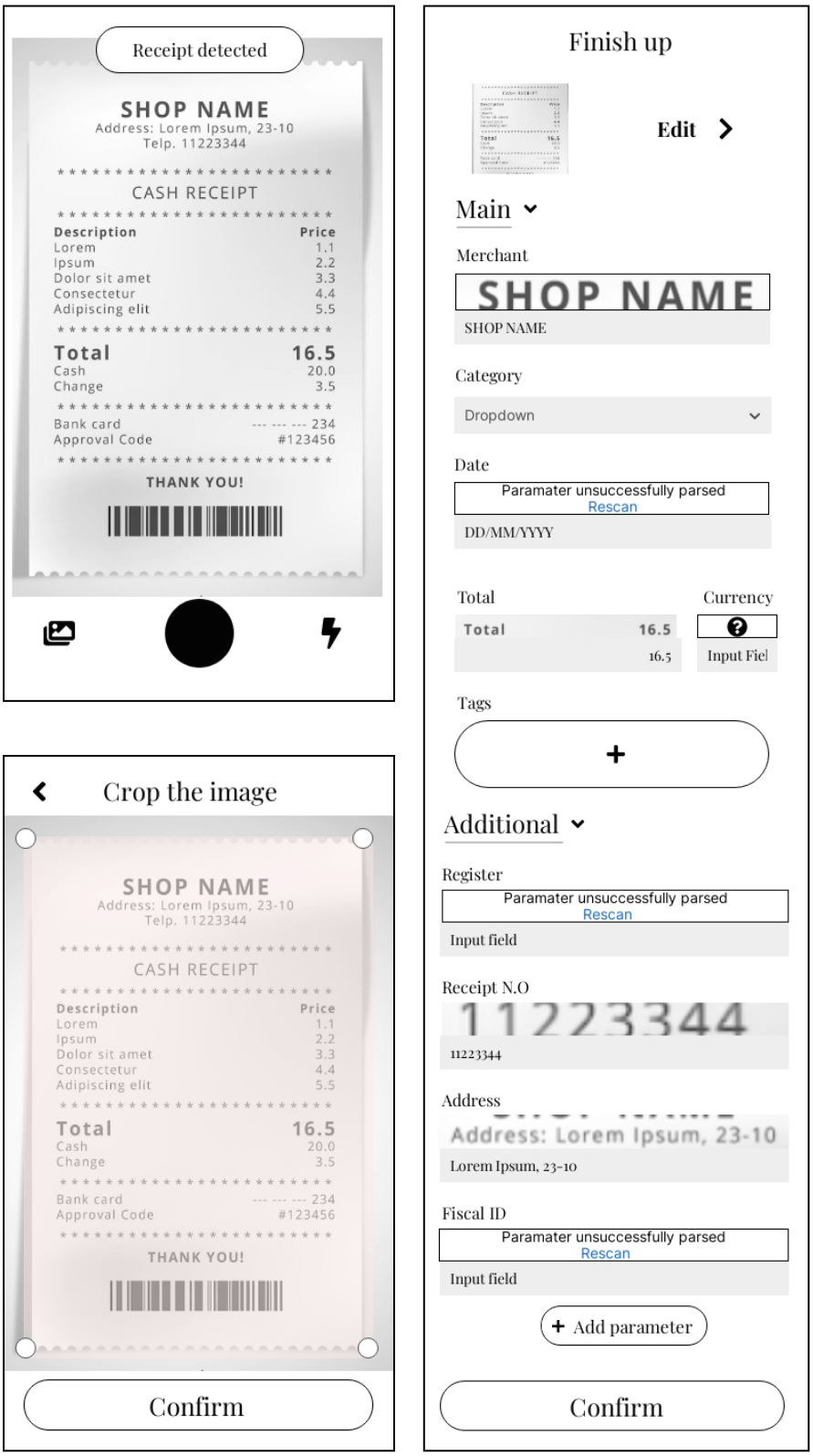
2.4.5 Android Image Cropper

Pro ořezání obrázků byla vybrána knihovna Android Image Cropper. V kontextu Androidu se jedná o známou a doporučovanou knihovnu díky její robustnosti a optimalizaci. Při ořezávání dynamicky přibližuje či oddaluje obrázek pro lepší přehled o výsledku. Mimo ořezávání je možné obrázek rotovat či překlápět. Dalším podstatným důvodem zvolení této knihovny byla implementace komponenty pro ořezání pro Jetpack Compose, kterým bude UI aplikace vytvořeno. Odkaz na repositář této knihovny je možný nalézt v příloze.

Obrázek 2.8 Ukázka návrhu rozhraní procesu editování – seznam všech účtenek (vlevo nahoře), detail určitého záznamu účtenky (vlevo dole), celý pohled editačního formuláře (vpravo)



Obrázek 2.9 Ukázka návrhu rozhraní procesu fotografování účtenky – pohled s kamerou (vlevo nahoře), ořezání obrázku (vlevo dole), celý pohled editačního formuláře (vpravo)



Kapitola 3
Implementace

Tato kapitola se zaobírá implementací prototypu aplikace. Zahrnuje postup, přiblížení jednotlivých částí s ukázkami kódu a zmínky o problematických částech s jejich vyrovnáním.

3.1 Postup

Implementaci byla rozdělena na části odpovídající jednotlivým funkčním požadavkům. Vývoj začal základními a kritickými částmi a ty byly postupně rozšiřovány. Toto bylo možné, jelikož požadavky byly vedoucím pevně dané a nebyly očekávané změny. Pořadí bylo určovány dle priorit jednotlivých funkčních požadavků. Díky tomu bylo možné se soustředit na stěžejní funkcionality aplikace a postupovat k těm s nižší prioritou.

Při implementování jednotlivých částí byl kód rozšiřován dle zvolené architektury 2.7 s postupem proti směru závislostí. Nejdříve byly navrhnuty modelové třídy (*Entities*) a nad nimi případy užití (*Use-cases*), které vystavovaly rozhraní repositářů (*Repositories*). Tyto tři části reprezentovaly business vrstvu aplikace, na které bylo navázáno implementací datové vrstvy. Následně byla implementována rozhraní repositářů vystavující obdobným způsobem rozhraní, které byly implementovány pro jednotlivé zdroje dat (*Data source*). Tím byla vyřešená společná (*Shared*) část kódu a následovala implementace prezentační vrstvy pro Android[1]. Přes *ViewModel* jsou vystavena data (získaná pomocí *Use-cases*) a pomocné metody pro zpracování vyvolaných akcí. Na to se navázalo *View*, které vykreslovalo uživatelské rozhraní, zobrazovalo data a reagovalo na interakce.

3.2 Business vrstva

Business neboli doménová vrstva se zaměřuje na logiku jádra aplikace. V této části nalezneme zpracování a manipulaci s daty získané přes repositáře (*Repositories*) vyvolané uživatelskými interakcemi s aplikací zavoláním případů užití (*Use-cases*). Mezi stěžejní logiku této vrstvy zároveň patří i analýza a zpracování obsahu účtenek.

3.2.1 Entities

V Clean architektuře je entita objekt, který představuje něco smysluplného v doméně. Nejedná se o databázovou tabulku nebo objekt v prezentační vrstvě. Entita je čistě datový objekt, který by se měl snadno serializovat a deserializovat.

[1] Zde by mohl navázat další vývojář implementací pro jinou platformu

V prototypu se pracuje celkově se třemi hlavními entitami:

1. Entita Receipt reprezentuje jeden záznam účtenky obsahující identifikátor, proměnné týkající se obsahu účtenky a cesty k obrázku účtenky (jedná lokální a druhá ke stažení ze zálohovacího repositáře). Mimo těchto údajů jsou zde dva odvozené – potenciální referenční název souboru v zálohovacím repositáři a příznak, jestli je fotografie účtenky zálohovaná.

```
data class Receipt(
    val id: ReceiptId,
    val merchant: String,
    val timestamp: Long,
    val category: ReceiptCategory,
    val total: Double,
    val imageLocalPath: String,
    val imageDownloadUrl: String? = null,
) {
    val imageRemoteRef get() = "$id $merchant"
    val isBackedUp: Boolean get() = imageDownloadUrl != null
}
```

Výpis kódu 1 Entita Receipt

2. Entita Report reprezentuje jeden report obsahující identifikátor, jméno, datum vytvoření, seznam identifikátorů zahrnutých účtenek a celkovou sumu.

```
data class Report(
    val id: ReportId,
    val name: String,
    val createdAt: Long,
    val total: Double,
    val receiptsIds: List<ReceiptId>
)
```

Výpis kódu 2 Entita Report

3. Entita ReceiptML obsahující obrazce z bodových souřadnic vyjadřující výřez z fotografie účtenky, odkud byl daný údaj účtenky rozpoznaný.

```
data class ReceiptML(
    val id: ReceiptId,
    val merchantPoints: MLReadPoints?,
    val datePoints: MLReadPoints?,
    val categoryPoints: MLReadPoints?,
    val totalPoints: MLReadPoints?
)
```

Výpis kódu 3 Entita ReceiptML

3.2.1.1 Práce s obrázky

Jednou z prvních překážek psaní sdíleného kódu nezávislého na platformě byla práce s multimediálními soubory, jelikož každá platforma je reprezentuje svými vlastními třídami a jsou dostupné přes různá rozhraní. Aby nebylo nutné implementovat na každé platformě kód vztahující k jejich správě a řešit jejich persistenci, aplikace využívá lokální systémovou galerii.

Se soubory pracuje na bázi URI[2], které obdrží pomocí nativního API po vyfocení účtenky či výběru ze zmíněné galerie. Knihovny pro práci s obrázky s touto možností počítají a přes URI si obrázek načtou do potřebné třídy pro svoji činnost. Tento formát je možné reprezentovat klasickým řetězcem – STRING, díky čemuž lze s obrázky účtenek pracovat ve sdíleném kódu a snadno definovat související rozhraní a třídy. Příkladem je entita RECEIPT a její parametr IMAGELOCALPATH obsahující lokální cestu k obrázku záznamu účtenky.

3.2.2 Use-cases

Případ užití je popis způsobu, jakým se systém používá. Určuje vstup, který má uživatel poskytnout, a výstup, který má být vrácený uživateli s kroky zpracování, které jsou součástí produkce tohoto výstupu. Případ použití popisuje obchodní pravidla specifická pro aplikaci. [14]

```
interface UseCase<in Params, out T> {
    suspend operator fun invoke(params: Params): T
}
```

Výpis kódu 4 Use Case interface

V multiplatformním přístupu KMM v kombinaci s Clean architekturou jsou USE-CASES ten bod sdílené logiky, na kterou navazuje prezentační vrsta. Pomocí definovaných vstupů je voláme a na základě jejich výstupů řídíme zobrazovaná data a operace na displeji konkrétního zařízení. Implementace daného případu užití je reprezentována třídou, jejíž činnost se spouští pomocí metody INVOKE. Její parametry odpovídají zmíněným vstupům a výsledek výstupům. V konstruktoru jsou předávané závislosti (*Repositories*) ve formě rozhraní, jejichž implementace se dosazuje přes *dependency injection* popsané v 3.5.

V ukázce 5 je popsaný případ užití odstranění záznamu účtenky. Jako vstup má identifikátor této účtenky RECEIPTID a nevrací nic – jedná se o proceduru. Nejdříve se pokusí načíst záznam účtenky z repositáře a pokud se povede, vymaže obrázek dané účtenky z lokálního i zálohovacího úložiště a zároveň odstraní záznamy ve souvisejících repositářích.

3.2.3 Repositories

Repositáři jsou myšlená rozhraní, které jsou využívány případy užití. Tato vrstva řídí a zpracovává data mezi aplikací a datovými zdroji. Je třeba zmínit, že REPOSITORIES se nemyslí pouze třídy implementované na základě návrhového *repository* vzoru, ale i například pomocné knihovny či nástroje zpracovávající data. Rozhraní repositáře záznamů účtenek je možné vidět v ukázce 6.

CSV generátor

Jednou z pomocných tříd patřící do této vrstvy je generický CSVTABLEGENERATOR (ukázka 7), který pro daný typ (v našem případě entitu) vytvoří metodou GENERATE řetězec ve formátu CSV. V konstruktoru přijímá generátor typu CSVITEMGENERATOR pro danou entitu definující sloupce a získání jejich hodnot pro jeden záznam.

[2] https://en.wikipedia.org/wiki/Uniform_Resource_Identifier

```
interface DeleteReceiptUseCase : UseCase<ReceiptId, Unit>

class DeleteReceiptUseCaseImpl(
    private val receiptsRepository: ReceiptsRepository,
    private val receiptsMLRepository: ReceiptsMLRepository,
    private val backupStorage: BackupStorage,
    private val localImageHandler: LocalImageHandler
) : UseCaseImpl<ReceiptId, Unit>(), DeleteReceiptUseCase {

    // Code executed on invoke() call
    override suspend fun doWork(params: ReceiptId) {
        when (val receipt = receiptsRepository.getReceipt(params)) {
            // Receipt found case
            is SResult.Success -> {
                localImageHandler.delete(receipt.data.imageLocalPath)
                backupStorage.deleteImage(receipt.data.imageRemoteRef)
                receiptsRepository.delete(params)
                receiptsMLRepository.delete(params)
            }
            else -> Unit
        }
    }
}
```

Výpis kódu 5 Případ užití – Odstranění záznamu účtenky

```
/*
 * Repository handling receipts data operations
 */
interface ReceiptsRepository {

    suspend fun getReceipt(id: ReceiptId): SResult<Receipt>

    fun getReceiptsById(ids: List<ReceiptId>): Flow<List<Receipt>>

    fun getAll(): Flow<List<Receipt>>

    suspend fun getAllAsync(): List<Receipt>

    suspend fun save(receipt: Receipt)

    suspend fun delete(id: ReceiptId)

    suspend fun clear()
}
```

Výpis kódu 6 Repository záznamů účtenek

```
/**
  * Simple CSV table generator using defined [CsvItemGenerator] for table
↪  items.
  * @param T is model of CSV file row item
  */
class CsvTableGenerator<T>(private val generator: CsvItemGenerator<T>) {
    fun generate(data: List<T>): String {
        var result: String = generator.getColumns() + LINE_SEPARATOR
        data.forEach {
            result += generator.getCsvRow(it) + LINE_SEPARATOR
        }
        return result
    }
}

interface CsvItemGenerator<T> {
    fun getColumns(): String
    fun getCsvRow(item: T): String
}
```

Výpis kódu 7 Rozhraní CSV generátoru pro tabulku a záznamů této tabulky

V aplikaci je využit k vygenerování tabulky pro report, jejíž záznamy (řádky) jsou jednotlivé účtenky v daném reportu zahrnuty. Tabulka ve formátu CSV řetězce je následně přepsána do souboru a sdílena pro další zpracování mezi aplikacemi na zařízení.

3.2.4 Rozhraní pro rozpoznání obsahu účtenek

Rozhraní ReceiptAnalyzer je navrženo tak, aby bylo možné jeho implementaci jinými (rafinovanějšími) algoritmy či využitím libovolné knihovny. Obsahuje jedinou metodu beginTextAnalysis, pomocí které je možné započít analýzu souboru na dané lokální cestě. Dále vyžaduje, aby implementace umožnila sledování stavu jejího průběhu přes Kotlin StateFlow, které funguje na principu návrhového vzoru *observable*.[3] Tento stav je zaznamenáván v proměnné textAnalyzeResultState.

Možné stavy odpovídají výčtu: nezahájená (*NotStarted*), probíhající (*Analyzing*), selhání (*Failure*) a úspěch (*Success*). Pokud analýza doběhne bez chyb (stav úspěch), vrátí jako výsledek přečtený text po řádcích a asociativní pole[4] (v Kotlinu je tento typ reprezentovaný třídou Map) rozpoznaných údajů. Klíče asociativního pole odpovídají jménu údaje a jejich hodnoty AnalysisItem obalující zpracovanou hodnotu a souřadnice ohraničující oblast, odkud byla tato hodnota rozpoznaná.

3.2.5 Implementace s využitím Google ML Kit

Jelikož z analýzy sady 86 účtenek od řady různých obchodníků bylo zjištěno, že účtenky mezi sebou nesdílí standardní formát, nelze se opřít o strukturu. To by bylo možné tehdy, pokud by byly zpracovávané účtenky pouze od určitého obchodníka, což (bohužel) není cílem této práce a nesplnilo by to potřeby zaměstnanců. Je třeba k problému přistoupit ze strany obsahu. V takovém případě se nabízí dva způsoby – klíčová slova a strojové učení (či jejich kombinace). Druhá

[3] Vysvětlení pojmu observable je možné nalézt na https://refactoring.guru/design-patterns/observer
[4] Vysvětlení pojmu asociativní pole je možné nalézt na https://brilliant.org/wiki/associative-arrays/

```
interface ReceiptAnalyzer {
    /**
     * Result of the most recent text analysis started by [beginTextAnalysis].
     */
    val textAnalyzeResultState: StateFlow<TextAnalyzeEvent>

    /**
     * Starts the text analysis of receipt image at given [imageLocalPath].
     * Result of the analysis can be tracked with [textAnalyzeResultState].
     */
    fun beginTextAnalysis(imageLocalPath: String)
}

/**
 * Events stating current progress of image analysis.
 */
sealed class TextAnalyzeEvent {
    object NotStarted : TextAnalyzeEvent()
    object Analyzing : TextAnalyzeEvent()
    object Failure : TextAnalyzeEvent()
    data class Success(
        val text: List<String>,
        val items: Map<String, AnalysisItem<*>>
    ) : TextAnalyzeEvent()
}
```

Výpis kódu 8 Rozhraní pro rozpoznání obsahu účtenek a možné stavy instance analýzy

možnost nebyla vhodná z důvodů rozsahu práce (a zaměření) a je předpokládané, že by byla potřeba mnohem objemnější předzpracovaná vstupní sada účtenek, aby dosahovala uspokojujících výsledků. Z těchto důvodů je obsah rozpoznávaný dle první možnosti – vyhledávání shod s klíčovými slovy pomocí regulárních výrazů.

Inicializace a předzpracování obsahu

Implementace metody BEGINTEXTANALYSIS začíná načtením obrázku z předané URI cesty, který si sama knihovna ML Kit převede na typ INPUTIMAGE, se kterým pracuje. Pokud by se načtení nezdařilo, skončí analýza neúspěchem. Následně se vytvoří instance třídy TEXTRECOGNIZER z modulu *Text Recogniction* pro rozpoznávání textu z obrázku. Na této instanci se zavolá metoda PROCESS, které se předá načtený obrázek, spouštějící zpracování knihovnou ML Kit. Na výsledek je možné reagovat pomocí *callbacks* (zpětných volání) – ADDONSUCCESSLISTENER a ADDONFAILURELISTENER. Pokud zpracování selhalo, je nastaven stav analýzy na neúspěch.

V opačném případě je poskytnut výsledek analýzy v podobě textových bloků. Tyto bloky obsahují shluky polohově sdružených slov. V případě, že by byla zpracovávaná stránka knihy, by bloky reprezentovaly jednotlivé odstavce. Účtenky mají kvůli svému formátování mnoho mezer a shluky často samy o sobě ztrácejí sémantický výraz. Pro rozpoznání určitých údajů je nutné si tyto bloky převést na řádky. Tím dojde ke sdružení částí, které se nacházejí v odlišných blocích, ale souvisí spolu. Významným příkladem je suma účtenky, kde často klíčové slovo (celkem, k platbě, suma, součet) je na levé straně účtenky a její hodnota na pravé oddělené dlouhou mezerou zapříčiňující rozdělení do odlišných bloků.

```
val recognizer: TextRecognizer =
    TextRecognition.getClient(TextRecognizerOptions.DEFAULT_OPTIONS)

recognizer.process(image)
    .addOnSuccessListener {
        ...
    }
    .addOnFailureListener {
        textAnalyzeStateFlow.value = TextAnalyzeEvent.Failure
    }
```

Výpis kódu 9 Rozhraní pro rozpoznání obsahu účtenek

```
/**
 * Creates line row from lines, which are horizontally aligning
 * but in different blocks, from block data.
 */
private fun List<Text.TextBlock>.toLineRows(): List<LineRow> {
    val result: MutableList<MutableList<Text.Line>> = mutableListOf()
    val allLines = this.map { it.lines }.flatten()

    val mergedLinesIndexes: MutableList<Int> = mutableListOf()

    for (i in allLines.indices) {
        if (mergedLinesIndexes.contains(i))
            continue

        val resultLineList = mutableListOf(allLines[i])
        val topLeftPointY = allLines[i].cornerPoints?.first()?.y ?: 0

        for (j in i until allLines.size) {
            if (j == i || mergedLinesIndexes.contains(j))
                continue

            allLines[j].cornerPoints?.first()?.y?.let {
                if (topLeftPointY + averageLineHeight >= it && topLeftPointY -
                ↪  averageLineHeight <= it) {
                    resultLineList += allLines[j]
                    mergedLinesIndexes.add(j)
                }
            }
        }
        result += resultLineList
    }

    return result
}
```

Výpis kódu 10 Konverze bloku na pseudo-řádky

K přeměně bloků na řádky zahrnující slova na stejné hladině od levého kraje účtenky po pravý bylo přistoupeno rozdělením všech bloků na pole „pseudo-řádků“ (obsažených v těchto blocích). Přes ně se postupně iteruje (prochází) za cílem najít vertikální shodu jejich ohraničujících souřadnic.[5] Při každé iteraci bylo vytvořeno pro daný pseudo-řádek nové pole a pokud byla nalezena zmíněná shoda, byla přidána do tohoto pole a označena, aby byla přeskočena. Tímto vzniklo dvourozměrné pole reprezentující text účtenky po řádcích složených z pseudo-řádků rozpoznaných bloků. Tento typ je zabalen ve třídě LineRow.

Pro snadnější práci byla ještě rozmělněna obalující třída LineRow do jednotlivých slov reprezentovaných třídou Text.Element, z čehož místo pole „pseudo-řádků“ bylo získáno pole slov. Tato struktura je označená typem ElementRow, která navíc obsahuje výšku řádku pro další analýzu.

```
private fun List<LineRow>.toElementLines(): List<ElementRow> =
    map { lineRow ->
        val elements = lineRow.map { it.elements }.flatten()
        val lineHeight = elements.first().boundingBox?.height()
        ElementRow(elements, lineHeight)
    }
```

Výpis kódu 11 Konverze pseudo-řádků na ElementRow

Rozpoznávání údajů

Implementace prototypu rozpoznává tři klíčové údaje:

1. obchodník pomocí metody getMerchant,
2. datum pomocí metody getDate,
3. suma účtenky pomocí metody getTotal.

Metoda getMerchant se snaží rozpoznat obchodníka na principu textového loga nebo výskytu v hlavičce s adresou. Za logo se považuje ElementRow s výrazně vyšší výškou písma – v implementaci alespoň o 20% než průměrná výška napříč celou účtenkou. Pokud takový text nebyl nalezen, obchodník bývá zmíněn ve fakturačních údajích společně s adresou. V tomto případě jsou užitečné bloky, jelikož formát adresy je často rozpoznán v rámci jednoho bloku. Blok s adresou lze rozpoznat podle PSČ, jehož formát (narozdíl od adresy či města) je poměrně unikátní. Pokud je v bloku PSČ rozeznáno, metoda zkontroluje či první řádek by mohl odpovídat nějaké adrese. Pokud ano, bohužel v tomto bloku není jméno obchodníka zahrnuto. V opačném případě by první řádek měl odpovídat právě jménu obchodníka vyplývající z formátu fakturačních údajů.

Metoda getDate se nejdříve snaží hledat datum dle klíčového slova (datum) v řádku a případně ho v něm rozpoznat. Pokud se nepodaří, datum vyhotovení často bývá uveden v hlavičce účtenky, a tedy je hledaná shoda s datovým formátem od shora dolů. Regulární výraz pro datový formát bylo třeba udělat obzvlášť robustní, jelikož jich je mnoho a navíc je třeba zohlednit možné chyby při čtení.

Metoda getTotal rozpoznává celkovou sumu čistě na proiterování řádků a hledání shody s klíčovými slovy pro označení sumy. Mezi tyto označení patří: celkem, k platbě, suma, součet …

[5] Pokud by účtenka byla mírně vychýlená, je použita relativní odchylka k průměrné výšce řádku.

```
private fun getMerchant(
    blocks: List<Text.TextBlock>,
    elementRows: List<ElementRow>
): Pair<String?, MLReadPoints?> {
    //Look for logo, logo is text with at least 20% increased line height that
    ↪  average
    elementRows.take(3).forEach {
        it.lineHeight?.let { lineHeight ->
            if (lineHeight > averageLineHeight * 1.2)
                return it.getLine() to it.getRowBoundaries()
        }
    }

    //get merchant from address - try match block with post code
    //and look for first line, if not street address then its merchant
    blocks.take(5).forEach {
        if (it.match(RegexMatchers.Postcode.regex) != null) {
            val firstLine = it.lines.first()
            val firstLineText = firstLine.text.normalize()
            if (RegexMatchers.Street.regex.find(firstLineText) == null) {
                return firstLineText to firstLine.getLineBoundaries()
            }
        }
    }

    return null to null
}
```

Výpis kódu 12 Rozpoznání obchodníka

```
private fun getTotal(text: List<ElementRow>): Pair<Double?, MLReadPoints?> {
    text.forEach { row ->
        AnalyzerMatchConstants.SUM.forEach { sumString ->
        val line = row.getLine()
            if (line.contains(sumString, true)) {
                val total = matchTotal(line)
                if (total != null)
                    return total to row.getRowBoundaries()
            }
        }
    }

    return null to null
}
```

Výpis kódu 13 Rozpoznání celkové sumy

```
private fun getDate(text: List<ElementRow>): Pair<Long?, MLReadPoints?> {
    // Try to find by date prefix
    text.forEach {
        AnalyzerMatchConstants.DATE.forEach { dateMatch ->
            val line = it.getLine()
            if (line.contains(dateMatch)) {
                val match = regexYYYY.find(line)
                if (match != null)
                    return RegexMatchers.Date
                    .convertMatchToTimestamp(match.value)
                    to it.getRowBoundaries()
            }
        }
    }

    ...

    text.forEach {
        dateMatchers.forEach { matcher ->
            val match = matcher.find(it.getLine())
            if (match != null)
                return RegexMatchers.Date
                .convertMatchToTimestamp(match.value)
                to it.getRowBoundaries()
        }
    }

    return null to null
}
```

Výpis kódu 14 Rozpoznaní data

3.3 Datová vrstva

Datová vrstva se zaměřuje na implementaci rozhraní získávání, ukládání a zálohování dat.

3.3.1 Implementace Repositories

Tato část spočívala v implementaci rozhraní repositářů využívaných v *use-cases* zmiňovaných v 3.2.3. Body implementace jsou: převod mezi síťovými daty (DTO) a doménovými, volání metod datových zdrojů a vypořádání se s chybami. Jako příklad je zvolena ReceiptsRepositoryImpl. Tato třída má jednu závislost ReceiptsLocalSource, což je rozhraní pro lokální zdroj (databázi), jehož implementace je zmíněná v 3.3.2.

Stěžejní je zde blokující[6] metoda getReceipt, která se na vedlejším vlákně pokusí dle předaného parametru načíst data z lokálního zdroje pro účtenku s daným id. Pokud zdroj vrátí data s tímto identifikátorem, metoda vrátí úspěšný výsledek (SResult.Success), který obsahuje doménová data účtenky převedená ze síťových. V opačném případě vrací neúspěšný výsledek s chybou (SResult.Error).

[6] Tyto funkce v Kotlinu poznáme dle klíčového slova suspend.

```
class ReceiptsRepositoryImpl(
    private val receiptsLocalSource: ReceiptsLocalSource
) : ReceiptsRepository {
    override suspend fun getReceipt(id: ReceiptId): SResult<Receipt> =
        withContext(Dispatchers.Default) {
            val receiptDTO = receiptsLocalSource.getReceipt(id)

            return@withContext if (receiptDTO != null) {
                SResult.Success(Receipt(receiptDTO))
            } else {
                SResult.Error(ErrorResult())
            }
        }

    override fun getAll(): Flow<List<Receipt>> =
        receiptsLocalSource.getAll().map { it.map { dto -> Receipt(dto) } }

    override suspend fun getAllAsync(): List<Receipt> =
        receiptsLocalSource.getAllAsync().map { Receipt(it) }

    override fun getReceiptsById(ids: List<ReceiptId>): Flow<List<Receipt>> =
        receiptsLocalSource.getReceiptsByIds(ids).map { it.map { dto ->
        ↪  Receipt(dto) } }

    override suspend fun save(receipt: Receipt) =
        receiptsLocalSource.updateOrCreate(receipt.toEntity())

    override suspend fun delete(id: ReceiptId) =
        receiptsLocalSource.delete(id)

    override suspend fun clear() =
        receiptsLocalSource.clear()
}
```

Výpis kódu 15 Implementace repository záznamů účtenek

3.3.2 SQLDelight Databáze

Lokální persistence dat je řešena pomocí databázové technologie SQLDelight. Přes ni jsou v souborech s příponou „.sq“ zadefinované tabulky a metody reprezentující databázové volání. Během kompilace se nám dle těchto souborů vygenerují třídy pro práci s danou databází. Ty se využijí na implementaci rozhraní lokálních zdrojů.

Pro ukázku je ukázaná definice databáze pro účtenky s tabulkou ReceiptEntity, ze které se vygeneruje třída ReceiptQueries. Využitím jejich metod je možné naimplementovat ReceiptsLocalSource.

3.3.3 Firebase Storage

Pro zálohování fotek záznamů účtenek je využita technologie Firebase Storage, přes kterou je naimplementované rozhraní BackupStorage (ukázka 18). To vyžaduje implementaci dvou metod. Metoda uploadImage zálohuje ve vzdáleném úložišti soubor s lokální cestou localImage-

```
CREATE TABLE ReceiptEntity (
    id TEXT PRIMARY KEY,
    merchant TEXT NOT NULL,
    date INTEGER NOT NULL,
    category TEXT NOT NULL,
    total REAL NOT NULL,
    imageLocalPath TEXT NOT NULL,
    imageDownloadUrl Text
);

getReceipt:
SELECT * FROM ReceiptEntity WHERE id = ?;

getReceiptsByIds:
SELECT * FROM ReceiptEntity WHERE id IN ?;

getAllReceipts:
SELECT * FROM ReceiptEntity;

insertOrReplace:
REPLACE INTO ReceiptEntity VALUES ?;

delete:
DELETE FROM ReceiptEntity WHERE id = ?;

deleteAllReceipts:
DELETE FROM ReceiptEntity;
```

Výpis kódu 16 Deklarace tabulky a metod pro databázi účtenek

Uri pod jménem name a vrací výsledek zálohování SResult. Pokud se zálohování zdařilo, výsledek obsahuje link, kde je možné tuto fotografii stáhnout. Tato informace se využívá v případě, kdyby došlo k odstranění fotografie z lokální galerie. Druhá metoda deleteImage odstraní ze vzdáleného úložiště soubor s názvem z předaného parametru name. Tato metoda se využívá při odstraňování záznamu účtenky, aby vzdálené úložiště nebylo zanášené neplatnými soubory.

Napojení na vzdálený repositář je zpracované přes Firebase SDK. V prototypu je napojení statické pomocí konfiguračního souboru obsahující identifikátor Firebase projektu, identifikátor klienta a „api klíč“. To je možné rozšířit na dynamickou variantu přihlašováním pomocí Google účtu, aby každý uživatel mohl mít své zálohovací úložiště. S tím se počítá v budoucím rozvoji. Pro potřebu zaměstnanců Matee Devs je statická varianta dostačující, jelikož se jedná o sběr účtenek v rámci jedné firmy.

Práce se soubory funguje na základě referenčních objektů obsahující proměnné odpovídající metadatám souborů v souborovém systému. Mezi ně patří i reference na kořenový adresář či potomky, díky nimž je možné se ve vzdáleném repositáři navigovat. Jak už bylo zmíněno v návrhu, knihovna sama řeší proces nahrávání a další operace mezi zařízením a vzdáleným repositářem. Podobně jako u zpracování obsahu účtenky stačí provolat metodu (zde konkrétně putFile) a na výsledek reagovat pomocí callbacku (zde addOnCompleteListener).

```
interface ReceiptsLocalSource {
    suspend fun getReceipt(id: String): ReceiptEntity?
    fun getReceiptsByIds(ids: List<String>): Flow<List<ReceiptEntity>>
    fun getAll(): Flow<List<ReceiptEntity>>
    suspend fun getAllAsync(): List<ReceiptEntity>
    suspend fun updateOrCreate(receipt: ReceiptEntity)
    suspend fun delete(id: String)
    suspend fun clear()
}

class ReceiptsLocalSourceImpl(
    private val receiptsQueries: ReceiptQueries
) : ReceiptsLocalSource {
    override suspend fun getReceipt(id: String): ReceiptEntity? =
        receiptsQueries.getReceipt(id).executeAsOneOrNull()

    override fun getAll(): Flow<List<ReceiptEntity>> =
        receiptsQueries.getAllReceipts().asFlow().mapToList()

    override suspend fun getAllAsync(): List<ReceiptEntity> =
        receiptsQueries.getAllReceipts().executeAsList()

    override fun getReceiptsByIds(ids: List<String>) =
        receiptsQueries.getReceiptsByIds(ids).asFlow().mapToList()

    override suspend fun updateOrCreate(receipt: ReceiptEntity) =
        receiptsQueries.insertOrReplace(receipt)

    override suspend fun delete(id: String) =
        receiptsQueries.delete(id)

    override suspend fun clear() =
        receiptsQueries.deleteAllReceipts()
}
```

Výpis kódu 17 Implementace logiky nad lokální databází

```
/**
  * Remote storage for images backup.
  */
interface BackupStorage {
    fun uploadImage(localImageUri: String, name: String): SResult<String>

    fun deleteImage(name: String)
}
```

Výpis kódu 18 Rozhraní pro zálohování obrázků účtenek

3.4 Prezentační vrstva

Prezentační vrstva je tvořena dvěma vrstvami – View a ViewModel. Zaměřuje se na zobrazování komponent a informací tvořící uživatelské rozhraní – obrazovky, se kterými uživatel interaguje a tyto interakce zpracovává. View vrstva je implementována pomocí knihovny Jetpack Compose. Postupně budou ukázané zajímavé části implementace využitých v obrazovkách (či na pozadí) prototypu. Výsledný vzhled obrazovek je možné nalézt na přiloženém médiu v ukázkovém videu nebo přes instalační balíček je možné si aplikaci nainstalovat a prozkoumat ji.

3.4.1 Navigace

Navigace mezi obrazovkami je implementována pomocí Compose verze navigačních komponent – *Compose Navigation API*. Navigace je reprezentovaná jako graf, kde vrcholy jsou jednotlivé obrazovky a hrany jako definované přechody mezi nimi. Přes hrany se naviguje pomocí komponenty NavController voláním metody *navigate*. Pro zavedení je třeba v kořenové Composable vytvořit komponentu NavHost, která spojuje graf a NavController. V této komponentě se registrují jednotlivé obrazovky, přechody a předávání argumentů mezi nimi.

Aby byla navigace přehledná, pro každou sekci (Účtenky, Reporty, Statistiky, Nastavení) byl vytvořen samostatný vnořený graf. Celkový graf je navázán na BottomBar nacházející se ve spodní části obrazovky, přes který se po kliku na ikonku uživatel naviguje do dané sekce a pohybuje se mezi obrazovkami dle jejího vnořeného grafu. Detailní implementaci je možné najít v kořenové Composable.

```
@Composable
fun Root() {
    val navController = rememberNavController()

    Scaffold(
        bottomBar = { BottomBar(navController) },
    ) {
        CompositionLocalProvider(LocalScaffoldPadding provides it) {
            NavHost(
                navController,
                startDestination = Feature.Receipts.route
            ) {
                ReceiptsRoot(navController)
                ReportsRoot(navController)
                StatisticsRoot(navController)
                SettingsRoot(navController)
            }
        }
    }
}
```

Výpis kódu 19 Zaregistrování navigace

3.4.2 Výřez z obrázku

K tvoření výřezů je využita metoda getValueReadAreaPreview z ReceiptEditFormViewModel. Na základě předané URI načte obrázek ve formátu Bitmap, ze kterého statickou

metodou createBitmap na třídě Bitmap vytvoří nový obrázek reprezentující výřez. Jako parametry přijímá originální obrázek, souřadnice levého horního rohu a šířku s výškou výřezu vypočtenou na základě ostatních hraničních bodů. Z toho vyplývá, že výřezy v tomto prototypu jsou vždy obdélníkové. Příkladný výřez je možné vidět v 3.1.

```
fun getValueReadAreaPreview(imageUri: Uri, points: MLReadPoints, context:
↪  Context): Bitmap {
    val bitmap = imageUri.getBitmap(context.contentResolver)

    return Bitmap.createBitmap(
        bitmap,
        points.points[0].x,
        points.points[0].y,
        points.points[1].x - points.points[0].x,
        points.points[2].y - points.points[0].y
    )
}
```

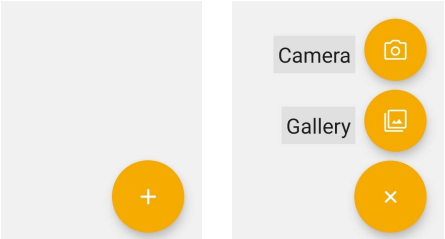
Výpis kódu 20 Výřez z bitmapového obrázku

Obrázek 3.1 Ukázka výřezu v editačním módu záznamu účtenky

3.4.3 MultiAction FAB

Komponenta, která se lehce odklonila od návrhu uživatelského rozhraní, je multi-akční *floating action button* (vznášející se akční tlačítko). V aplikaci se využívá pro akci s několika možnostmi. Důvodem bylo, že tento koncept je uživatelům Android zařízení bližší a dodává důvěrnější pocit z aplikace oproti nestandardnímu pop-up okénku. Samostatný FAB je již navržená kompo-

Obrázek 3.2 MultiAction FAB na obrazovce seznamu účtenek



nenta přímo od Jetpack Compose, která byla rozšířena o parametrizovatelné rozklikávací menu s animací. Jednotlivé položky s jejich akcemi lze definovat pomocí MultiActionFABEntry. Příkladné využití je možné vidět v 3.2 pro zvolení způsobu, jak přidat účtenku.

```
@Composable
fun MultiActionFAB(
    fabIcon: ImageVector,
    entries: List<MultiActionFABEntry>,
    modifier: Modifier = Modifier,
) {...}

data class MultiActionFABEntry(
    val title: String,
    val icon: Painter,
    val action: () -> Unit
)
```

Výpis kódu 21 MultiActionFAB

3.4.4 Swipe to delete

Swipe to delete je funkce, která uživatelům umožňuje mazat položky v jejich seznamu, aniž by museli klikat na tlačítko pro mazání. Uživatelé mohou smazat položku tím, že na ní „přejedou“ prstem odpovídajícím směrem – takzvaným gestem. Jelikož toto chování není rovnou implementované pro LazyColumn (komponenta pro zobrazování seznamů), bylo třeba si jej implementovat pomocí komponenty SwipeToDismiss. Parametry lze nastavit: směr, práh pro vyvolání akce, pozadí pod položkou při přesunu a akci (vymazání položky).

V ukázce je možné vidět implementaci seznamu záznamů účtenek, kde směr posunu je nastavený zprava doleva. Pozadí je nastaveno na Composable DismissReceiptBackground odpovídající červenému obdélníku s ikonou koše znázorňující akci smazání. Při posunu přes práh (ponechána výchozí hodnota) dojde k zavolání funkce deleteAction, které se předá identifikátor záznamu, se kterým se manipuluje. Tato funkce provolá metodu ViewModelu deleteReceipt, která je popsaná v následující sekci.

3.4.5 Detekce účtenky

Jedním z funkčních požadavků je detekce účtenky v reálném čase při fotografovaní. K rozeznání účtenky v obraze je využit modul z Google ML Kit – Image Labeling. Při vytváření spojení mezi aplikací a kamerou je možné připojit obrazový analyzátor. Je definovaný jako rozhraní s metodou

```
...
SwipeToDismiss(
    state = DismissState(initialValue = DismissValue.Default,
    ↪  confirmStateChange = {
        if (it == DismissValue.DismissedToStart) {
            deleteAction(receipt.id)
        }
        true
    }),
    directions = setOf(DismissDirection.EndToStart),
    background = { DismissReceiptBackground() },
    modifier = Modifier
        .padding(horizontal = small, vertical = xsmall)
        .animateItemPlacement()
) {
    ReceiptView(receipt = receipt) {
        onDetailClicked(receipt.id)
    }
}
...
```

Výpis kódu 22 Swipe to Delete implementace

ANALYZE. Ta je při každém zachyceném snímku provolána a je možné jej zpracovat. Je třeba, aby zpracování bylo efektivní, aby nešlo k velkému snížení snímků za sekundu. Naštěstí s tímto případem využití knihovna počítá a vytvořené instanci IMAGELABELER stačí každý snímek v metodě analyze předat. Podobně jako u zpracování textu je možné na výsledek reagovat pomocí callbacku – ADDONSUCCESSLISTENER. Výsledek obsahuje výčet rozeznaných objektů a pokud mezi nimi je i účtenka (objekt klasifikovaný jako Receipt), je uživateli zobrazen informující rámeček.

3.4.6 Implementace ViewModel

Vrstva vystavující data a metody využívané ke zpracování uživatelských interakcí s aplikací propojuje obrazovky a jádro aplikace. Jak je možné vidět v ukázce kódu, ViewModel přijímá ve svém konstruktoru jednotlivé případy užití odpovídající získání dynamického obsahu, který se má uživateli zobrazit, a s ním spojenou manipulaci. V tomto případě jsou vystavená data aktuálních záznamů účtenek přes proměnnou RECEIPTSDATA typu FLOW[7].

Pokud uživatel například pomocí *Swipe to delete* odstraní účtenku, zavolá se metoda DELETERECEIPT, která asynchronně spustí případ užití DELETERECEIPTUSECASE s ID účtenky. Tento případ užití zapříčiní změnu v databázi, která se promítne do RECEIPTSDATA a daný záznam účtenky se z listu odebere.

Plánování pomocí Work Manager

Další funkcionalitou RECEIPTSLISTVIEWMODEL je zálohování fotek záznamů účtenek. Jelikož se jedná o náročnou operaci a není nutné ji vykonat ihned po vytvoření, nabízí se možnost naplánování. K těmto účelům mají vývojáři pro OS Android nástroj WORK MANAGER, který spravuje aktuálně probíhající procesy a plánuje následující dle jejich priorit.

[7] Flow zjednodušuje správu asynchronního načítání dat, díky tomu je snadné získávat aktuální data z databáze.

```
class ReceiptsListViewModel(
    application: Application,
    getReceiptsFlowUseCase: GetReceiptsFlowUseCase,
    private val getReceiptsUseCase: GetReceiptsUseCase,
    private val deleteReceiptUseCase: DeleteReceiptUseCase,
    private val clearReceiptsUseCase: ClearReceiptsUseCase
) : AndroidViewModel(application) {
    ...
    val receiptsData: Flow<List<Receipt>> =
    ↪  getReceiptsFlowUseCase.invoke().map { list ->
        list.sortedByDescending { it.timestamp }
    }

    fun deleteReceipt(receiptId: ReceiptId) {
        viewModelScope.launch {
            deleteReceiptUseCase.invoke(receiptId)
        }
    }

    fun clearData() {
        viewModelScope.launch {
            clearReceiptsUseCase.invoke()
        }
    }
    ...
}
```

Výpis kódu 23 ViewModel pro seznam záznamů účtenek

Počátek zálohování lze vidět ve funkci runBackup. Proces se naplánuje vytvořením žádosti pomocí OneTimeWorkRequestBuilder, které se jako parametr předá třída obsahující logiku, co se má vykonat (pro zjednodušení vynechána). Žádosti se přidělí *tag* (označení) a data – ID záznamu účtenky. Následně pomocí metody beginUniqueWork zavolané na instanci Work Manager se žádost předá a pokud splňuje kritéria pro naplánovaní, Work Manager ji naplánuje. Jakmile doběhnou prioritnější procesy, vykoná se na pozadí. Stav a průběh zálohování je možné sledovat přes proměnnou outputWorkInfo díky označení.

3.5 Dependency Injection

Dependency injection (injektování závislostí) je jedna z forem *Inversion of Control*[8] a návrhový vzor, který umožňuje dynamicky načíst potřebné závislosti a přidělit je k určitému objektu, bez nutnosti tyto objekty propojovat. Injekce závislostí je jeden z nejdůležitějších principů objektově orientovaného programování. Pomáhá zabránit závislostem mezi objekty a umožňuje jim pracovat společně bez toho, aby znaly detailní implementaci toho druhého. Zároveň pomáhá zjednodušit proces testování aplikací, jelikož je možné za objekty, na kterých je daná testovaná komponenta závislá, dosadit zjednodušenou logikou (čímž je například možné se vyhnout řešení závislostí dosazovaného objektu).

[8]Termín označující situace, kdy objekty nevytvářejí jiné objekty, na které se spoléhají při své práci. Místo toho získávají objekty, které potřebují, z vnějšího zdroje.

```
private val workManager = WorkManager.getInstance(application)

val outputWorkInfo: LiveData<List<WorkInfo>> =
    workManager.getWorkInfosByTagLiveData(TAG_IMAGE_BACKUP)

fun runBackup() {
    viewModelScope.launch {
        val receipts = getReceiptsUseCase.invoke()
        receipts.forEach {
            if (!it.isBackedUp) {
                val data: Data = Data.Builder().putString(RECEIPT_ID_KEY,
                ↪  it.id).build()
                val request = OneTimeWorkRequestBuilder<BackUpImageWorker>()
                    .setInputData(data)
                    .addTag(TAG_IMAGE_BACKUP)
                    .build()

                workManager.beginUniqueWork(
                    it.id,
                    ExistingWorkPolicy.KEEP,
                    request
                ).enqueue()
            }
        }
    }
}
```

Výpis kódu 24 Zálohování přes Work Manager

Příkladem v této práci jsou například případy užití, které mají závislosti na repositářích. Případ užití vystavuje rozhraní (*interface*), ale na jeho konkrétní implementaci mu nezáleží. S využitím frameworku spravující injektování zmíníme toto rozhraní a objekt, který se má dosadit (jak ho vytvořit) a pokud by ho bylo v nějakém objektu (či případu užití) třeba, dosadí se. Díky tomu je možné obměňovat implementaci daného objektu – pouze se pozmění specifikace v rámci frameworku, ale není nijak třeba zasahovat do závislých tříd.

Pro tyto účely byl v implementaci zvolen nástroj Koin, jelikož mám s ním zkušenosti a je velice jednoduché ho nastavit. Je nutno podotknout, že není označovaný přímo za DI, ale jako *Service locator*. Rozdíl mezi těmito vzory je, že s lokátorem služeb si hlavní aplikační třída vyžádá objekt od lokátoru. S injekcí není třeba žádného explicitního požadavku, framework automaticky injektuje danou závislost. Nicméně výše zmíněné výhody také přináší a hlavní alternativa Dagger je výrazně složitější, zejména bez předešlých zkušeností.

Ukázku implementace je možné ukázat na případu užití z 3.2.2 DeleteReceiptUseCase se čtyřmi závislostmi, který je třeba injektovat do ViewModelu. Nejdříve si tyto závislosti zaznamenáme v Koin modulech. Zaznamenání je možné pomocí dvou metod – single a factory. Metoda single vytvoří a bude si uchovávat pouze jednu instanci daného objektu – *singleton*. Oproti tomu factory vytvoří vždy novou instanci při každém vyžádání objektu. Syntax je ale pro obě metody stejná. Nejdříve specifikujeme typ, který by měl být žádaný (rozhraní), a jako argument předáme způsob, jak ho vytvořit – jeho konkrétní implementaci. Následně zaznamenáme daný případ užití a místo specifikace, jak vytvořit jeho závislosti, použijeme pouze metodu get. Tím dáváme najevo, že se o ně má Koin postarat.

```
// in android module
val receiptsModule = module {
    single<BackupStorage> { FirebaseStorage() }
    single<LocalImageHandler> { AndroidLocalImageHandler(androidContext()) }
    ...
}
// in shared module
val commonModule = module {
    single<ReceiptsRepository> { ReceiptsRepositoryImpl(get()) }
    single<ReceiptsMLRepository> { ReceiptsMLRepositoryImpl(get()) }
    ...

    factory<DeleteReceiptUseCase> { DeleteReceiptUseCaseImpl(get(), get(),
    ↪  get(), get()) }
}
```

Výpis kódu 25 Koin nastavení pro DeleteReceiptUseCase

3.6 Kotlin Multiplatform Mobile

Vyplývající ze zvolené architektury v 2.2.5 a frameworku KMM, projekt má specifickou strukturu (ukázanou v obrázku B.5 v příloze B). Sdílený kód napříč platformami je možné nalézt ve složce (modulu) SHARED. Ta obsahuje při platformách Android a iOS tři složky se zdrojovými kódy – ANDROIDMAIN a IOSMAIN obsahující definici podpůrných tříd specifických pro platformu (například DI nebo definici databáze) a COMMONMAIN obsahující sdílenou logiku a jádro aplikace. Při zkompilování sdíleného modulu je vygenerována složka BUILD obsahující importovatelné balíčky pro nativní projekty, díky kterým je možné na něj navázat.

Pro platformu Android jsou to soubory ve formátu „.jar“, typické pro jazyky založené na programovacím jazyku Java. Mohou být automaticky generovány při otevření projektu ve vývojovém prostředí Android Studio přes nástroj Gradle. Pro platformu iOS je to balíček ve formátu „XCFramework“, což je aktuálně[9] moderní kompatibilní formát se všemi službami a platformami Apple. Je třeba je manuálně vygenerovat pomocí Gradle příkazu či použití skriptu.[10]

Ve složkách ANDROID a IOS jsou již nativní projekty pro jednotlivé platformy. V nich je možné nalézt implementaci platformně-specifických rozhraních a prezentační vrstvu aplikací. Pro výsledný prototyp této práce je implementovaná pouze Android část. Pro iOS část je připravený zkompilovaný XCFramework balíček ve zmíněné build složce. Pokud by nastaly komplikace při kompatibilitě přeloženého sdíleného kódu s jazykem platformy iOS (Swift nebo Objective-C), počítá se s drobnými úpravami sdíleného kódu. Známé problémy s řešením jsou shrnuty v článku *Interoperability with Swift/Objective-C* [28].

Ve verzi Android prototypu 1.0.1 na přiloženém médiu odpovídá 36,94% sdílenému kódu oproti 63,06% nativnímu kódu. Do tohoto výpočtu byly zahrnuty pouze soubory se zdrojovými kódy a testy. To znamená, že pokud na iOS platformě implementace uživatelského rozhraní a platformně specifických rozhraní odpovídá zhruba podobnému rozsahu, bylo ušetřena nejméně třetina potřebného vývoje pro další platformu. Velkým bonusem je již vyřešená kritická business logika, kterou je navíc potřeba testovat pouze jednou. Tento poměr by mohl být i vyváženější, jelikož až na zálohování fotografií účtenek aplikace neprovádí jiný *networking* (externí komunikace přes internet). Ten je s čím dál rostoucí rychlostí a dostupností internetového připojení (pomocí dat) podstatnější částí procesů na pozadí a patří právě do sdílené části kódu.

[9] k datu 1.12.2022
[10] tento proces je popsán v dokumentaci iOS části

Kapitola 4
Testování

4.1 Způsob testování

Částečné testování uživatelského rozhraní proběhlo v sekci 2.3.3 v rámci low-fidelity prototypu. Z výstupu se poté poznatky a navrhované změny promítly do návrhu uživatelského rozhraní, od kterých se odráží i skutečný design aplikace (až na malé detaily). Z tohoto hlediska je tato kapitola zaměřená na automatizované testy – zejména jednotkové.

Jelikož se jedná o aplikaci se sdíleným kódem, který využívá více platforem, je velice přínosné mít tuto část automatizovaně testovatelnou. Za prvé, testy umožní rychleji vyvíjet, jelikož je možné je spouštět libovolně opakovaně a není třeba chování ručně ověřovat. Za druhé, při vzniku neočekávaného chování či pádech aplikace mohou analyticky pomoci k identifikaci chyby, což nemusí být vždy snadné při multiplatformním vývoji obecně. Za třetí, výhodou KMM je, že společnou logiku je třeba testovat pouze jednou oproti čistě nativnímu přístupu, kde se v praxi často od testování upouští z časových či finančních důvodů. Ale hlavně, chrání před změnou, která by mohla ovlivnit (nebo i „rozbít“) chování na všech platformách.

4.2 Jednotkové testy

Jednotkové testy se soustředí na testování malých částí kódu, především chování jednotlivých tříd (jejich metod). Tyto malé části jsou zkoumány samostatně a pokud fungují správně, je pravděpodobné, že celý systém bude fungovat správně. Jelikož se zaměřují na krátké kusy kódu a mají binární výstup – úspěch či neúspěch, běží krátkou dobu a je možné je libovolně často opakovat.

Pomocí jednotkových testů lze testovat třídy každé jednotlivé vrstvy sdíleného modulu (vrstvy označené „shared“ v 2.7). Nejvhodnějším kandidátem jsou případy užití (*Use cases*), jelikož zahrnují kritickou doménovou logiku aplikace a je to bod, na který navazují platformy implementací nativní části. Zároveň dle zvolené architektury mají závislost pouze na entitách (*Entities*), u kterých se nepočítá se zásadními změnami, a testy by neměly zastarat. Tím se liší od ostatních vrstev, kde je více pravděpodobné, že dojde ke změnám ve „vyšších“ vrstvách a nutnosti tyto testy přepsat či dokonce zahodit.

Pro psaní testů společné části je využita knihovna JUnit4. Důvodem je její snadná použitelnost a dlouhodobě ověřená funkčnost (vydána v roce 2006). Pro simulaci chování komponent, na kterých je případ užití závislý, tak zvaných *mocků* je využita knihovna MocKMP. Tato knihovna přináší implementaci rozhraní knihovny Mockk, jenž je hojně využívaná v nativním Android vývoji, pro KMM. Odkazy na tyto knihovny je možné nalézt na přiloženém médiu.

4.2.1 Ukázka testu případu užití #1

V první ukázce bude testována implementace již zmíněného (v sekci 3.2.2) případu užití DeleteReceiptUseCase. Tento případ užití přijímá jako vstup identifikátor záznamu účtenky, který má být smazán. Pokud je nalezen, je odstraněn a s ním i všechny související data.

Pomocí knihovny MocKMP je možné jednoduše nasimulovat chování závislých komponent, respektive možné stavy, které mohou nastat. Definici takového chování je možné v kódu poznat dle klíčového slova every následující danou metodou a druhým klíčovým slovem returns následované simulovaným výstupem této metody. Postfix suspend se objevuje v případě, kdy simulujeme či kontrolujeme suspend metodu. Tyto metody je možné volat pouze v „blokovatelném“ prostředí. V testech je simulované pomocí metody runBlocking.

Jelikož konkrétní případ užití nemá výstup[1], testování stojí na kontrole provolání metod na závislých komponentách. Pro názornost, pokud záznam účtenky v repositáři dle předaného identifikátoru v parametru nebyl nalezen, nemělo by dojít k dalším provolání na tomto repositáři, ani ostatních komponentách. V opačném případě by na závislých komponentách mělo dojít k provolání „odstraňovacích“ metod.

```
@Test
fun receiptNotFoundTest() {
    runBlocking {
        // Mock receipt with given ID not found behaviour
        everySuspending { receiptsRepository.getReceipt(isAny()) } returns
        ↪  SResult.Error(ErrorResult())

        deleteReceiptUseCase.invoke("ID")

        verifyWithSuspend {
            receiptsRepository.getReceipt(isAny())
        }
    }
}
```

Výpis kódu 26 Testování DeleteReceiptUseCase – záznam účtenky nenalezen

4.2.2 Ukázka testu případu užití #2

V druhé ukázce bude testovaná implementace případu užití getReceiptEditDataUseCase. Tento případ užití slouží k získání souvisejících dat při editaci záznamu účtenky. Na vstupu přijímá identifikátor záznamu a při úspěchu vrací objekt zabalující tyto data. V opačném případě vrací objekt vyjadřující neúspěch.

Simulace chování závislých komponent je obdobná jako v 4.2.1. Jsou testovány 3 možné případy – záznam účtenky nebyl nalezen, data o rozpoznaném obsahu účtenky nebyla nalezena, úspěch. Oproti předchozí ukázce se testuje výstup, konkrétně jeho typ. U prvních dvou je očekáváno selhání (SResult.Error). U třetího je očekávaný úspěch obsahující nalezená data (SResult.Success). Pro kontrolu je využita metoda assertTrue ze standardní Kotlin test knihovny.

[1] není jej třeba, změna odstraněním se projeví ve sledovaných datech (*observable pattern*)

```
@Test
fun receiptFoundTest() {
    runBlocking {
        // Mock receipt with given ID found behaviour
        everySuspending { receiptsRepository.getReceipt(isAny()) } returns
        ↪  SResult.Success(testReceipt)

        // Mock dependencies call behaviour
        everySuspending { receiptsRepository.delete(isAny()) } returns Unit
        everySuspending { receiptsMLRepository.delete(isAny()) } returns Unit
        every { localImageHandler.delete(isAny()) } returns
        ↪  SResult.Success(Unit)
        every { backupStorage.deleteImage(isAny()) } returns Unit

        deleteReceiptUseCase.invoke("ID")

        verifyWithSuspend(inOrder = false) {
            receiptsRepository.getReceipt("ID")
            receiptsRepository.delete("ID")
            receiptsMLRepository.delete("ID")
            localImageHandler.delete(testReceipt.imageLocalPath)
            backupStorage.deleteImage(testReceipt.imageRemoteRef)
        }
    }
}
```

Výpis kódu 27 Testování DeleteReceiptUseCase – záznam účtenky nalezen

```
@Test
fun receiptNotFound() {
    runBlocking {
        // Mock receipt with given ID not found behaviour
        everySuspending { receiptsRepository.getReceipt("ID") } returns
        ↪  SResult.Error(ErrorResult())

        val result = getReceiptEditDataUseCase.invoke("ID")

        assertTrue("Result type mismatch") { result is SResult.Error }
    }
}
```

Výpis kódu 28 Testování getReceiptEditDataUseCase – záznam účtenky nenalezen

```
@Test
fun receiptMLNotFound() {
    runBlocking {
        // Mock receipt ML data with given ID not found behaviour
        everySuspending { receiptsRepository.getReceipt("ID") } returns
        ↪  SResult.Success(testReceipt)
        everySuspending { receiptsMLRepository.getReceiptMLData("ID") }
        ↪  returns SResult.Error(ErrorResult())

        val result = getReceiptEditDataUseCase.invoke("ID")

        assertTrue("Result type mismatch") { result is SResult.Error }
    }
}
```

Výpis kódu 29 Testování getReceiptEditDataUseCase – data o rozpoznaném obsahu účtenky nenalezena

```
@Test
fun dataFound() {
    runBlocking {
        // Mock receipt edit data with given ID found behaviour
        everySuspending { receiptsRepository.getReceipt("ID") } returns
        ↪  SResult.Success(testReceipt)
        everySuspending { receiptsMLRepository.getReceiptMLData("ID") }
        ↪  returns SResult.Success(testReceiptMl)

        val result = getReceiptEditDataUseCase.invoke("ID")

        assertTrue("Result type mismatch") { result is SResult.Success }
        (result as SResult.Success).run {
            assertEquals(data.receipt, testReceipt)
            assertEquals(data.mlData, testReceiptMl)
        }
    }
}
```

Výpis kódu 30 Testování getReceiptEditDataUseCase – úspěšný případ

Kapitola 5
Nasazení

5.1 Vývojové nástroje

V této sekci jsou zmíněné použité nástroje tvořící vývojové prostředí prototypu aplikace a jsou doporučované pro její další rozvoj.

5.1.1 Android Studio

Android Studio je oficiální integrované vývojové prostředí (IDE) pro vývoj aplikací pro Android. Je založené na výkonném editoru kódu a vývojářských nástrojích od IntelliJ IDEA nabízející mnoho užitečných funkcí, které zvyšují produktivitu při vytváření aplikací pro Android. Mezi tyto funkce patří například: flexibilní sestavovací systém založený na technologii Gradle, rychlý a na funkce bohatý emulátor, nástroje pro měření výkonu, použitelnosti, kompatibility verzí a dalších problémů. [29] Vývoj prototypu aplikace probíhal v tomto nástroji od Jetbrains. Jednou z největších předností je jednoduchost instalace (ať už na platformě Windows, MacOS či Linux) a robustnost v odstínění uživatele od složitých konfigurací, kterými by ztrácel čas. Pro vývoj multiplatformní aplikace pomocí KMM je využit plugin Kotlin Multiplatform Mobile, umožňující vývoj i ostatních platforem najednou v tomto prostředí.

5.1.2 GitHub

Pro správu verzí a zálohování zdrojových kódu byla využita technologie Git, konkrétně internetová hostující služba [30] GitHub. Ta byla vybrána na základě preference zaměstnanců Matee Devs. Aplikace dodržuje sémantické verzování. Odkaz na repositář této práce je možný nalézt v příloze. Mimo zdrojových souborů je zde i soubor README.md obsahující popisné informace o projektu, jeho konfiguraci a jak si jej zkompilovat na vlastním zařízení.

Jak už bylo možné vidět v obrázku B.5, projekt je strukturovaný jako jeden „mono“ repositář. Jinými slovy je to jeden projekt obsahující tři moduly – sdílenou logiku, Android aplikaci a iOS aplikaci. Jelikož aplikace je vyvíjená pouze mnou či v budoucnu maximálně menším týmem, byla zvolena v ohledu kompaktnosti. Není třeba spravovat více projektů a řešit mezi nimi napojení. Druhou variantou je rozdělit tyto moduly na tři samostatné projekty a sdílený kód koncipovat jako samostatnou importovatelnou knihovnu. Tato možnost je vhodnější při více platformách nebo vývoji ve větším týmu.

5.2 Firebase App Distribution

Pro distribuci aplikace mezi zaměstnance firmy Matee Devs je využit modul App Distribution od Firebase. Tato volba pramení z již stávajícího napojení aplikace na tuto platformu kvůli zálohovacímu repositáři zprostředkovaným modulem Firebase Storage. Při dokončení nové Android verze se zkompilovaný balíček ve formátu APK nebo AAB nahraje do Firebase a tento modul automaticky rozešle emaily přizvaným testerům. Email informuje o dané nové verzi a poskytuje odkaz na stažení.

Proces nahrávání balíčku do Firebase je zautomatizován pomocí Gradle (nástroj sestavující Android část projektu) příkazu. Ten je možné vidět v ukázce 31 ve dvou verzích. Horní verze zkompiluje a nahraje *release* verzi (verze k vydání) určenou pro koncové uživatele. Dolní verze je určená testerům s povolenými funkcemi pro *debugging* (ladění chyb).

```
// To distribute release version
ReceiptsManager % ./gradlew assembleRelease appDistributionUploadRelease

// To distribute debug version
ReceiptsManager % ./gradlew assembleDebug appDistributionUploadDebug
```

Výpis kódu 31 Gradle příkaz pro distribuci nové verze

5.3 Obchod Google Play

Pro distribuci aplikace dalším uživatelům po dokončení celkového vývoje bude nahrána a volně dostupná v obchodě Google Play. Jedná se o centrální „tržiště“ aplikací všech zařízení s operačním systémem Android. Vývojáři mohou vydávat, spravovat a monitorovat své aplikace přes portál Google Play Console. Nahrání může probíhat ručně, ale lze jej též zautomatizovat propojením s Firebase projektem. Proces zautomatizování obnáší pouze propojení účtu na těchto dvou platformách a jelikož je aplikace již na Firebase napojená, nejedná se o složitý proces.

V době před odevzdáním této práce (16. 2. 2023) je aplikace zde nahrána pouze pro uzavřené testování. Aplikace je v obchodě viditelná uživatelům, kteří obdrželi email s pozvánkou. Tímto způsobem je aplikace nasazená pro zaměstnance Matee Devs na první testovací období. Na základě zpětné vazby a návrhu na vylepšení bude probíhat dodatečný vývoj. Jakmile boudou základní funkce dostatečně odladěny, zpřístupní se všem uživatelům.

Závěr

Cílem této bakalářské práce bylo navrhnout aplikaci pro skenování účtenek s rozpoznáním jejího obsahu a implementovat prototyp pro zařízení s operačním systémem Android. Pro textovou analýzu měla být využita knihovna Google ML Kit. Účel této aplikace je ulehčení procesu měsíčního vyúčtování výdajů zaměstnancům Matee Devs s.r.o. a správy účtenek s ním související. V práci se podařilo splnit všechny požadované body zadání.

Prvním krokem byla analýza domény pro bližší porozumění problému rozeznávání obsahu účtenek. Jejím výstupem byly poznatky o komplexitě vycházející z malé podobnosti struktury účtenek od různých obchodníků, jelikož není nijak standardizovaná, a je třeba se opírat pouze o jejich obsah. Dále byly zanalyzované existující řešení a shrnuty jejich silné stránky a nedostatky. Na základě těchto informací a potřeb zaměstnanců zmíněné firmy byly definované funkční požadavky, nefunkční požadavky a případy užití aplikace.

Po analýze následovala fáze návrhu aplikace. Aplikace byla navrhnuta semi-multiplatformně (nativní UI, sdílený kód pro logiku na pozadí) využitím frameworku KMM a postavením nad Clean architekturou. Dle nadefinovaných požadavků bylo navrhnuto rozhraní aplikace v nástroji Uizard a uživatelsky otestováno zaměstnanci. Jejich připomínky a návrhy byly promítnuty do finálního návrhnu, od kterého se velmi úzce odvíjel design skutečného prototypu. Nakonec byly navrhnuty technologie a nástroje pro realizaci aplikace.

Další fází této práce byla implementace prototypu aplikace. Díky frameworku KMM bylo možné vyvinout prototyp pro OS Android s připravenou sdílenou logikou pro budoucí rozšíření na iOS. Až na pár omezení probíhal vývoj podobně standardnímu nativnímu vývoji, zvolení KMM se vyplatilo. Až na FP13 se povedlo implementovat funkční požadavky s vysokou a střední prioritou a prototyp dodržuje všechny nefunkční požadavky. Kvůli nedostatku času nebyly implementovány požadavky s nízkou prioritou.

Pro rozpoznávání obsahu účtenek byla zvolena metoda hledání shod s klíčovými slovy pomocí regulárních výrazů. Náročnou překážkou bylo zpracování výstupu textové analýzy knihovny Google ML Kit, nad kterým aplikace regulárními výrazy vyhledávala. V důsledku toho a zmíněné různorodosti formátu účtenek se nedaří vždy rozpoznat uvedeného obchodníka. S nalezením celkové sumy a data účtenky si aplikace poradí uspokojivě.

Budoucím rozšířením by mohlo být rozšíření na platformu iOS, pro kterou je připravený zkompilovaný balíček sdílené logiky. Dále doimplementování zbylých funkčních požadavků či, při řádově větší vstupní sadě účtenek, využití algoritmů znalostního inženýrství.

Příloha A

Srovnání nástrojů pro návrh uživatelského rozhraní

Nástroj by měl umožnit s malou náročností vytvořit low fidelity (a potenciálně i medium fidelity) prototyp v doméně mobilních aplikací pro představu, jak by měla aplikace vypadat před implementací. Výběr pochází ze tří srovnávacích článků [19, 20, 21], které byly z aktuálně dostupných na pohled nejvíce propracované. Nástroje sloužící primárně k tvorbě webových stránek, které byly ve srovnávacích žebříčcích zmíněny, budou vynechané. Jednotlivé nástroje jsou zhodnocené v tabulkách dle 4 kategorií:

- cena nástroje k datu analýzy[1],
- zaměření na úroveň fidelity,
- náročnost práce s nástrojem na stupnici 1 až 5, kde 1 znamená lehká bez potřeby se s ním učit a 5 znamená velmi náročná s nutností praxe a učení se,
- užitečné funkcionality (oddělené středníkem).

Adobe Experience Design

Nástroj vyvíjený firmou Adobe, která je známá v této doméně nástrojů tvoření kreativního obsahu, je ve srovnání doporučovaný právě začínajícím uživatelům. Sází na jednoduchost a přívětivost k uživateli pro rychlé tvoření prototypů. Jedná se o desktop aplikaci.

Cena	7 denní zkušební období a poté 10$ měsíčně
Úroveň fidelity	Zahrnuje celé spektrum od nízké po vysokou se zaměřením na design celé aplikace se všemi detaily
Náročnost	1: Jednoduchá, ve všech srovnáních dostává titul nástroje vhodného pro začátečníky
Užitečné funkcionality	Kompatibilita s Photoshop, Illustrator, Sketch; výběr z již předdefinovaných komponent s lehkou funkcionalitou pro lepší představu; možnost designu stavů komponent

U tohoto nástroje je velice zajímavá právě jeho lehká učící křivka, ale jeho zaměření je spíše na high fidelity, což pro prvotní návrh aplikace není relevantní. Je vhodný v případě tvoření komplexního UX/UI celé aplikace a její budoucích komponent.

[1] 26.5.2022

Axure

Axure je UX/UI designový nástroj používaný k vytváření interaktivních prototypů. Užitečný je zejména pro návrháře desktopových a mobilních aplikací a mezi podnikovými společnostmi je uznáván pro vysokou úroveň funkčnosti a vizuálních detailů, které lze zabudovat do jeho prototypů. Jedná se o desktop aplikaci.

Cena	30 denní zkušební období a poté 29$ měsíčně
Úroveň fidelity	Celospektrální s velkou hloubkou
Náročnost	Záleží, jak uživatel nástroj používá, lze velmi jednoduše vyprodukovat low fidelity prototypy, ale zároveň velmi komplexní hi-fi s nutnou znalostí nástroje
Užitečné funkcionality	Bohatá možnost přizpůsobitelnosti pomocí pluginů a knihoven vytvořených komunitou; import assetů z Sketch, Figma, Adobe XD; podporuje podmíněné logické a matematické funkce pro vytváření prototypů řízených daty

Nástroj Axure je jeden z nejvíce používaných v korporátním prostředí. Podporuje velkou přizpůsobitelnost, která firmám vyhovuje. Dobře se s ním pracuje, ale pro efektivní práci je nutné si jej důkladně přednastavit. Má k dispozici mnoho pluginů vytvořených komunitou, ale většina z nich je placená. Tomu bohužel nepřidává jeho vyšší tarifní cena, což pro účely BP není žádoucí.

Balsamiq

Tento nástroj se pyšní svým uvedením na trh již v roce 2008, tedy prověřeností časem a dlouhodobými spokojenými zákazníky. Specializuje se na návrh low fidelity prototypů pro prvotní představu vzhledu aplikací. Jedná se o nástroj s webovým rozhraním.

Cena	30 denní zkušební období a poté 10$ měsíčně pro 2 projekty
Úroveň fidelity	Primárně low fidelity s přesahem do medium fidelity
Náročnost	1: Jednoduchá
Užitečné funkcionality	Uzpůsobenost na tvorbu low fidelity prototypů; stovky předdefinovaných a komunitně vytvořených komponent

Pro účely vytvoření a demonstraci prvotního prototypu bez designových detailů se zdá jako ideální jednoduchý nástroj. Zkušební období 30 dní je dostačujících a dá se brát za zdarma možnost. Mínusem je rozhraní s neaktuálním designem.

Figma

Figma je návrhářský nástroj, který umožňuje vytvářet, spolupracovat a sdílet vektorovou grafiku ve webovém rozhraní. V dnešní době se jedná o jeden z nejrozšířenějších nástrojů pro tvorbu UX/UI napříč všemi platformami. Využívají ho firmy jako Google, Facebook, Uber nebo například Netflix. Jeho oblíbenost pramení z jednoduchého a funkčního provedení aktivní spolupráce na tvorbě a prezentování prototypů.

Tento nástroj je hojně využívaný v praxi, dalo by se hovořit o trendu. Oproti předchozím je náročnější a vyžaduje praxi, protože se jedná o komplexnější nástroj – není na bázi *drag'n'drop* komponent. Lákavý je zdarma plán a potenciální možnost využití v dalších fázích vývoje.

Cena	„Free forever“ plán pro 3 projekty, jinak 15$ měsíčně
Úroveň fidelity	Celospektrální
Náročnost	4: Náročná, je třeba studium, jak se nástroj ovládá
Užitečné funkcionality	Promítnutí grafiky přímo do zařízení pro lepší představu; velká podpora pro kolaboraci více grafiků najednou; komunitní balíčky a pluginy

InVision

InVision je další velmi populární nástroj pro tvoření UX/UI s nadstavbou pro management práce a úkolů. Pyšní se integrací komunikačních platforem a aplikací pro zjednodušení *workflow* (pracovního postupu) vývojářů. Jedná se o nástroj s webovým rozhraním.

Cena	Pro jednotlivce zdarma, pro menší tým 4$ měsíčně pro jednoho člena
Úroveň fidelity	Zaměřená na high fidelity
Náročnost	2: Mírně náročná, je třeba si rozhraní proklikat, případně nahlédnout do dokumentace
Užitečné funkcionality	Podpora pro řízení workflow/spolupráce vývojářů, integrace s aplikacemi jako Slack, Dropbox, Box, Trello, JIRA; velká komunita

InVision má zajímavý plán zdarma pro jednotlivce, díky kterému si uživatel může nástroj zkoušet, jak dlouho potřebuje. Má poutavé a moderní rozhraní. Nicméně jeho silné stránky se vyznačují zmiňovanou integrací nejrůznějších aplikací, což se velmi hodí pro práci v týmu či spolupráci více týmů. Podpora řízení práce a úkolů spolu se zaměřením na high fidelity prototypy je nad rámec potřeb pro tuto práci.

Mockplus

Další oblíbený nástroj ve webovém rozhraní pro prototypování low fidelity prototypů, který se specializuje spíše na rychlost navrhování než robustnost. Tento program umožňuje uživatelům vytvářet interaktivní prototypy s rychlostí a minimálním úsilím.

Cena	15 dní zdarma, poté 200$ ročně
Úroveň fidelity	Celospektrální
Náročnost	2: Mírně náročná, pokud je třeba využít složitějších funkcionalit
Užitečné funkcionality	Podpora pro kolaboraci více uživatelů; hodně předdefinovaných komponent, stylů a ikon ve formátu SVG k využití

Tento nástroj může být zajímavý pro expertní tým, který by využil předdefinovaných komponent pro rychlou tvorbu různorodých prototypů. Svojí cenou patří mezi nejdražší pro jednotlivce až malý tým.

Uizard

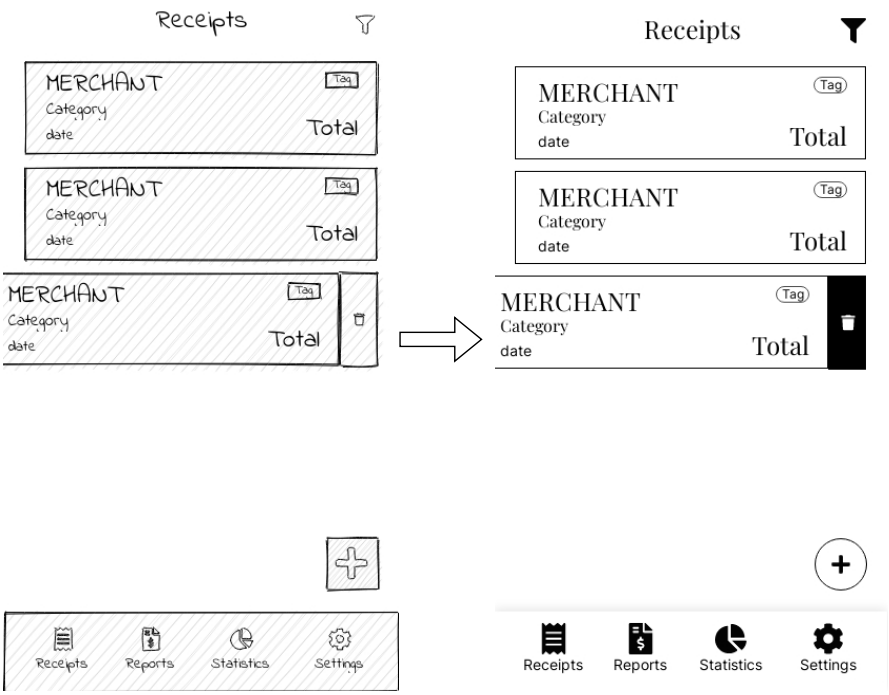
Uizard začal jako výzkumný projekt strojového učení s názvem *pix2code* v roce 2017 v Kodani v Dánsku. Jako společnost se plně zformoval na začátku roku 2018 po výletu do San Francisca a hackerské seanci v garáži Mountain View –- jako skutečné klišé ze Silicon Valley. [22] Jako mladší projekt se mohl odrazit od slabých stránek nástrojů té doby a zároveň inspirovat. Opět slibuje rychlou tvorbu prototypu, kterou zvládne úplně každý bez předešlých zkušeností, ve webovém rozhraní.

Cena	Zdarma pro 2 projekty s omezenou knihovnou, 15$ měsíčně plná verze
Úroveň fidelity	Celospektrální
Náročnost	1: Jednoduchá
Užitečné funkcionality	Kolaborace stejného typu jako u Figma; komponenty mají svou high i low fidelity podobu, mezi kterými lze lehce přepínat; podobně jako u Mockplus velký výběr předdefinovaných šablon; vygenerování prototypu z naskenovaných náčrtků

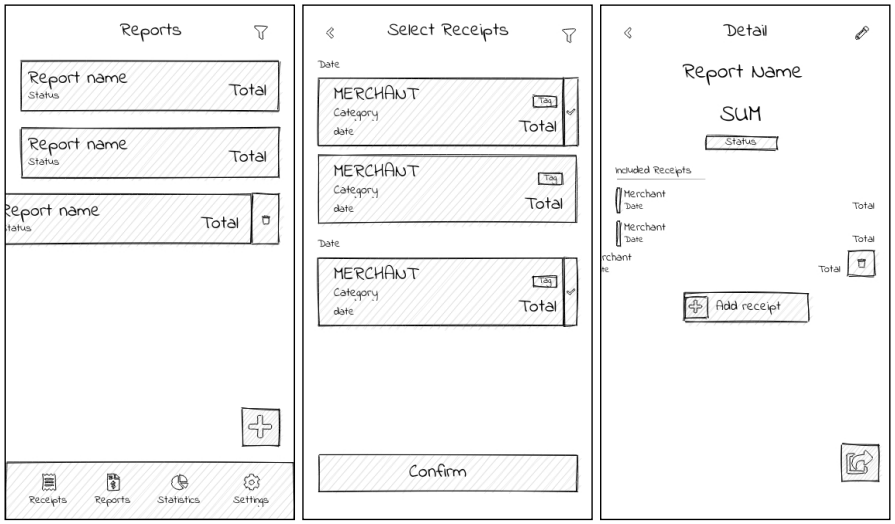
Uizard může velice zaujmout svými webovými stránkami a svojí prezentací. Přestože se jedná o mladý projekt, má už nasbíráno několik referencí od velkých firem. Od pohledu je znát silná inspirace Figmou ve vzhledu rozhraní, nicméně jde cestou jednoduššího prototypování jako například Adobe Experience Design. Unikátní funkcionalitou je možnost tvoření prvotního low fidelity prototypu, pomocí kterého se navrhne koncept rozhraní, a s klikem na jeden přepínač se přegeneruje na grafiku vyšší fidelity, která se dle potřeb dotáhne do finálního produktu.

Příloha B

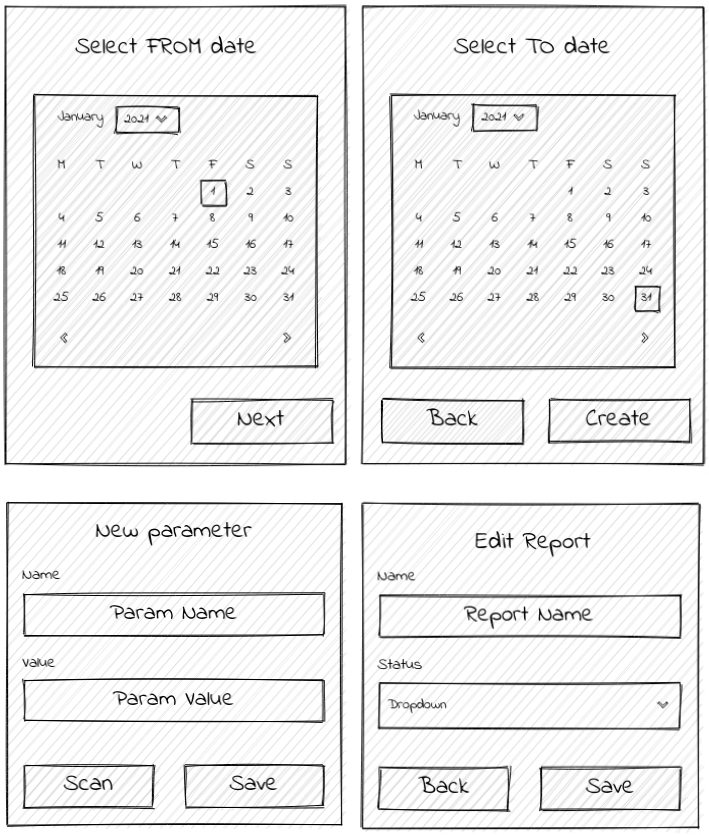
Obrázek B.1 Přepnutí mezi low fidelity a high fidelity bez jakýchkoliv úprav



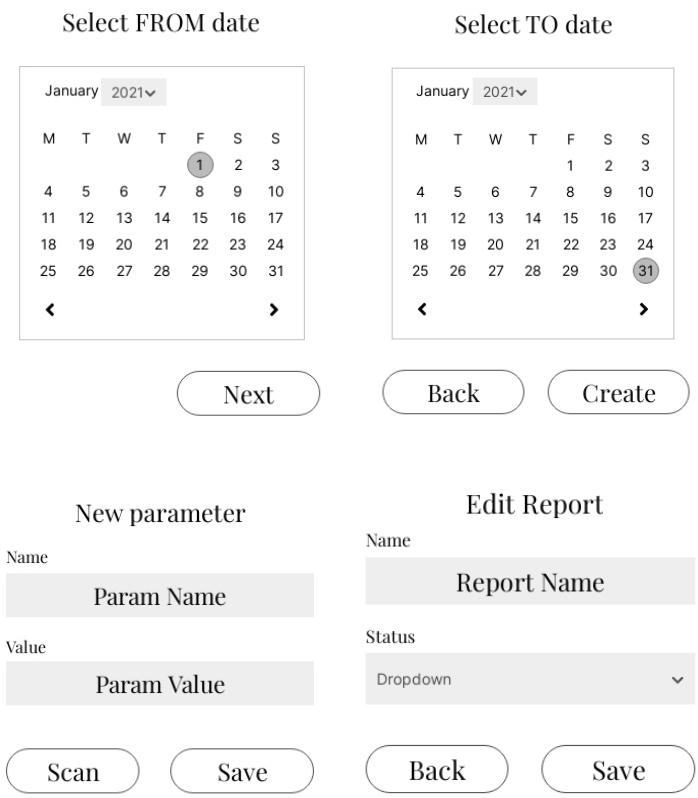
Obrázek B.2 Ukázka návrhu rozhraní procesu vytvoření reportu ručním výběrem účtenek



Obrázek B.3 Low fidelity prototyp – dialogy



Obrázek B.4 High fidelity prototyp – dialogy



Obrázek B.5 Kořenová struktura projektu

```
root
├── android ..... modul implementace Android aplikace
├── buildSrc ..... složka konfiguračního nastavení pro sestavení
├── ios ..... modul implementace iOS aplikace
├── shared ..... modul sdíleného kódu
│   ├── (build) ..... složka obsahující zkompilované balíčky pro jednotlivé platformy
│   ├── androidMain ..... kód definující podpůrné třídy sdíleného kódu pro platformu Android
│   ├── commonMain ..... sdílený kód
│   ├── commonTest ..... jednotkové testy sdíleného kódu
│   └── iosMain ..... kód definující podpůrné třídy sdíleného kódu pro platformu iOS
├── build.gradle.kts ..... kořenový konfigurační Gradle soubor
├── gradlew ..... Gradle skript pro operace s projektem
├── README.md ..... informační textový soubor o projektu
└── ...
```

Literatura

[1] Jaké údaje musí obsahovat účtenka. Https://www.fastcentrik.cz/ [online]. 2021 [cit. 2022-06-12]. Dostupné z: https://www.fastcentrik.cz/blog/jake-udaje-musi-obsahovat-uctenka?feed=blog

[2] Daňový doklad. Wikipedia.org [online]. 2008 [cit. 2022-06-12]. Dostupné z: https://cs.wikipedia.org/wiki/Daňový_doklad

[3] Zákon č. 112/2016 Sb.: Zákon o evidenci tržeb. In: Sbírka zákonů. 2016, ročník 2016, číslo 112. Dostupné také z: https://www.zakonyprolidi.cz/cs/2016-112

[4] Expensify - Expense Reports. Google Play [online]. 2022 [cit. 2022-05-28]. Dostupné z: https://play.google.com/store/apps/details?id=org.me.mobiexpensifyg

[5] Smart Receipts. Google Play [online]. 2022 [cit. 2022-05-28]. Dostupné z: https://play.google.com/store/apps/details?id=wb.receipts

[6] MrReceipt. Google Play [online]. 2022 [cit. 2022-05-28]. Dostupné z: https://play.google.com/store/apps/details?id=pl.primesoft.mrreceipt

[7] Receipt Scanner: Easy Expense. Google Play [online]. 2022 [cit. 2022-05-28]. Dostupné z: https://play.google.com/store/apps/details?id=pl.primesoft.mrreceipt

[8] Foreceipt Receipt Tracker App. Google Play [online]. 2022 [cit. 2022-05-28]. Dostupné z: https://play.google.com/store/apps/details?id=com.foreceipt.android.cloud

[9] TKACHENKO, Igor. Functional vs non-functional requirements: List and examples of systems engineering best practices [online]. The App Solutions, 2019 [cit. 2022-11-05]. Dostupné z: https://theappsolutions.com/blog/development/functional-vs -non-functional-requirements/.

[10] MARTIN, Matthew. What is non-functional requirement in software engineering? types and examples [online]. 2022 [cit. 2022-11-05]. Dostupné z: https://www.guru99.com/non-functional-requirement-type-example.html.

[11] Mobile Operating System Market Share Worldwide. StatCounter [online]. 2022 [cit. 2022-11-05]. Dostupné z: https://gs.statcounter.com/os-market-share/mobile/worldwide

[12] What is cross-platform mobile development? [online]. JetBrains [cit. 2022-11-05]. Dostupné z: https://kotlinlang.org/docs/cross-platform-mobile-development.html

[13] Kotlin Multiplatform Mobile. Kotlin [online]. 2021 [cit. 2022-11-05]. Dostupné z: https://kotlinlang.org/lp/mobile/

[14] MARTIN, Robert C. Clean architecture: a craftsman's guide to software structure and design. Boston: Prentice Hall, [2018]. Robert C. Martin series. ISBN 978-0-13-449416-6.

[15] Web Docs Glossary: Definitions of Web-related terms. Developer.mozilla.org [online]. [cit. 2022-10-25]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/MVC

[16] MVC (Model View Controller) Architecture Pattern in Android with Example. GeeksForGeeks [online]. 2020 [cit. 2022-10-25]. Dostupné z: https://www.geeksforgeeks.org/mvc-model-view-controller-architecture-pattern-in-android-with-example/

[17] Difference Between MVC and MVP Patterns. Www.baeldung.com [online]. [cit. 2022-10-25]. Dostupné z: https://www.baeldung.com/mvc-vs-mvp-pattern

[18] MVVM (Model View ViewModel) Architecture Pattern in Android. Www.geeksforgeeks.org [online]. [cit. 2022-10-29]. Dostupné z: https://www.geeksforgeeks.org/mvvm-model-view-viewmodel-architecture-pattern-in-android/

[19] 10 Best Prototyping Tools For UI/UX Designers [online]. 2018 [cit. 2022-05-26]. Dostupné z: https://medium.theuxblog.com/11-best-prototyping-tools-for-ui-ux-designers-how-to-choose-the-right-one-c5dc69720c47

[20] The 6 best wireframe tools in 2022. Zapier [online]. 2022 [cit. 2022-05-26]. Dostupné z: https://zapier.com/blog/best-wireframe-tools/

[21] 18 Best Prototyping Tools for UI/UX Designers in 2022. Qualaroo [online]. 2022 [cit. 2022-05-26]. Dostupné z: https://qualaroo.com/blog/best-prototyping-tools/

[22] About Uizard. Uizard [online]. 2022 [cit. 2022-11-12]. Dostupné z: https://uizard.io/about

[23] Jetpack Compose. Android Developers [online]. 2022 [cit. 2022-11-12]. Dostupné z: https://developer.android.com/jetpack/compose

[24] What Is SQLite?. Sqlite.org [online]. 2022 [cit. 2022-11-12]. Dostupné z: https://www.sqlite.org/index.html

[25] SQLDelight Overview. Cash App Github [online]. 2019 [cit. 2022-11-12]. Dostupné z: https://cashapp.github.io/sqldelight

[26] Cloud Storage for Firebase. Firebase [online]. 2022 [cit. 2022-11-12]. Dostupné z: https://firebase.google.com/docs/storage

[27] Machine learning for mobile developers. Developers.google.com [online]. [cit. 2022-11-12]. Dostupné z: https://developers.google.com/ml-kit

[28] Interoperability with Swift/Objective-C. Kotlinlang [online]. 3. 2. 2023 [cit. 2023-02-05]. Dostupné z: https://kotlinlang.org/docs/native-objc-interop.html

[29] Meet Android Studio. Android Developers [online]. 2022 [cit. 2022-11-12]. Dostupné z: https://developer.android.com/studio/intro

[30] GitHub. Wikipedia [online]. 2023 [cit. 2023-01-06]. Dostupné z: https://en.wikipedia.org/wiki/GitHub

Obsah přiloženého média

- root
  - src
    - receipts-manager-master ..... zdrojové kódy implementace
    - thesis ..... zdrojová forma práce ve formátu LaTeX
  - ui-design
    - detailed ..... návrh uživatelského rozhraní – detailní verze
    - wireframes ..... návrh uživatelského rozhraní – wireframe verze
  - app-video-demonstration.mp4 ..... video ukázka implementace ve formátu mp4
  - links.txt ..... textový dokument obsahující užitečné odkazy
  - readme.txt ..... stručný popis obsahu média
  - thesis.pdf ..... text práce ve formátu PDF