Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Implementace Android klienta pro webovou aplikaci DCIx

Plzeň 2016 Bc. Tomáš Krásný

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 12. května 2016

Bc. Tomáš Krásný

Poděkování

Děkuji Janu Klikovi ze společnosti Aimtec a. s. za rady a konzultace při tvorbě diplomové práce a Ing. Ladislavu Pešičkovi za odborné vedení mé práce.

V Plzni dne 12. května 2016

Bc. Tomáš Krásný

Abstract

This thesis is related to the area of Enterprise Information Systems. DCIx is a system for companies which are specialized in manufacturing, logistics and distribution. A part of DCIx is a module for warehouse management that is widely used on warehouse enterprise devices. These devices help users with typical warehouse activities to do them fast and effectively.

The main goal of this thesis is to design and implement a mobile application for Android platform. This application can connect to DCIx server and communicate with it. A user can log in to DCIx through this application and see all actions that can be performed. The user can start one of these actions and the application walks the user through each step.

The application is tested with both, unit and instrumented tests for user interface. The applicaion's behavior was also verified and tested on one of the enterprise mobile devices.

Abstrakt

Tato práce se pohybuje v oblasti podnikových informačních systémů. DCIx je systém pro výrobní, distribuční a logistické společnosti. Modul skladového hospodářství systému je hojně využíván na podnikových skladových zařízeních, která svým uživatelům umožňují provádět typické skladové aktivity rychle a efektivně.

Cílem této práce je navrhnout a implementovat mobilní aplikaci pro platformu Android. Výsledná aplikace se umí připojit k DCIx a komunikovat s ním. Aplikace umožňuje uživateli přihlásit se a zobrazit všechny aktivity, které může provádět. Dále aplikace umožní vybrat a spustit nějakou aktivitu a následně provést uživatele jejími jednotlivými kroky.

Výsledná aplikace je otestována jednotkovými testy i testy uživatelského rozhraní a její fungování je ověřeno i na jednom z reálných podnikových zařízení.

Obsah

1 Úvod

Mobilní platformy a řešení postavená na míru mobilním zařízením vybaveným nějakým operačním systémem zažívají v současné době velký vzestup. Pojmem mobilní zařízení nemusí být vždy myšlen chytrý telefon, může se jednat například o různá podniková zařízení, která se díky integraci operačního systému také stávají chytrými.

Díky velkému rozvoji a velké budoucnosti chytrých mobilních zařízení bylo nakonec zvoleno téma, jehož hlavní náplní je implementace mobilní aplikace pro operační systém Android.

Toto externí téma diplomové práce bylo zadáno plzeňskou společností Aimtec a. s. (dále jen *zadavatel*) zabývající se softwarovou podporou navržených a doporučených podnikových procesů. Hlavním produktem společnosti je informační systém DCIx nabízející řešení pro výrobní, logistické a distribuční společnosti. DCIx se skládá z několika modulů, mezi které patří například DCIxWMS (Warehouse Management System) jako systém pro řízení logistických procesů nebo DCIxMES (Manufacture Execution System) jako systém pro sběr dat z výroby.

Modul skladového hospodářství (WMS) se mimo jiné používá i na bezdrátových mobilních terminálech a průmyslových zobrazovacích zařízeních. Jelikož je v současné době rozšířenost těchto zařízení velmi vysoká, objevuje se přirozeně silná poptávka po jejich využití pro přístup k DCIx. Účelem této aplikace je tedy vytvoření nového standardu pro klienty, kteří se budou touto cestou připojovat na DCIx.

1.1 Cíl práce

Cílem práce je navrhnout a implementovat klientskou aplikaci pro zařízení s operačním systémem Android, která uživatelům umožní přihlásit se na DCIx a provádět vybrané úkoly. To zahrnuje definování standardu uživatelského rozhraní, který bude používán na dotykových displejích, a bude jednotný na zařízeních s různým rozlišením a velikostí obrazovky. Výsledné řešení musí splňovat podmínku jednoduché rozšiřitelnosti z důvodu očekávaného nárůstu množiny podnikových úkolů, které bude možné na těchto zařízeních řešit. V neposlední řadě je také cílem práce vytvoření automatizovaných testů.

1.2 Obsah práce

Celá diplomová práce je členěna do několika základních částí. V této kapitole byl stručně uveden účel a cíl práce. Kapitola 2 popisuje obecně kontext, ve kterém se celá diplomová práce pohybuje, tedy podnikové informační systémy, skladové hospodářství a základní systémy a metody, které se v moderním výrobním nebo logistickém podniku používají. V poslední sekci kapitoly 2 je detailněji rozebráno skladové hospodářství, neboť se v této oblasti pohybuje celá diplomová práce.

Kapitola 3 více představí informační systém DCIx včetně jeho zasazení do kontextu práce a dále popíše způsoby, jak lze s DCIx komunikovat a jaké úkoly budou moci uživatelé výsledné aplikace provádět. V poslední části této kapitoly budou představena současná mobilní zařízení a terminály, pro které je aplikace primárně určena.

Kapitola č. 4 blíže představí systém Android a ostatní konkurenční operační systémy, dále jsou popsány hlavní prezentační a aplikační prvky operačního systému a je představena sada vývojářských nástrojů Enterprise Mobile Development Kit pro podniková zařízení od výrobce Zebra Technologies.

Implementaci by měla vždy předcházet analýza problémů a nástin jejich možných řešení. Tímto tématem se zabývá kapitola 5. Jsou v ní analyzovány možné architektury aplikace, jejich porovnání, výhody a nevýhody. Dále kapitola popisuje a vysvětluje principy a účel dependency injection[1] a způsob, jak toho docílit na platformě Android. Poslední částí kapitoly jsou principy a způsoby testování Android aplikací.

Následující kapitola č. 6 je jednou z nejdůležitějších kapitol diplomové práce. Popisuje samotný vývoj aplikace od grafického návrhu, přes návrh vnitřní architektury až po samotnou implementaci. Kapitola také popisuje komunikaci s DCIx, způsob skenování čárových kódů a v neposlední řadě také obsahuje popis implementace jednotlivých obrazovek, se kterými může uživatel interagovat.

Automatické testování je nedílnou součástí každého důležitého projektu. Ani tato aplikace není výjimkou, a proto kapitola 7 popisuje, jakým způsobem lze testovat mobilní aplikace pro Android, jaké jsou k tomu nástroje a knihovny a jak je používat.

V poslední části diplomové práce je výsledek zhodnocen a jsou popsána další rozšíření a vylepšení. Poslední kapitola, kterou je uživatelská příručka, seznamuje čtenáře, jak mobilní aplikaci ovládat a používat.

[1] dependency injection - vkládání závislostí

2 Podnikové informační systémy

Za podnikové informační systémy jsou považovány aplikace a softwarové nástroje, které pomáhají podnikům v dosáhnutí jejich cílů a přispívají k celkovému zlepšení jejich fungování. Cílem podnikového informačního systému je podpořit podnik tak, aby to, čím se zabývá, bylo efektivní.

2.1 Enterprise Resource Planning

Enterprise Resource Planning (ERP) představuje společný název pro podnikové informační systémy (viz obr. 2.1). ERP postupně vznikalo ze samostatných aplikací přidáváním více specifičnosti (např. účetnictví) a vzájemnou integrací. Vše začalo u MRP[1], ze kterého později vyšlo přidáním dalších aplikací MRP II[2], ze kterého ve finále vzniklo ERP. ERP tedy samo o sobě není jediným systémem, který lze v podniku nalézt. ERP se skládá z více subsystémů nebo modulů specifických pro určitou oblast. Obecně ale ERP tvoří jádro podnikového informačního systému [6] a ostatní systémy jsou buď jeho součástí, nebo jsou s ním alespoň propojené. ERP se tím snaží o integraci všech oddělení podniku do jednoho systému včetně jedné společné centrální databáze.

Integrace dalších heterogenních aplikací měla za následek vznik rozšířené verze ERP, ERP II. Do rozšířené verze ERP byly přidány 3 hlavní oblasti. První se týká řízení dodavatelského řetězce (SCM[3]), druhá řízení vztahu se zákazníkem (CRM[4]) a třetí oblastí byl manažerský informační systém zvaný Business Intelligence (BI).

2.1.1 Účel ERP

ERP tvoří aplikace sloužící k řízení podnikových dat a plánování logistického řetězce od nákupu, přes uskladnění až po prodej. ERP dále řídí veškerou práci spojenou se zakázkami, od jejich přijetí, vyhotovení, fakturování až po dodání zákazníkovi. Plánování a řízení výroby je také nedílnou součástí

[1] MRP - Material Requirements Planning (materiálové plánování výroby)
[2] MRP II - Manufacturing Resource Planning (řízení a optimalizace dodávek materiálu)
[3] SCM - Supply Chain Management
[4] CRM - Customer Relationship Management

Obrázek 2.1: Znázornění ERP a jeho možných částí (zdroj: *awaraitsolutions.com*)

ERP, resp. nějaké jeho části za to zodpovědné. ERP pokrývá zejména dvě hlavní funkční oblasti [6]:

- **logistiku**: činnosti týkající se zejména plánování zdrojů, nákupu, skladování, výroby a prodeje,
- **finance**: účetnictví, mzdy, investice a podnikový controlling.

2.1.2 Problémy s ERP

Nejsou to ale jen samé výhody, které ERP přináší. Často se potenciál ERP přeceňuje a rozhodnutí o jeho pořízení nepodléhá důkladné analýze, která je základem pro rozhodování o pořízení nebo změně stávajícího ERP. Špatná nebo nedostačující analýza cílového podniku a jeho procesů může mít za následek nesoulad použitého ERP s potřebami podniku. Takové používání nejen že nemusí přinášet výhody, v horším případě bude jeho provoz dokonce prodělečný.

Při rozhodování o implementaci ERP musí být zapojena všechna oddělení podniku včetně vedení a vrcholového managementu. Jakmile někdo z uvedených chybí, může být implementace nebo následné využívání problematické a zbytečně nákladné.

Pokud je ERP již nasazený, mohou nastat komplikace ze strany uživatelů, kteří nechtějí se systémem pracovat. Důvodem může být špatná ovladatelnost nebo pomalá odezva systému. Další komplikace přinášejí uživatelé, kteří nejsou k ovládání systému dostatečně proškolení nebo nemají dostatečnou motivaci se systémem pracovat. Řešení tohoto problému musí ovšem zajistit management podniku, nikoliv ERP.

2.2 Manufacturing Operations Management

Eliminace plýtvání, efektivita, zvyšování spolehlivosti a spoustu dalšího. To vše vede podniky při neustále rychlejším zavádění nových výrobků na trh k hledání nových řešení, která jim k realizaci těchto požadavků pomohou. Jedním z takových řešení je právě Manufacturing Operations Management (MOM)[13].

MOM systém je centrem řízení výroby jedinečného výrobku na míru. Jedná se o vrstvu v informačním systému, která je umístěna nejníže u samotných fyzických výrobních či logistických procesů. MOM se snaží aplikovat myšlenky dnešního tzv. Industry 4.0, tedy podle [16] *čtvrté průmyslové revoluce*.

Koncept Industry 4.0 byl představen v Hannoveru v Německu roku 2013 a jeho hlavní myšlenkou je vytvoření tzv. *chytré továrny* s digitalizovanou výrobou. To zahrnuje vytvoření počítačových propojení mezi výrobními stroji, polotovary a produkty a všemi dalšími informačními systémy a subsystémy podniku. Toto propojení vytvoří inteligentní distribuovanou síť podél celého výrobního řetězce, ve kterém budou jednotlivé subsystémy pracovat relativně autonomně, dokážou si mezi sebou vyměňovat informace, analyzovat data, předcházet chybám a přizpůsobovat se změnám.

Díky aplikaci myšlenek Industry 4.0 je zřejmé, že systém MOM musí integrovat nějaké další subsystémy, které se budou starat o jednotlivé oblasti v podnicích (např. zajišťování kvality, řízení výroby a další). Takové systémy jsou součástí MOM a konkrétně se jedná o systémy zabývající se:

- skladovým hospodářstvím (WMS - Warehouse Management System),
- řízením výroby (MES - Manufacturing Execution System),
- řízením kvality (QMS - Quality Management System),
- řízením dodávek mezi dodavateli a odběrateli (JIT - Just in Time a JIS - Just in Sequence),
- řízením údržby (EAM - Enterprise Asset Management),

- pokročilým plánováním (APS - Advanced Planning and Scheduling).

Kvůli vysoce specializovaným a konkrétním procesům každého podniku je nezbytné, aby byl systém MOM nasazen vždy na míru konkrétního podniku se zachováním jeho jednoduchého používání. Další důraz je kladen na jeho rozšiřitelnost, pružnost a variabilitu, a to zejména kvůli neustálému vývoji podniku a zdokonalování podnikových procesů.

Před samotným rozhodnutím, jestli by měl být MOM systém v podniku zaveden, se musí zohlednit i náklady na implementaci. Nabízí se otázka, zda podnik všechna řešení ukrytá v MOM opravdu potřebuje. Většina středních a větších podniků už se bez dílčích řešení neobejde, a tak je vhodné implementovat raději jednotný MOM než jednotlivá samostatná řešení typu MES, QMS atd. Důvodem je jednak to, že v implementaci jednoho systému je již zahrnuta implementace dílčích systémů, a zejména to, že celková cena pořízení jednoho integrovaného MOM je výhodnější než pořízení jednotlivých dílčích řešení. Z dlouhodobého hlediska se pak jedná o snížení pořizovacích a provozních nákladů.

Systém budoucnosti

Spolu s Industry 4.0 a jeho realizací v podobě MOM lze na MOM nahlížet jako na systém budoucnosti [13], který:

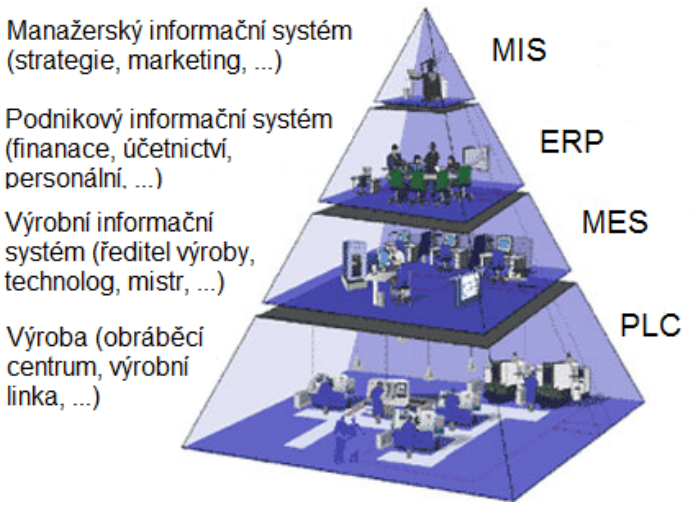
- umí řídit materiálové toky výrobků, polotovarů či jiných komponent,
- umí přebírat instrukce a informace o výrobku z aplikací k tomu určených,
- umí si vyměňovat informace a data se stroji a roboty,
- umí řídit a koordinovat úkoly pracovníka se současným vyhodnocením hodnot provedených úkonů,
- umí plánovat a synchronizovat veškeré zdroje potřebné pro výrobu,
- umí detekovat zastavení činnosti pracovišť a okamžitě upozornit příslušné osoby,
- umí poskytovat přesná a dostatečně detailní data pro manažerské analýzy a BI.

2.2.1 WMS - Warehouse Management System

Systémy skladového hospodářství jsou také součástí MOM, nicméně je jim věnována samostatná sekce 2.3.

2.2.2 MES - Manufacturing Execution System

Manufacturing Execution System, neboli výrobní informační systém, je systém propojující podnikový informační systém a systém pro automatizaci výroby (obr. 2.2). MES primárně komunikuje s nízkoúrovňovými systémy jednotlivých strojů a sbírá z nich data o výrobě [2].



Obrázek 2.2: Zařazení MES mezi podnikové informační systémy [2]

Sběr dat o výrobě není jediným úkolem výrobních informačních systémů. Během celé historie vývoje těchto systémů bylo definováno několik hlavních aktivit systému MES[2]. Jedná se například o správu výrobních zdrojů a postupů, detailní plánování a řízení výroby, různé výkonnostní analýzy a další.

Správa výrobních zdrojů a postupů systémů MES zahrnuje přidělování, plánování a sledování zdrojů a kapacit potřebných pro výrobní proces (osoby, materiál, . . .). Dále zajišťuje evidenci, správu verzí výrobních postupů a výměnu kmenových dat týkajících se výrobků s okolními systémy. Poskytované informace jsou založeny na aktuálním stavu se zohledněním stavu budoucího (např. plánované rezervace zdrojů).

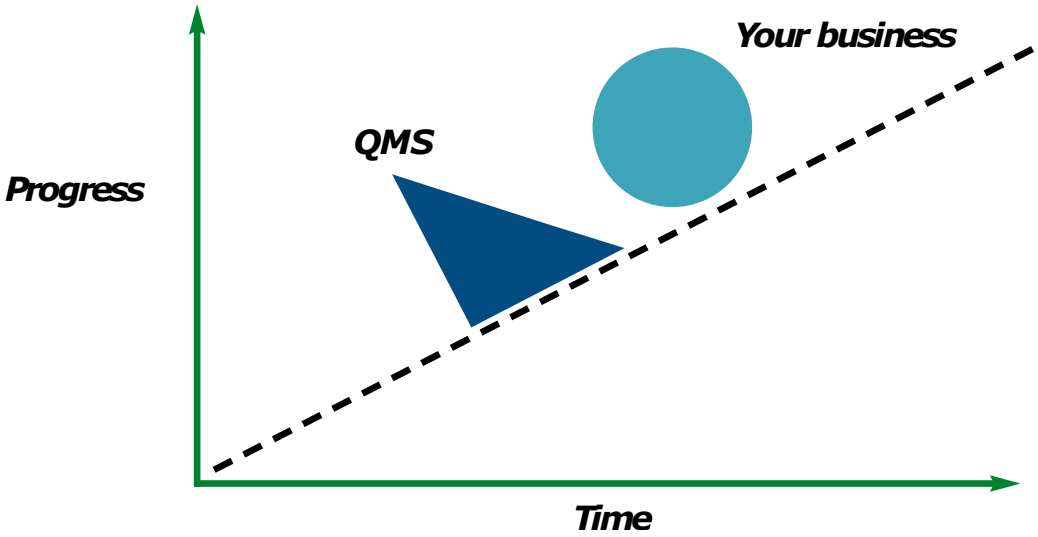
Další kritickou součástí MES je plánování výroby, jehož výstupem je vždy tzv. fronta práce definující pořadí zpracování jednotlivých výrobních příkazů. Plánovacích metod a algoritmů existuje více, za zmínku stojí například dopředné a zpětné plánování výroby, plánování vycházející pouze z priorit čekajících zakázek nebo komplexní plánování založené na genetických algoritmech.

Řízení výroby v MES poskytuje vedlejším informačním systémům informace o aktuálním stavu výroby (např. informačnímu systému typu ERP a jeho uživatelům zejména ve vedení podniku), informace o rozpracované výrobě, počtu dokončených a plánovaných výrobků a další.

Výkonnostní analýzy poskytované systémy MES využívají podniky k vyhodnocování úspěchu buď celého výrobního procesu, nebo jen nějaké jeho části. Obecně nelze říci, že mají všechny podniky hodnotící ukazatele stejné. Často se totiž tyto ukazatele odvíjejí od jejich výrobní strategie. Asi nejznámějším ukazatelem z oblasti výroby je podle [13] ukazatel OEE[5]. OEE je kvantitativní ukazatel efektivnosti zařízení poskytující měřitelné srovnání jednotlivých výrobních zařízení.

2.2.3 QMS - Quality Management System

Quality Management System je systém pro řízení kvality (někdy se uvádí jakosti), jehož úkolem je zajistit dodržování pravidel napříč logistikou a výrobou. Tato pravidla bývají stanovena oddělením kvality. Cílem QMS systémů je přijmout taková opatření, která budou splňovat požadavky zákazníka a díky kvalitě produktů zvyšovat jeho spokojenost.



Obrázek 2.3: Účel QMS systémů [4]

QMS systém musí splňovat [4]:

- **požadavky zákazníka**: podnik musí neustále splňovat očekávání a požadavky zákazníka a musí mu být schopen dodat požadovaný produkt v požadované kvalitě,
- **požadavky podniku**: podnik musí být schopen zajistit, že náklady na výrobu produktu budou optimální a využití dostupných zdrojů bude efektivní.

[5]OEE - Overall Equipment Effectiveness

Jediným důkazem, že požadavky zákazníka a podniku byly splněny, je představení cílového produktu v podobě informací a dat. Tyto informace musí dokazovat, že se QMS systém opravdu podílel na splnění uvedených požadavků, že kvalita produktu je odpovídající a že jsou obě strany s výsledkem spokojeny.

Standardizace QMS

Existují i celosvětové standardy definované managementem kvality. Jedná se o ISO 9000, což je skupina standardů pro QMS vytvořených Mezinárodní organizací pro normalizaci (ISO), která popisuje základy managementu kvality a vymezuje základní pojmy používané v této oblasti. Definované standardy se netýkají konkrétních produktů nebo služeb. Týkají se těch procesů, které je vytvářejí. Tyto standardy jsou obecné a nijak specifické, a proto mohou být použity v jakémkoliv podniku nebo organizaci poskytující nějaké služby. Důležitou roli hraje také norma ISO 9001, která je částí normy ISO 9000. Norma ISO 9001 již popisuje konkrétní požadavky na systém řízení kvality, které podnik musí splnit, aby danou certifikaci získal.

2.2.4 JIT - Just in Time

Just in Time je metoda řízení logistiky, která se snaží o maximální koordinaci mezi dodavatelem na jedné straně a odběratelem na druhé straně tak, aby byly minimalizovány dopravní a skladovací náklady. Metoda se snaží organizovat výrobní linky a dodávat materiál přesně v té chvíli, kdy jej výroba vyžaduje [3]. Tím dochází k minimalizaci pohybu materiálu v podniku a skladových halách. Metoda sama o sobě nepodmiňuje ke svému uplatnění implementaci nějakého informačního systému, naopak se snaží řešení dosáhnout soustředěním se na lidské aspekty. Týká se tedy i změny podnikové kultury.

JIT lze spojovat i s plánováním a řízením výroby, kdy jako nástroj řízení může být použit systém KANBAN (v překladu znamenající „kartička“). Účelem KANBANu je rozdělit výrobu na tzv. „prodavače“ a „kupující“. „Kupující“ dodává „prodávajícímu“ kartičku se svým požadavkem, na kterou „prodávající“ reaguje splněním požadavku neboli dodáním požadovaného výrobku. Požadavkem na uplatnění metody je jednosměrný materiálový tok včetně synchronizace operací [6].

Cíle v podobě *sedmi nul*

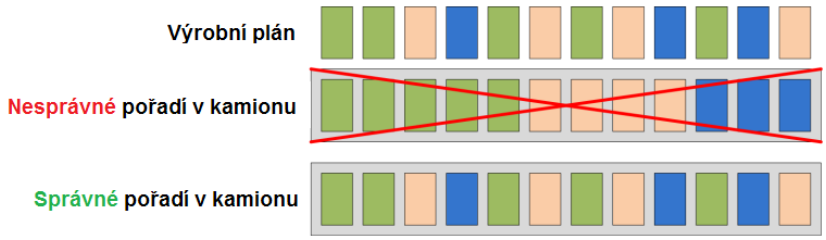
JIT bývá často charakterizováno dosažením tzv. *sedmi nul* [7]:

- nulové množství zmetků (*zero defects*),
- nulové časy seřízení (*zero set-up time*),
- nulové stavy zásob (*zero inventories*),
- žádná manipulace (*zero handling*),
- žádná přerušení strojů (rovnoměrné vytížení, *zero breakdowns*),
- okamžité časy dodávek (*zero lead time*),
- dávky o velikosti jedna (*lot size of one*).

V praxi se podniky výše uvedeným hodnotám spíše přibližují než aby jich přímo dosahovaly [6].

2.2.5 JIS - Just in Sequence

Metoda Just in Sequence úzce souvisí s JIT a dá se považovat za její rozšířenou verzi. Úkolem JIT je vyrábět jen to, co je potřeba, a dodat to včas s nulovými skladovými zásobami. JIS k tomu všemu ještě dodává to, že každá dodávka musí obsahovat jednotlivé položky v pořadí definovaném zákazníkem, ať už se jedná o pořadí jednotek na paletě nebo o pořadí palet v přepravním zařízení, viz obr. 2.4.



Obrázek 2.4: Ukázka uplatnění JIS při přepravě [1] (*pozn.: upraveno*)

S narůstajícími požadavky na flexibilitu, náklady, kvalitu apod. roste i náročnost dodržování obou metod. Pro dnešní dodavatele nikdy nebylo důležitější mít robustní a modulární IT řešení, které bude vyhovovat požadavkům odběratelů. Dodávání v požadovaném pořadí se stalo pro podniky klíčovým požadavkem, který umožňuje udržovat krok s konkurencí [5].

JIS díky svému uspořádání eliminuje časy, které by jinak byly potřebné k manipulaci s jednotkami z důvodu jejich nesprávného pořadí. Důsledkem snížení manipulačních časů je potom i zrychlení výroby, což je jedním z cílů každého podniku.

2.2.6 EAM - Enterprise Asset Management

Enterprise Asset Management neboli Správa majetku a údržby, jiným názvem Řízení majetku a správa kritických aktiv [19] nebo také Správa podnikových zdrojů, je také součástí moderního MOM. Ačkoliv nepatří v oblasti podnikových informačních systémů mezi ty často používané, poskytuje velmi důležité nástroje týkající se správy majetku či jiných výrobních podnikových zařízení (např. výrobní stroje, suroviny, vozový park, ...). Z důvodu spotřebovávání nebo opotřebovávání těchto zdrojů je vyžadována softwarová podpora, která řízení těchto aktivit zjednoduší, zpřehlední a poskytne vhodné informace např. o nákladech vynaložených na údržbu nebo nákupu nového vybavení.

Výhody a nevýhody

Pravidelné kontroly a dodržování postupů při odstraňování výpadků zvyšují životnost podnikových zařízení a zkracují průměrný čas prostojů výrobních linek. Na základě historie prováděných oprav často poruchových zařízení poskytuje EAM detailní analýzy podporující rozhodování, zda-li se poruchový stroj vůbec vyplatí znovu opravovat. Historie těchto zásahů také poskytuje informace o tom, jestli je volba dražšího náhradního dílu z dlouhodobého hlediska výhodná. Další výhodou je lepší organizace práce pracovníků údržby, kdy systém poskytuje kalendář se všemi plánovanými zásahy a aktivitami týkajícími se údržby. Nechybí ani poskytování souhrnných výstupních informací pro vedení podniku nebo podkladů pro ekonomický sektor a tvorbu mezd.

Má-li správa zdrojů a zařízení v podniku nedostatečnou podporu nebo dokonce v podniku úplně chybí, musí být vynaloženy zbytečné náklady, které tyto komplikace, většinou jen dodatečně, řeší. V případě podniků s drahými a složitými výrobními stroji je přímo nezbytné jejich využití a údržbu řídit, jinak by plánování těchto zdrojů ztrácelo význam. Důsledkem by byly nepřesné informace o využití jednotlivých strojů, nerovnoměrné provádění údržby a další. Má-li ovšem údržba dostatečnou podporu v podobě EAM, přináší podniku hned několik pozitiv, které z dlouhodobého hlediska snižují náklady, zvyšují zisk a přispívají k zefektivnění výroby.

2.2.7 APS - Advanced Planning and Scheduling

Advanced Planning and Scheduling neboli systémy pokročilého plánování definuje [6] jako výrobní plánování až po úroveň detailního dílenského zpracování. Systémy na základě výchozích podmínek a vstupních parametrů hledají

pomocí optimalizačních algoritmů a kriteriálních funkcí optimální řešení.

APS systémy na rozdíl od běžných ERP systémů zohledňují tzv. omezené zdroje. Běžný ERP systém totiž neřeší současná omezení v podniku. ERP dokáže naplánovat materiál tak, že při plánování výroby dokáže určit, kdy a jaký materiál bude potřeba a jeho množství. Nezohledňuje ovšem omezení typu omezený počet pracovníků. Tento nedostatek řeší APS, který vždy před začátkem plánování určí požadované zdroje (stroje, dělníky atd. a jejich počty) a při samotném plánování tato omezení nepřekročí [14].

Implementace APS je komplikovaná záležitost. Tento systém navíc není vhodný pro každý podnik. Pro malé podniky není APS vhodný z jednoho prostého důvodu, a tím jsou pořizovací náklady. U středních a větších podniků už může být APS přínosem, vyskytují-li se nějaká dříve zmíněná omezení řešitelná pomocí APS. Nicméně pokud je podnik schopný dosáhnout těchto přínosů nějakým způsobem i bez samotného APS, je jeho implementace zcela zbytečná a přinesla by pouze zbytečné náklady [14]. Základní podmínky pro implementaci APS je podle [20] splnění následujících podmínek:

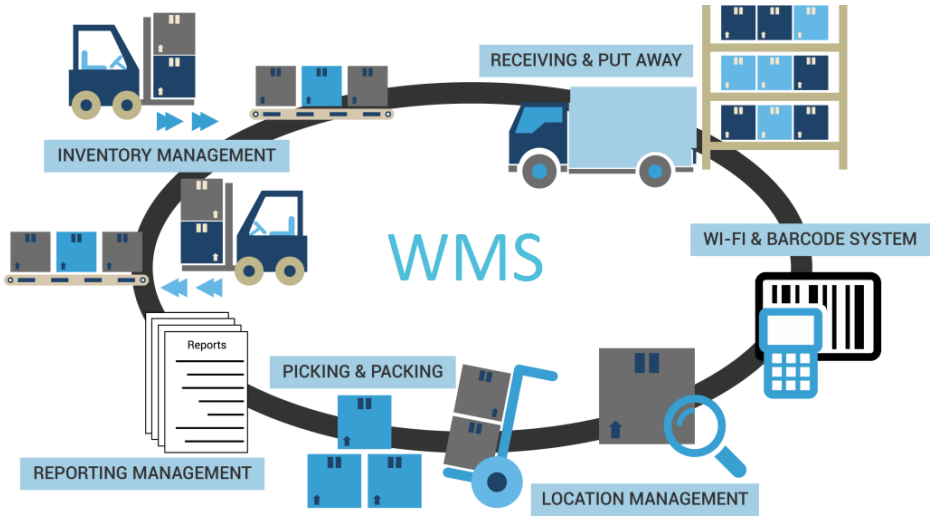
- zakázková výroba (nulové skladové zásoby),
- nákladné výrobní postupy s omezeními,
- soutěžení produktů podniku o výrobní stroje,
- obsáhlý kusovník nebo hodně výrobních operací,
- časté nepředvídatelné změny ve výrobě.

Splňuje-li podnik výše uvedené podmínky, může uvažovat o implementaci APS.

2.3 WMS - Warehouse Management System

Warehouse Management System neboli systém skladového hospodářství je systém sloužící ke komplexnímu řízení logistických procesů ve skladech, distribučních centrech a výrobních společnostech (viz obr. 2.5). Úkolem WMS je usnadnit a zjednodušit provádění skladových operací, zpřehlednit stavy a pohyby zásob, poskytovat přehledné informace o využití skladových prostor a provádět jiné aktivity spojené se skladovým hospodářstvím.

V oblasti WMS probíhá neustálý vývoj kopírující z větší části směr udávaný obecně IT odvětvím. Tím je myšlen postupný přechod od samostatných



Obrázek 2.5: Schéma systému skladového hospodářství (zdroj: *zenventory.com*)

licencovaných produktů ke službám a online řešením prostřednictvím internetu (řešení typu SaaS[6]). Software jako služba je dnes postupně se rozvíjejícím trendem, který je vhodný především pro menší podniky, a to z důvodu nižších pořizovacích a provozních nákladů. Systém není hostován zákaznickým podnikem, ale poskytovatelem, který zodpovídá za veškerou bezpečnost, údržbu a náklady s tím spojené.

2.3.1 Požadavky na systém

Na standardní skladové systémy se klade postupně více a více nároků [23]. Systémy musí podporovat vlastnosti multiklientského prostředí, neboť každý zákazník, který se rozhodne konkrétní systém používat, funguje vždy trochu jinak a jeho procesy se mohou lišit. Systémy tedy musí být co nejvíce univerzální včetně takové míry konfigurace, která pokryje co možná největší spektrum zákazníků. Nejčastěji vyžadovanými procesy po WMS jsou například sledování sériových čísel jednotlivých kusů výrobků, sekvencování (řazení), skenování čárových kódů při manipulaci se skladovanými jednotkami nebo tištění různých forem etiket, zákaznických dokumentů, čárových kódů a štítků.

Po WMS systémech je taktéž vyžadována interakce s okolím. Systémy musí být schopné komunikovat a vyměňovat si informace s okolními informačními systémy. Běžnou praxí totiž je, že každý klient má jiný informační systém, který definuje svůj formát, ve kterém přijímá a odesílá data - ať

[6]Saas - Software as a Service (software jako služba)

už se jedná o textové nebo XML soubory nebo v lepším případě o EDI[7]. Na takovou variabilitu musí být moderní WMS a jeho dodavatel připraven. Předtím, než se dodavatel rozhodne dodávat své WMS řešení, musí být srozuměn s interními logistickými procesy zákazníka (nejčastěji logistických společností), musí znát jeho problematiku a musí rozumět komunikaci logistických společností s jejich zákazníky a dodavateli.

2.3.2 Identifikace zboží a orientace ve skladu

Každá skladovaná položka, manipulační jednotka a skladová pozice musí být jednoznačně identifikovatelná, aby bylo možné provádět plánování a efektivně řídit sklad. Jednoznačná identifikace podporuje provádění skladových operací a zvyšuje jejich rychlost a odolnost k chybám. Za metody pro identifikaci zboží lze považovat:

- **čárové kódy (ČK)**: metoda podporovaná širokou škálou podnikových zařízení umožňujících čtení a tisk čárových kódů,
- **RFID**[8]: méně rozšířená metoda pro identifikaci zboží. Jedná se o štítky, které ke komunikaci s RFID čtečkou využívají rádiových vln.

Čárové kódy mají ve svém rozšíření stále ještě náskok před RFID. Hlavním důvodem jsou především nízké pořizovací a provozní náklady. Mezi hlavní problémy RFID patří vyšší pořizovací náklady (řádově koruny za kus) a různá technická omezení [18], jako například zajištění, aby jeden čip nebyl automatickou čtečkou přečten vícekrát.

Srovnání čárových kódů a RFID

Tato sekce čerpá převážně z [11].

I přesto, že technologie RFID přináší více výhod a lepší využití ve srovnání s čárovými kódy, je stále kvůli svým pořizovacím nákladům méně rozšířenou metodou identifikace. K implementaci RFID je potřeba nákupu RFID štítků, ručních čteček a čteček fungujících jako průjezdní brány vyžadující jejich efektivní rozložení. Náchylnost čárových kódů na poškození ovšem může také způsobit zvýšení nákladů kvůli neustálému přetisku kódů a s tím spojenými časovými prodlevami. Ve srovnání s RFID je to nicméně stále příznivější volba.

Co se týče rozdílů v oblasti jejich použití, přináší RFID významné výhody:

[7]EDI - Electronic Data Interchange (elektronická výměna dat)
[8]RFID - Radio Frequency Identification

- větší kapacita (malá kapacita ČK),
- viditelnost nerozhoduje (ČK musí být viditelné),
- nezávislost na natočení (ČK závislé na linii pohledu),
- editovatelný obsah (nutnost přetisku ČK),
- obousměrná komunikace se čtečkou (ČK jsou jen pro čtení),
- skupinové skenování (jednotlivé skenování ČK).

Technologie RFID přináší i několik výhod v oblasti bezpečnosti. Na rozdíl od čárových kódů, jejichž „výroba“ je jednoduchá (tím je tedy jednoduché i jejich padělání např. v podobě přelepení originálního kódu), umožňuje RFID data zabezpečit nějakým šifrovacím protokolem. S tím souvisí i zabezpečení proti krádežím skladových jednotek, kdy jednotky vybavené RFID štítkem lze sledovat a monitorovat v případě, že opustí svoji skladovou pozici. O neplánovaném pohybu lze poté informovat pověřené pracovníky. Pohyb položky lze za pomoci průjezdních bran sledovat a určit tak její polohu v rámci skladu.

2.3.3 Zařízení a navigace ve skladech

Veškerá zařízení používaná ve skladech jsou propojená s WMS a každé se svým způsobem snaží usnadnit a zefektivnit skladové aktivity. Rozložení každého skladu se liší a každý sklad se dále dělí na menší oddělení, sektory nebo zóny např. podle charakteru skladovaných jednotek. Provádění skladových aktivit musí být optimální, bez jakýchkoli zbytečných ztrát v podobě hledání volné pozice nebo zbytečně dlouhé trasy při manipulaci s jednotkami.

Pro navigaci ve skladu a manipulaci s jednotkami vyžadují pracovníci skladu ke své efektivní práci podporu WMS. Ta se jim dostává pomocí speciálních zařízení, která jsou připojena k podnikovému WMS, který činnost pracovníka řídí. Spolupráce s WMS a skladníkem funguje i v druhém směru, kdy pracovník dává informace WMS o úkonech, které provedl, a jak tyto úkony dopadly.

Času docházelo k postupnému vývoji těchto zařízení od jednoduchých mobilních terminálů až po tzv. rozšířenou realitu. Nejčastěji používané jsou mobilní terminály v podobě různých zařízení, která čtou čárové kódy nebo RFID štítky. Tyto terminály mají displej a klávesnici umožňující pracovníkovi skladu interakci s WMS v reálném čase. Velké oblibě se v poslední době [18] těší tzv. wearable terminály (viz obr. 2.6). Použití těchto terminálů

umožňuje pracovníkům používat obě ruce. Vývoj těchto terminálů díky integraci operačního systému silně zvyšuje jejich potenciál a rozšiřuje možnosti jejich využití. Taková podniková zařízení v sobě obsahují skenery většiny typů čárových kódů a RFID štítků, adaptéry pro Wi-Fi a Bluetooth komunikaci, slot pro paměťové karty a hlavně operační systém, který tuto funkcionalitu zpřístupňuje. Často používané operační systémy v těchto zařízeních jsou Android a Windows Embedded. Některá podniková zařízení obsahují dokonce oba dva a je na uživateli, jaký si vybere.

Obrázek 2.6: Wearable terminál Motorola WT41N0 (zdroj: *zebra.com*)

Hlasové rozpoznávání [18] (tzv. Pick by Voice) se začíná také stávat velmi populárním. Spočívá v komunikaci s WMS za pomoci hlasu a hlasových povelů. Pracovník je vybaven náhlavní soupravou připojenou k terminálu. Ten je schopen na základě instrukcí od WMS generovat lidský hlas a přehrávat jej pracovníkovi, který hlasem odpoví, terminál odpověď zachytí a předá ji ke zpracování WMS. Jedná se tak o přirozenou a pohodlnou spolupráci s WMS, kdy veškerá komunikace probíhá hlasově. Skladníkům v tu chvíli odpadá jakákoliv interakce s terminálem a klávesnicí.

Další strategií vychystávání[9] zboží je Pick by Light. Jedná se o technologii, kdy při vyhledávání vychystávané pozice dojde ve správné chvíli k rozsvícení příslušného indikátoru u hledané pozice, kterých je většinou hodně a jsou na malém prostoru. Pracovník skladu následně odebere zboží a odebrání potvrdí.

Rozšířená realita znamená podle [12] nový trend v logistice. Jedná se zatím pouze o studii, nikoliv o ověřenou praxi. Studie popisuje čtyři hlavní oblasti, ve kterých by mohla být rozšířená realita užitečná.

- **Skladovací činnosti.** Kamery umístěné na hlavě pracovníka mohou rozpoznávat objekty, číst čárové kódy a navigovat pracovníka skladem.

[9]Vychystávání je proces vyskladnění zboží z lokací určených systémem a přípravu tohoto zboží pro balení a expedici.

- **Optimalizace přepravy.** Rozšířená realita může pomáhat při kontrole úplnosti objednávky při jejím přijímání nebo odesílání. Kamery a skenery dokáží zjistit souhrnné informace typu počet palet, jednotlivých položek atd.
- **Distribuce.** Rozšířená realita pomůže řidiči při distribučních činnostech najít hledaných balík a zajistit optimální seřazení ostatních balíků tak, aby byly časové prodlevy minimální.
- **Rozšířená nabídka služeb s přidanou hodnotou.** Díky rozšířené realitě by mohli výrobní pracovníci sledovat vizuální sérii instrukcí vedoucí k nejrychlejšímu provedení daného úkolu.

Výše uvedené příklady jsou pouze ukázkou užití rozšířené reality v praxi. Ve skutečnosti je konkrétních případů mnohem více.

2.3.4 Výhody a nevýhody

Hlavní výhodou nebo spíše přínosem WMS je snížení nákladů a zvýšení produktivity skladu. Pracovníci ve skladu díky WMS nemusí ztrácet čas manipulací s papírovými materiály. Odpadá ruční vkládání dat do systému a dlouhodobé rozhodování, kde ve skladu nalézt jakou jednotku, případně kam do skladu novou jednotku uložit či přemístit. Díky počítačovému zpracování dochází i k menšímu výskytu chyb a vyšší prevenci proti nim.

Největší nevýhodou WMS je jeho pořizovací cena. K tomu je ještě nutné připočítat ceny mobilních zařízení používaných ve skladech, které sice postupně zlevňují, ale stále se nedají považovat za levné. Implementace moderního WMS vyžaduje školení budoucích uživatelů, což lze považovat za další náklady. Tato školení nemusí probíhat jen při implementaci nového WMS. Jsou vyžadována i při aktualizaci současného WMS z důvodu přidání nové funkcionality nebo při změně pracovní pozice pracovníků. Častou praxí je také mírné přeorganizování skladů tak, aby si seděly s implementovaným WMS co možná nejvíce na míru [10].

2.3.5 Příklady procesů

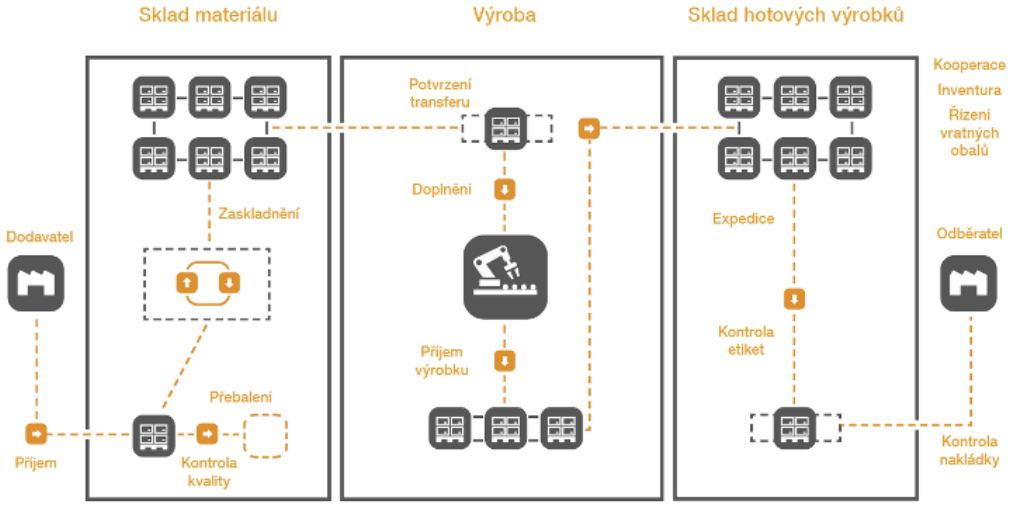
Procesů v oblasti skladování a logistiky je hodně. Zde jsou vyjmenovány jen ty typické pro dané odvětví.

- Skladování: příjem na sklad, vstupní kontrola kvality, zaskladnění a vyskladnění materiálu, přebalení, výdej zboží a další.
- Logistika: interakce s přepravci, procesy s vratky, expedice a další.

3 Informační systém DCIx

Cílem diplomové práce je mobilní aplikace pro platformu Android, která bude komunikovat s informačním systémem DCIx, konkrétně s jeho modulem skladového hospodářství WMS.

DCIx je informační systém společnosti Aimtec a. s. poskytující řešení pro výrobní, logistické a distribuční společnosti. Je unikátní pro celý dodavatelský řetězec, nejenom pro interní logistiku. Systém podporuje instalaci rozšiřujících modulů, které lze přizpůsobovat na míru speciálním požadavkům zákazníka. Systém je plně integrovatelný s ostatními zákaznickými systémy.



Obrázek 3.1: Procesní schéma DCIxWMS společnosti Aimtec a. s.

3.1 Popis a účel DCIxWMS

DCIxWMS je systém skladového hospodářství (o skladovém hospodářství více v 2.3) pro řízení logistických činností a procesů ve skladech, distribučních centrech a výrobních podnicích. Procesní schéma systému popisuje obrázek 3.1. Tok zásob a dokonce i pracovníci logistiky jsou pod detailní evidencí WMS. Systém vše řídí a kontroluje v reálném čase. To přináší pro uživatele přesné informace o aktuálním stavu skladových zásob. Systém přináší efektivní využití skladové plochy včetně optimalizace skladových tras při manipulaci se skladovými jednotkami.

K automatické identifikaci systém využívá čárové kódy a RFID štítky popisované v 2.3.2. Pracovníci skladů jsou vybaveni mobilními terminály

pro čtení identifikátorů a k navigaci po skladu. Výše postavení pracovníci k systému přistupují přes jeho webové rozhraní a veškeré informace mají k dispozici v reálném čase.

3.2 DCIx transakce

Sklady jsou typickým místem s velkým počtem procesů a skladových aktivit. Aby provádění těchto operací bylo efektivní, vyžaduje ke spolupráci podporu WMS (v tomto případě již DCIxWMS). Skladové procesy musí být nějakým způsobem zavedeny v systému, aby mohl práci pracovníků skladu řídit. V tu chvíli přicházejí na řadu tzv. transakce.

Transakce v kontextu DCIxWMS reprezentuje procesy a aktivity prováděné nejčastěji ve skladech a jeho nejbližších oblastech. Transakce se skládají z konečného počtu jednotlivých kroků, které musí pracovník provést, aby transakci dokončil. Jednotlivé kroky mohou a nemusí vyžadovat interakci pracovníka. Typickým příkladem transakce je zaskladnění zboží, kdy pracovník naskenuje kód skladové jednotky, od systému dostane kód skladové pozice, na kterou ji má uložit, a po fyzickém provedení uložení potvrdí.

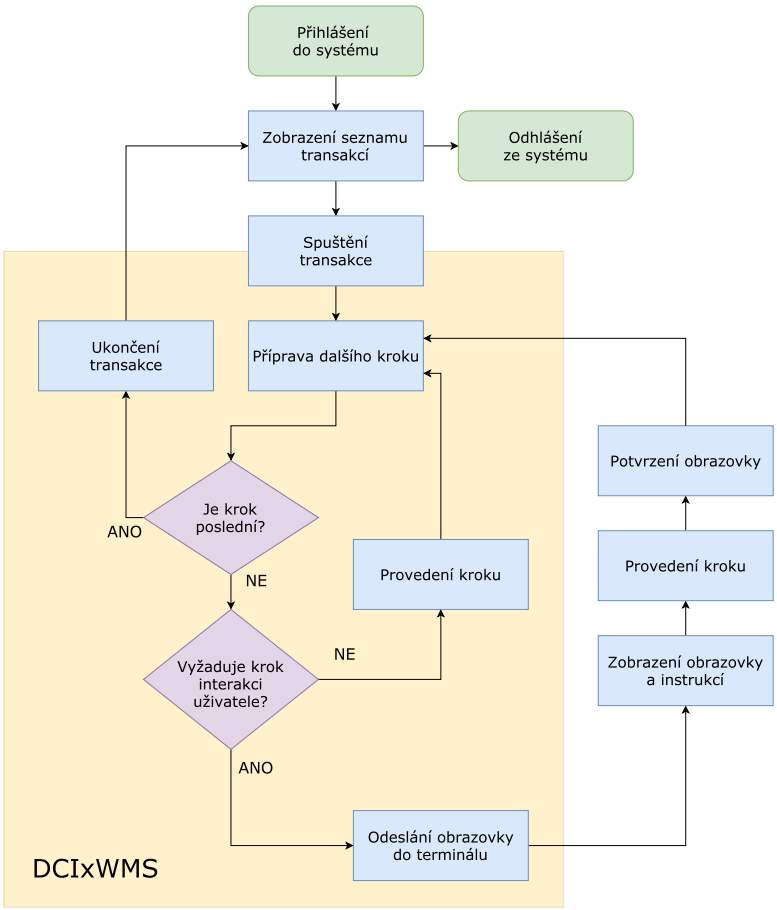
3.2.1 Výběr transakce a oprávnění

Aby bylo vůbec možné transakce spouštět a provádět, musí být pracovník k systému přihlášen. Úspěšné přihlášení v systému vytváří relaci a umožňuje přihlášenému pracovníkovi zobrazit seznam transakcí, které je oprávněn spouštět. Nastavování těchto oprávnění provádí nadřízená osoba. Po dokončení transakce se pracovník vrací zpět na seznam *svých* transakcí, z nichž může opět libovolnou spustit.

Po dokončení práce se pracovník z terminálu odhlásí. V případě neaktivity po definovanou dobu je relace systémem ukončena.

3.2.2 Průchod transakcí

Po spuštění transakce začne systém sekvenčně provádět jednotlivé kroky transakce. Ve chvíli, kdy systém vyžaduje interakci uživatele, je transakce pozastavena a uživateli odeslán aktuální krok v podobě instrukcí, které má provést. Po fyzickém provedení aktuálního kroku pracovník potvrdí, že jej provedl a transakce pokračuje dál. Kroky, které vyžadují interakci uživatele a jsou zobrazovány v mobilních terminálech, se v kontextu DCIx nazývají *obrazovky*. Potvrzení provedení požadovaných instrukcí se v kontextu DCIx nazývá *potvrzení* nebo *odeslání obrazovky*.



Obrázek 3.2: Diagram používání transakcí v DCIxWMS

Po provedení posledního kroku, ať už pracovníkem nebo systémem, je transakce ukončena. Diagram průchodu transakcí od samotného přihlášení k odhlášení je vidět na obrázku 3.2.

3.2.3 Transakční definice

Transakční definicí nebo procesem definování transakce se rozumí definování jednotlivých kroků transakce včetně jejich vstupů a výstupů. Jedná se o předpis zpracování určité události. Jednotlivé kroky v rámci transakční definice se nazývají moduly a jsou sekvenčně očíslované. Moduly mohou být obrazovkové nebo neobrazovkové. Obrazovkové moduly vyžadují interakci s pracovníkem, a proto se zobrazují na displejích mobilních terminálů pracovníků (odtud dříve zmiňované *obrazovky*). Neobrazovkové moduly se provádějí na serveru do té doby, dokud server nenarazí na nějaký obrazovkový modul, který odešle na mobilní terminál.

V DCIxWMS se ze skupiny obrazovkových modulů používají nejčastěji

moduly **scan** a **type**. Modul scan slouží k výzvě pracovníka, aby *něco* naskenoval. Modul type vyzývá pracovníka, aby vložil nějakou hodnotu ručně, z klávesnice. Neobrazovkové moduly, které chtějí zobrazit nějakou informaci (např. modul **hint** pro nápovědu nebo popis aktuálního kroku), se pracovníkovi zobrazí s prvním následujícím obrazovkovým modulem.

Konfigurace modulů

Každý modul podle svého charakteru nabízí určitou možnost konfigurace. Všechny obrazovkové moduly z důvodu interakce s pracovníky skladu nabízí množinu akcí, ze které si pracovník jednu vybere a potvrdí (v kontextu DCIx *odešle transakční obrazovku*). Nejčastěji se jedná o akční klávesy F2 - F8. Z důvodu úspory místa jsou standardně zobrazeny jen akce pro dopřednou (ENTER) a zpětnou (F3) navigaci v transakci. Klávesa F1 slouží k zobrazení všech ostatních akcí. Klávesa F9 slouží k odhlášení pracovníka ze systému. Příkladem akcí vázaných na akční klávesy může být přechod na následující nebo předchozí transakční obrazovku. V transakční definici se u akční klávesy vybere pořadové číslo modulu, na který se má při stisku dané klávesy přejít.

Většina transakčních modulů je vybavena možností nastavení návratového bodu při chybě nebo varování. Jakmile při zpracování transakčního kroku dojde k chybě, skočí se na modul s pořadovým číslem uvedeným v tomto poli a chybová hláška se zobrazí na prvním obrazovkovém modulu po cíli odskoku.

Moduly je možné ještě více parametrizovat, deaktivovat, aktivovat nebo kopírovat. V případě modulů pro vstup dat od pracovníka skladu lze určit, jaká pole pracovník vyplňuje, jaký je jejich datový typ a jestli je vyplnění pole povinné. Pokud by povinné pole nevyplnil, server to vyhodnotí jako chybu, přejde na modul definovaný jako návratový bod při chybě a ten včetně chybové hlášky zobrazí.

Některé parametry modulu mohou být také jen pro čtení, kdy hodnota v nich mohla být vyplněna dříve a nyní už se nesmí měnit.

3.3 Komunikace s DCIxWMS

Veškeré informace v této sekci nejsou součástí praktické části této práce.

S DCIxWMS se dá komunikovat prostřednictvím webového prohlížeče nebo nějakého mobilního klienta, kterým je většinou nějaký mobilní terminál. Tato diplomová práce rozšiřuje skupinu klientských zařízení o zařízení s operačním systémem Android. Všechna tato klientská řešení potřebují se serverem nějakým způsobem komunikovat.

S DCIx lze komunikovat pomocí SOAP webových služeb, prostřednictvím telnetu nebo prostými HTTP dotazy. Z důvodu složitější implementace SOAP klienta pro Android byly zvoleny prosté HTTP dotazy.

3.3.1 Datová struktura

Odpověď od serveru chodí ve formátu XML. Datová struktura odpovědi od serveru je bez větších detailů následující:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <action>
3   <hints></hints>
4   <inputs></inputs>
5   <errors></errors>
6   <submits></submits>
7 </action>
```

Kód 3.1: Kostra XML odpovědi

Element `action` je kořenový element dokumentu a reprezentuje jednu transakční obrazovku. Jeho vnitřní elementy detailněji popisují obsah transakční obrazovky a jsou vysvětleny dále.

Ukázku konkrétní odpovědi popisuje ukázka č. 3.2. V ukázce kódu je vidět konkrétní použití všech polí transakční obrazovky. Konkrétně se jedná o jednu nápovědu, dvě pole pro vstup dat, jednu chybu, jedno varování a nakonec 2 akce sloužící k odeslání obrazovky. Každá přijatá informace, ať už se jedná o vstupní pole nebo akci, má své jedinečné ID v poli `name`. To slouží k identifikaci pole při odesílání obrazovky.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <action name="Jméno testovací transakce/5.type.type">
3    <hints>
4      <hint name="hint1" title="Nápověda k této obrazovce"/>
5    </hints>
6    <inputs>
7      <input name="itemCode" title="Kód zboží"
           renderType="MANDATORY" value=""/>
8      <input name="quantity" title="Množství"
           renderType="MANDATORY" value="1"/>
9      <input name="length" title="Délka" renderType="TYPING"
           value=""/>
10   </inputs>
11   <errors>
```

```
12      <error code="E01" title="Text chyby" type="error"/>
13      <error code="W05" title="Text varování" type="warning"/>
14    </errors>
15    <submits>
16      <submit name="continue" title="Pokračovat"/>
17      <submit name="F3" title="Zpět"/>
18    </submits>
19  </action>
```

Kód 3.2: Ukázka odpovědi ze serveru

Popis atributů jednotlivých komponent zprávy

Každý element obsahuje alespoň jeden atribut s dodatečnou informací. Popis všech atributů všech elementů:

- `action` - transakční obrazovka,
  - `name` - informativní jméno akce v následujícím tvaru `A/B.C.D`, kde `A` je jméno transakce, `B` je sekvenční číslo modulu, `C` je jméno modulu a `D` je jméno podkroku. Poslední krok transakce je identifikován jménem akce ve tvaru `A/finishTransaction`,
- `hints` - jednotlivé nápovědy sloužící především k zobrazení pokynů pro pracovníka,
  - `name` - ID nápovědy,
  - `title` - text nápovědy,
- `inputs` - vstupní pole obrazovky,
  - `name` - ID pole,
  - `title` - titulek pole,
  - `renderType` - popisuje *editovatelnost* pole, možné hodnoty jsou:
    - `VISIBLE` - hodnota jen pro čtení,
    - `TYPING` - hodnotu lze měnit,
    - `MANDATORY` - hodnota je povinná,
  - `value` - hodnota pole, nově nebo dříve vložená,
- `errors` - výjimky detekované serverem, dělí se na řízené a neřízené,
  - `code` - kód chyby (pouze v případě řízené chyby),

  - `title` - text chyby/varování,
  - `type` - typ výjimky, možné hodnoty jsou:
    - `error` - jedná se o chybu,
    - `warning` - jedná se o varování.

- `submits` - akce, z nichž lze jednu vybrat k odeslání obrazovky,
  - `name` - ID akce,
  - `title` - titulek akce (často popisek tlačítka).

3.3.2 Případy komunikace

Konkrétní případy komunikace s DCIx jsou pouze 4: přihlášení, spuštění transakce, další krok transakce a odhlášení.

Přihlášení

HTTP dotaz při přihlašování musí obsahovat dva parametry: `username` s uživatelským jménem a `password` s heslem. Volitelným parametrem je parametr `logOver`, který, je-li uveden a je-li přihlašovaný uživatel už přihlášen, je jeho relace násilně ukončena a je vytvořena nová. HTTP dotaz jde na URL `adresa_serveru/s_logon.do`.

Konkrétně se jedná o metodu `POST`, kdy přenášené parametry jsou v těle metody (z důvodu omezené délky URL v případě metody `GET`). Jelikož zadavatel nijak nespecifikoval větší bezpečnost, je k přenosu dat použit prostý nezabezpečený `HTTP` protokol[1].

Odpověď serveru při úspěšném přihlášení obsahuje seznam transakcí, na které má přihlášený pracovník právo. Přijatý seznam transakcí je obsažen v kontejneru `hints` a jednotlivých elementech `hint`. Atribut `name` obsahuje identifikátor dané transakce a atribut `title` obsahuje titulek transakce. Jméno akce obsahuje konstantu `logon`. Kontejner `hints` obsahuje jediné pole, kterým je identifikátor vytvořené relace. Pokud byl uživatel už přihlášen, vrací se v kontejneru `submits` jediná akce, a to `logOver`.

Při neúspěšném přihlášení vrací server chybu, která byla příčinou neúspěšného přihlášení (např. nesprávné heslo nebo že uživatel je již přihlášen).

[1]Použitá třída `HttpURLConnection` používá `HTTP/1.1`, viz `http://developer.android.com/reference/java/net/HttpURLConnection.html`

Odhlášení

Dotaz odhlášení jde na URL `adresa_serveru/s_logoff.do` a jeho jediným parametrem je `ID` s identifikátorem relace přihlášeného pracovníka. Odpovědí je akce se jménem `logoff` a prázdným polem `sessionId`. Jedinou chybu, kterou server může vrátit, je pokus o odhlášení nepřihlášeného uživatele (neexistující nebo nenalezené `sessionId`).

Spuštění transakce

Transakce se spouští zasláním HTTP dotazu na URL `adresa_serveru/s_transactionStart.do`. Povinným parametrem je identifikátor relace přihlášeného pracovníka, jinak je vrácena chyba, že pracovník není přihlášen. Druhým povinným parametrem je `transactionDefinitionName`, což je identifikátor transakční definice, která má být spuštěna. Transakční definice musí být registrována v DCIx a uživatel musí mít právo jí spustit. Není-li tomu tak, vrací server chybu. Je-li už nějaká transakce spuštěná, je tímto voláním ukončena bez garance uložení rozpracovaných dat.

Odpověď serveru už vypadá tak, jak je popsáno v 3.3.1. Jméno přijaté akce obsahuje titulek spuštěné transakce včetně pořadí a jména aktuálního modulu. Při vykonávání mohou nastat chyby, například „nepřihlášený uživatel“, „kód transakční definice nenalezen“ nebo „nedostatečná práva pracovníka ke spuštění transakce“.

Další krok transakce

Nejčastěji posílaným dotazem je dotaz na další obrazovku transakce. Ten je odesílán na URL `adresa_serveru/s_nextPageTransaction.do`. Dotaz obsahuje vyplněné vstupy, identifikátor akce, která byla k odeslání aktuální obrazovky použita, a identifikátor současné relace. Server následně provede navigaci na další krok transakce, která vyžaduje interakci klienta. Tato obrazovka je poté odeslána klientovi jako odpověď. Konec transakce je identifikován jménem akce ve tvaru `<jméno transakce>/finishTransaction`. Před odesláním dotazu musí být pracovník přihlášen, jinak server vrací neřízenou chybu.

Odpověď od serveru vypadá opět podobně jako v 3.3.1.

3.4 Koncová zařízení

Řízení skladů se v dnešní době nemůže obejít bez koncových mobilních zařízení, která usnadňují, respektive umožňují práci se skladovým systémem.

Těchto mobilních zařízení je několik druhů (například zařízení do ruky, náhlavní zařízení nebo zařízení do manipulačních vozíků; více v 2.3.3) a kladou se na ně určité požadavky. Jednou z nezbytných podmínek je flexibilita zařízení. Skladové hospodářství vyžaduje zařízení, které pracuje bez připojení do elektrické sítě, tedy na baterii. U těchto zařízení je taktéž žádoucí, aby umožňovala připojení k firemní síti, ať už prostřednictvím Wi-Fi sítí, nebo jinak. Nicméně existují také *off-line* varianty zařízení, které žádnou konektivitu neposkytují a s podnikovými systémy komunikují až po připojení k počítači. Tato zařízení nejsou již tak rozšířená, neboť po informačních systémech je vyžadována dostupnost dat v reálném čase, a proto i koncová zařízení musí poskytovat informace a komunikovat se systémem v reálném čase. Opačný postup je značně neefektivní a může být zdrojem chyb a zbytečných nákladů.

Sklady a skladové haly bývají často považovány za neklidná a aktivní prostředí. Z toho důvodu je po zařízeních často vyžadována robustnost, odolnost proti prachu a zejména odolnost proti pádům a různým nárazům, které se v tomto odvětví často vyskytují.

3.4.1 Bezdrátová podniková zařízení

Podobně jako mobilní telefony, i koncová skladová zařízení se neustále vyvíjejí a jejich potenciál se zvyšuje. Díky integraci operačních systémů se zařízení stávají chytrými a poskytují mnohem větší možnosti jejich využití.

K vývoji aplikace k této diplomové práci bylo od zadavatele zapůjčeno zařízení (nebo, jak jej výrobce nazývá, *mobilní počítač*) Symbol MC9200 (viz obr. 3.3). Jeho výrobce, Symbol Technologies, je dceřinou společností Zebra Technologies, proto je někdy zařízení označováno jako Zebra MC9200[2].

Jedná se o velmi pokročilé podnikové zařízení, které je díky svým vlastnostem považováno současně za jedno z nejlepších. Zařízení je dodáváno v různých konfiguracích s různými operačními systémy: Android KitKat 4.4.4, Windows Embedded Compact 7, nebo Windows Embedded Handheld 6.5.3. Zatímco uvedené operační systémy Windows jsou cíleny pro malá podniková mobilní zařízení, je operační systém Android spíše zákaznicky orientovaným operačním systémem. Z toho důvodu je k němu dodávána sada Mobility Extensions (Mx). Mobility Extensions je předinstalovaná sada nástrojů a funkcí, které standardní verze systému Android nemají a díky kterým začne být i Android vhodný pro použití v podnikovém prostředí. Jako příklad lze uvést schopnosti skenování dat pomocí vestavěných skenerů a čteček.

[2] `www.zebra.com/us/en/products/mobile-computers/handheld/mc9200.html`

Obrázek 3.3: Zařízení Symbol MC9200, na kterém probíhal vývoj (zdroj: zebra.com)

Co se týká hardwarové konfigurace, jsou zařízení osazena dvoujádrovým procesorem OMAP 4. generace běžící na frekvenci 1GHz. Dále 512MB RAM (prémiová verze 1GB) a 2GB flash paměti s možností rozšíření až na 32GB díky slotu pro SD karty. Pro zajímavost, zařízení váží 765 g včetně baterie (2400 mAh).

Zařízení mají tlakový displej o velikosti 3,7 palce s rozlišením 640x480. Dále mají Bluetooth a bezdrátové připojení se standardy `IEEE 802.11 a/b/g/n/d/h/i` a v případě systému Android také `k` a `r`.

Zařízení podlehla nárazovým zkouškám z výšky 1,8m na beton a splňují IP64.

4 Platforma Android

Android je operační systém v současné době vyvíjený společností Google s otevřeným zdrojovým kódem založený na Linuxovém jádře. Systém je primárně určen pro zákaznickou komunitu využívající mobilní zařízení jako mobilní telefony nebo tablety (dnes už i hodinky, televize a automobily). První verze systému 1.0 byla vydána 23. září 2008 [17] a neměla ještě své kódové označení. Poslední oficiálně vydanou verzí je verze s kódovým označením Marshmallow 6.0 vydaná 5. října 2015.

K zajištění kompatibility mezi aplikacemi a zařízeními existují tzv. revizní čísla. Každá verze Androidu včetně svých „podverzí“ má své revizní číslo. Revizní číslo, v terminologii Androidu `API level`, je celé číslo, které jednoznačně identifikuje verzi API Android frameworku. Toto číslo není shodné s verzí Androidu tak, jak je obecně známo (např. Marshmalow **6.0**, API level 23). API level je používán zejména při programování aplikací k jednoznačné identifikaci verze API frameworku. Při vývoji aplikace lze specifikovat minimální API level, na kterém aplikace poběží. Například aplikace s minimálním API levelem 16 nemůže běžet na zařízení s takovou verzí Androidu, jejíž API level je nižší než 16.

Další sekce v této kapitole již budou věnovány konkrétním prvkům a nástrojům, které buď byly při vývoji použity, nebo je vhodné je díky své důležitosti vysvětlit.

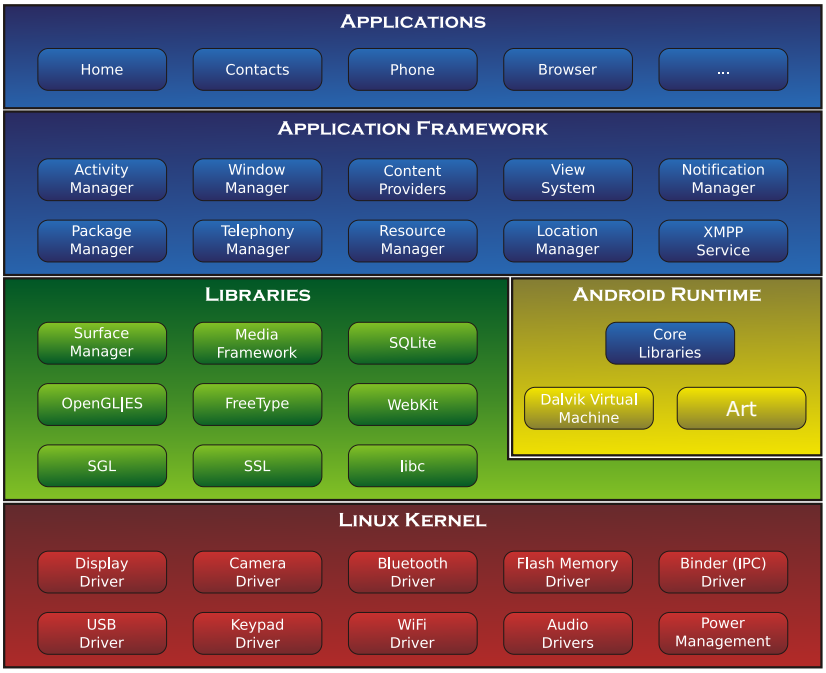
4.1 Architektura systému

Systém je postaven na Linuxovém jádře, což je nejnižší vrstvou architektury, viz obr. 4.1. Vrstva obsahuje ovladače, které umožňují vyšším vrstvám používat hardware zařízení. Vrstva zabezpečuje například správu paměti nebo provádění síťových operací a obsahuje ovladače například pro ovládání kamery zařízení, zvuku a dalšího.

Nad Linuxovým jádrem je vrstva s knihovnami napsanými v jazyce C. Tato vrstva poskytuje nástroje opět pro vyšší vrstvu. Jedná se například o knihovnu pro bezpečnost internetové komunikace (SSL[1]) nebo o SQLite (databázový systém pro ukládání dat).

Další vrstva zabezpečuje běh aplikací a liší se z hlediska historie vývoje systému. Jedná se o dvě běhová prostředí, z nichž jedno postupně nahrazuje

[1]SSL - Secure Sockets Layer



Obrázek 4.1: Architektura operačního systému Android (*zdroj: edureka.co, pozn.: upraveno*)

druhé. První běhové prostředí se nazývá Dalvik. Dalvik je virtuální stroj, který umí spouštět Dalvik byte kód, na rozdíl od JVM, který spouští Java byte kód. Zdrojové kódy aplikace jsou nejdříve přeloženy do Java byte kódu a následně do Dalvik byte kódu, který je nezávislý na architektuře cílového zařízení. Po instalaci aplikace probíhá její spuštění vytvořením nové instance DVM (Dalvik Virtual Machine), ve které aplikace poběží. Před samotným spuštěním Dalvik vykonávaný kód pokaždé kompiluje do nativního kódu zařízení - JIT (Just-in-time) kompilace.

Druhým a novějším běhovým prostředím je ART (Android Runtime), který na místo JIT kompilace využívá tzv. AOT (Ahead-of-time) kompilaci, neboli dopřednou kompilaci. Ta spočívá v kompilaci byte kódu Dalviku do nativního kódu zařízení již při instalaci aplikace. Kompilace je tedy jednorázová záležitost prováděná při instalaci.

Největší výhodou ART s dopřednou kompilací je zvýšení rychlosti spouštění a vykonávání aplikace, a to z toho důvodu, že kód byl přeložen již při instalaci a není tedy nutné jej kompilovat v průběhu běhu aplikace znovu. Výhodou Dalviku jsou menší nároky na paměť, neboť nekompiluje vždy celý kód jako ART, ale segmenty, které zrovna potřebuje. Nevýhodou ART je delší doba instalace, a to právě kvůli dopředné kompilaci.

ART se začal do Androidu dostávat postupně od verze 4.4 v podobě voli-

telného běhového prostředí se zachováním standardního běhového prostředí v podobě Dalviku. Uživatel měl tedy na výběr mezi Dalvikem a ART. Od verze 5.0 byl Dalvik kompletně nahrazen prostředím ART.

Předposlední vrstvou je aplikační framework systému. Jsou to nástroje, se kterými pracují samotní vývojáři mobilních aplikací, pokud chtějí, aby jejich aplikace měla danou funkčnost, například notifikace prostřednictvím správce notifikací (Notification Manager) nebo využívání polohových služeb prostřednictvím správce lokace (Location Manager).

V poslední (nejvyšší) vrstvě jsou samotné aplikace zařízení, ke kterým mají uživatelé přístup.

4.2 Prezentační prvky

Tato sekce čerpá z [15] a popisuje prezentační prvky Androidu použité v této práci.

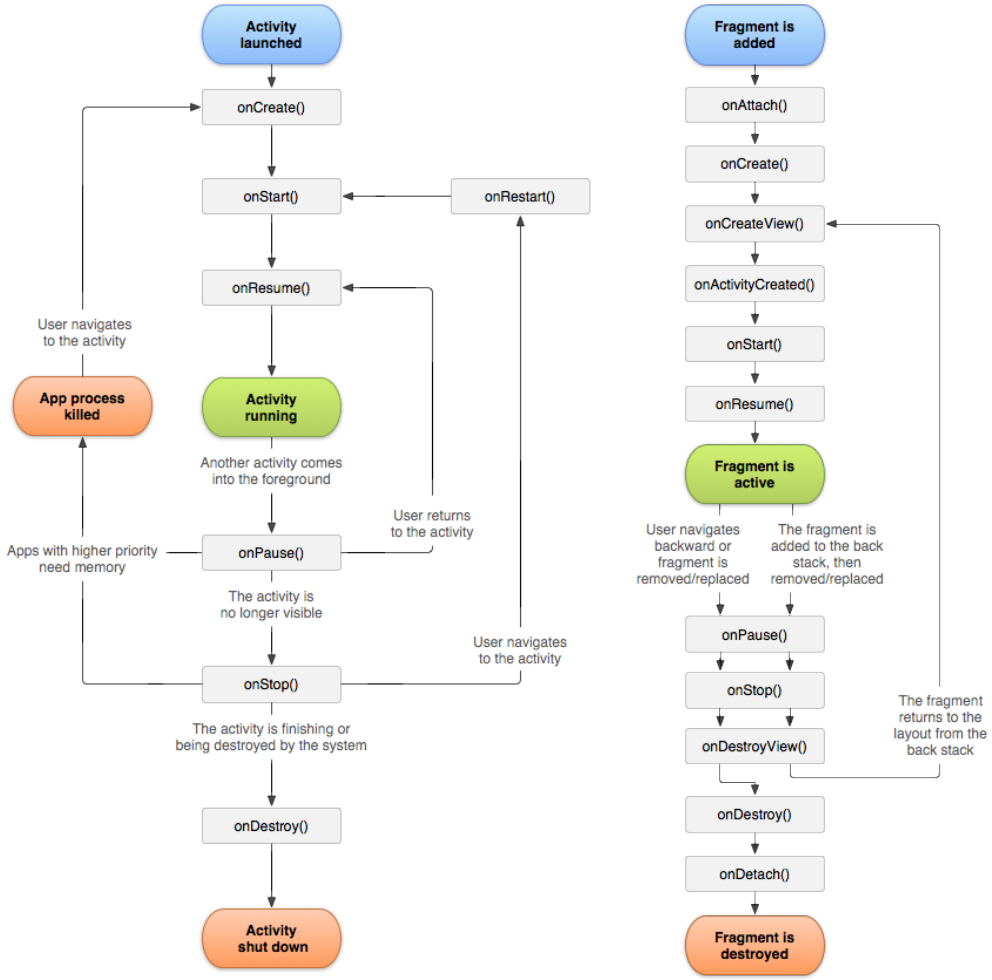
4.2.1 Aktivity a fragmenty

Aktivita je komponenta systému, která reprezentuje obrazovku, na kterou se zrovna uživatel dívá a se kterou může interagovat. Příklad aktivity může být seznam kontaktů, detail kontaktu, vytáčecí dialog atd. Aplikace tedy standardně obsahují více aktivit, které se různě střídají podle úkonů uživatele. Otevře-li uživatel nějakou aktivitu z jiné, je ta nová standardně zobrazena nad tou původní, není-li specifikováno jinak. Aktivity se takto staví do zásobníku (v terminologii Androidu tzv. *back stack*). Pokud uživatel novou aktivitu jakkoliv ukončí (např. tlačítko *Zpět*), je nová aktivita zrušena a původní, která byla překryta, je zpět obnovena a zobrazena.

Fragmenty jsou komponenty uvnitř aktivit zobrazující části obrazovky. Aktivita může mít žádný až několik fragmentů. Fragmenty jsou znovupoužitelné, to znamená, že více aktivit může obsahovat stejný fragment, tudíž zobrazovat stejná data. Příkladem může být opět situace s kontakty. První fragment může zobrazovat seznam kontaktů a druhý může zobrazovat detail vybraného kontaktu. To lze prakticky využít například při zobrazení na šířku, kdy je celá obrazovka rozdělena na dvě části, fragment v levé části zobrazuje seznam kontaktů a fragment v pravé části zobrazuje detail vybraného kontaktu.

Aktivity i fragmenty se postupně zobrazují, skrývají a různě mění svou viditelnost, mají tedy každý svůj životní cyklus (viz obr. 4.2). Životní cyklus (dále ŽC) fragmentů závisí na aktivitě, ve které jsou. Je-li aktivita pozastavena (např. něčím překryta), jsou pozastaveny i její fragmenty. O všech

těchto situacích informuje systém aktivitu a fragment voláním metod jejich ŽC, aby na situaci mohly patřičně reagovat (například uložit svůj stav nebo odeslat dodatečná data).



(a) Životní cyklus aktivity (b) Životní cyklus fragmentu

Obrázek 4.2: Životní cykly komponent (zdroj: *developer.android.com*)

Vytvoření nové aktivity nebo fragmentu se provede děděním od patřičné třídy z Android frameworku a přepsáním callback metod ŽC, kde je třeba provést nějakou akci. Po vytvoření aktivity je nutné ji zapsat do **manifestu** aplikace. Manifest je XML soubor, který poskytuje základní informace o aplikaci včetně jejích aktivit.

Nejvýznamnější metody ŽC aktivity jsou:

- `onCreate()` - volána, když je aktivita vytvářena (v této metodě se nastavuje rozložení, layout, aktivity),

- `onStart()` - volána, když se má aktivita zobrazit,
- `onResume()` - volána, když se aktivita stane viditelnou (na rozdíl od `onStart()`, kdy může být částečně překryta např. dialogem),
- `onPause()` - volána, když aktivita ztratila focus, např. kvůli dialogu,
- `onStop()` - volána, když aktivita už není viditelná,
- `onDestroy()` - volána, když je aktivita a její objekt zrušen.

Velmi podobně to funguje u fragmentů, kde významnou metodou je `onCreateView()`, která je volána, když má být obsah fragmentu vykreslen. Ostatní metody se shodným názvem fungují velmi podobně. Například metoda `onResume()` fragmentů je volána těsně po `onResume()` u aktivity. Naopak `onPause()` u fragmentů je voláno těsně před `onPause()` u aktivity.

4.2.2 UI elementy a kontejnery

Každá aktivita a fragment skládají svůj obsah z jednotlivých elementů. Tyto elementy jsou organizovány do speciálních kontejnerů, které podle svého typu definují, jak budou elementy v kontejneru rozvrženy. Základní třídou v Androidu pro tvorbu uživatelského rozhraní je třída `View`, od které ostatní třídy dědí. Veškeré kontejnery jsou zase potomky třídy `ViewGroup`, která je sama potomkem `View`.

Nejčastěji používanými elementy jsou textové nápisy (`TextView`), pole pro vstup dat (`EditText`) nebo tlačítka (`Button`). Mezi nejčastěji používané kontejnery pak patří `LinearLayout` pro vertikálně nebo horizontálně zarovnaná `View`, `RelativeLayout` pro zarovnávání `View` relativně vzhledem k nějakému jinému `View` nebo `ScrollView`, které umožňuje rolování, je-li obsah moc velký.

Různých dalších `View` a `ViewGroup` je celá řada. Je možné si vytvořit své vlastní `View` s rozšířenou funkčností, a to pouhým děděním od požadovaného `View`. Konkrétním příkladem a použitím tohoto způsobu tvorby vlastních `View` v této práci je třída `ClearableEditText`, která dědí od třídy `EditText` a rozšiřuje její funkcionalitu o možnost smazání obsahu pole kliknutím na křížek, který se v poli objeví, když pole obsahuje nějaký text.

4.2.3 Styly

Jednotlivým UI komponentám je možné přiřadit styl. Styl je sada atributů a jejich hodnot, které především definují, jak se prvek uživatelského rozhraní

vykreslí. Styly se definují v XML souboru s libovolným názvem uloženým v `res/values`. Konvencí je pojmenovat tento soubor `styles.xml`. Kořenovým elementem souboru musí být element `resources`, který obsahuje jednotlivé styly. Každý styl má své unikátní jméno, přes které se dá v XML styl referencovat zavináčem: `@styles/<my_style>`. Každý styl může mít svého předka, od kterého dědí všechny jeho atributy, jejichž hodnoty lze měnit. Předka je možné uvést v atributu stylu nebo pomocí tečkové notace v názvu stylu. Například styl s názvem `BasicText.Red` je potomkem stylu `BasicText`, je-li definovaný.

Styly lze použít i ke stylování jednotlivých aktivit a dokonce také celé aplikace. V takovém případě je styl aplikován na všechny aktivity a v terminologii Androidu se jedná o `theme` (lze přeložit jako *motiv*). Nicméně zápis motivů je ekvivalentní se zápisem stylů.

4.3 Aplikační prvky

Tato sekce popisuje hlavní aplikační prvky použité v této diplomové práci. Jedná se o záměry, broadcast přijímače a nástroje pro asynchronní zpracování. Do této sekce patří i EMDK (Enterprise Mobile Development Kit), ale jelikož je to relativně rozsáhlé téma, je popsáno samostatně v sekci 4.5.

4.3.1 Intenty a jejich filtry

Záměry, v terminologii Androidu též *intenty* (instance třídy `Intent`), jsou jednoduché objekty s primitivními daty. Intenty jsou nezbytné při provádění nějaké akce nad jinou komponentou, ať už stejné, nebo jiné aplikace. Komponentou se v tomto kontextu rozumí aktivita (*Activity*), služba (*Service*), poskytovatel obsahu (*Content provider*) a broadcast přijímač (*Broadcast receiver*). Nejčastější způsoby použití intentů jsou 3: spuštění aktivity, spuštění služby a doručení broadcast zprávy.

Intenty se dělí na dva typy:

- **explicitní** - *explicitně* specifikují konkrétní jméno komponenty, která se bude spouštět (například jméno aktivity v aplikaci). Používají se ve chvíli, kdy je jméno cílové komponenty známé,
- **implicitní** - nespecifikují konkrétní komponentu, naopak toto rozhodnutí nechávají na systému. Požadovanou akci tedy klidně může provést komponenta z jiné aplikace.

Například zobrazení konkrétní lokace na mapě; vývojář aplikace nemusí kvůli jednomu zobrazení lokace na mapě implementovat celou funkcionalitu ohledně zobrazení mapy a vycentrování hledané pozice. Naopak si vytvoří implicitní intent, do něhož uloží informaci o tom, že chce zobrazit lokaci na mapě, a nechá systém, ať aplikaci vybere. Pokud jich je víc, systém zobrazí dialog s aplikacemi schopnými provést tuto akci. Jak ale systém pozná, že to nějaká aplikace umí? K tomu slouží intent filtery (třída `IntentFilter`).

Intent filtery se uvádějí u komponent (převážně aktivit) a je to způsob, jakým aktivita dává najevo, že umí nějaký implicitní intent provést. V případě aktivit se intent filtery uvádějí v manifestu u příslušné aktivity. Pokud tedy nějaká aktivita obsahuje intent filter, znamená to, že ji lze spustit i implicitním intentem. Pokud žádný nemá, lze ji spustit jen explicitním intentem. Při spuštění implicitního intentu operační systém prohledá manifesty a intent filtery odpovídající danému intentu. Pokud nalezne právě jednu shodu, spustí danou aktivitu, pokud jich nalezne více, zobrazí dialog, ze kterého si uživatel vybere.

Intent může obsahovat jméno komponenty (**component name**), které, je-li specifikované, je intent explicitní. Pokud ne, systém jej bere jako implicitní a podle dalších informací v intentu se rozhoduje. Dalším polem v intentu je akce - **action**. Je to řetězec, který popisuje akci, kterou má intent provést. Provedení akce může vyžadovat nějaké parametry zvané intent **extras** (např. souřadnice pozice, na kterou se má vycentrovat mapa v jiné aktivitě). Extras také slouží jako kontejner pro přenos dat pomocí intentu. Extras se identifikují klíčem, pomocí kterého se do intentu ukládají a následně získávají. Další část intentu se nazývá **data** a jedná se o URI nebo MIME type popisující cílovou akci a typ dat. Poslední částí intentu je jeho kategorie (**category**). Je to, jako v případě akce, textový řetězec popisující kategorii, do které intent patří.

```
1 <intent-filter>
2  <action android:name="android.intent.action.MAIN" />
3  <category android:name="android.intent.category.LAUNCHER" />
4  <category android:name="android.intent.category.DEFAULT" />
5 </intent-filter>
```

Kód 4.1: Ukázka intent filteru spouštěcí (hlavní) aktivity

Ukázka intent filteru 4.1 popisuje intent hlavní aktivity, kde akce `*.MAIN` říká, že daná aktivita je vstupním bodem aplikace, kategorie `*.LAUNCHER` zařazuje intent mezi tzv. launchery, tzn. že aplikace bude viditelná v seznamu

aplikací, a kategorie `*.DEFAULT` říká, že tato aktivita bude použita v případě, když volající intent nebude mít specifikovanou kategorii.

4.3.2 Broadcast přijímače

Intenty úzce souvisí s broadcast přijímači [15]. Broadcast přijímače slouží především k distribuování informace, že nastala nějaká událost. Příkladem může být připojení sluchátek, odpojení nabíječky zařízení, dostupnost internetového připojení a další. Android umožňuje definování vlastních broadcast receiverů. Standardně jsou broadcasty globální (napříč všemi aplikacemi), Android ale umožňuje zasílat vlastní broadcasty jen v rámci procesu aplikace. Výhodou je bezpečnost (data neopustí aplikaci) a efektivita (globální broadcast náročnější na provedení).

Zprávy zasílané broadcastem jsou již známé intenty. Aplikace se zaregistruje k odběru broadcastů buď záznamem v manifestu, nebo programově. V obou případech specifikuje intent filter, kterým si řekne o typy zpráv (intentů), které chce dostávat. Broadcast receiver je vlastní třída, která dědí od abstraktní třídy `BroadcastReceiver`. Ta má jednu abstraktní metodu `onReceive(Context, Intent)`, která je zavolána, byl-li daný broadcast splňující intent filter *někde* odeslán. Druhým parametrem metody je intent s daty broadcastu (například změna úrovně nabití baterie).

4.3.3 Asynchronní úlohy

Aby uživatelské rozhraní zůstalo aktivní a reagovalo na povely uživatele, je třeba provádět delší operace asynchronně. Android k tomu poskytuje několik nástrojů, které se liší podle doby trvání úloh, které se provádějí asynchronně. Nejpoužívanějším řešením je třída `AsyncTask` [15], která se hodí zejména na kratší asynchronní úlohy (řádově několik vteřin). Na déle trvající úlohy jsou vhodnější třídy balíku `java.util.concurrent`.

AsyncTask se vytvoří děděním od abstraktní třídy `AsyncTask` a překrytím metody `doInBackground()`, která bude prováděna na pozadí (z jiného než UI vlákna, ve kterém aplikace běží). AsyncTask poskytuje další metody (všechny prováděné v UI vlákně), které úzce souvisí s prováděním nějaké operace na pozadí, a pokud chtějí být použity, musí se překrýt:

- `onPreExecute()`: zavolána **před** spuštěním operace na pozadí,
- `onProgressUpdate()`: slouží k informování o změně postupu asynchronní operace. Její volání způsobuje metoda `publishProgress()`, která se volá z `doInBackground()`,

- `onPostExecute()`: zavolána **po** dokončení asynchronní operace.

Konkrétním příkladem použití těchto metod může být stahování souboru z internetu. `onPreExecute()` zobrazí dialog s informací, že stahování probíhá, `onProgressUpdate()` může postupně zobrazovat množství dosud stažených dat a `onPostExecute()` nakonec dialog zavře, případě zobrazí informaci o dokončeném stahování.

4.4 Zdroje

Veškeré zdroje používané v aplikaci, ať už layouty, obrázky a zvukové stopy nebo texty, jsou uloženy centrálně ve složce `res`. Ta obsahuje další podsložky specifické pro konkrétní typ zdroje. Pro obrázky, ať už v bitmapové verzi, nebo XML pro vektorovou verzi, je zde složka `drawables`, pro layouty `layout`, pro texty, styly a další složka `values` atd.

Je zbytečné popisovat všechny složky s jejich významem, jsou zde dále vysvětleny zajímavější a důležitější fakta týkající se zdrojů.

4.4.1 Kvalifikátory zdrojů

Kvalifikátory zdrojů slouží k podpoře různých zařízení s různou konfigurací, ať už se jedná o různou velikost displeje nebo o jinou jazykovou lokalizaci. Systém za běhu aplikace vybírá vhodné zdroje právě s ohledem na konfiguraci zařízení, na kterém běží. Složka, která je nějak specifická, má název ve tvaru `<název složky>-<kvalifikátor1>`. Kvalifikátorů může být více, nicméně je nezbytné dodržet jejich pořadí, jinak bude zdroj ignorován. Nejpoužívanější kvalifikátory jsou:

- jazyk a region: např. `en`, `cs`, `cs-rCZ`, kde `r` značí region,
- orientace displeje: `land` (*landscape* mód) pro rozložení na šířku a `port` (*portrait* mód) pro rozložení na výšku,
- hustota displeje: např. `ldpi`, `mdpi`, `hdpi` a další, viz dále,
- verze platformy: např. `v4`, `v7` a další, viz dále.

Po kompilaci aplikace je vygenerována speciální třída s názvem `R`, která všechny zdroje zpřístupňuje. Pro každý typ zdroje obsahuje třída `R` vnitřní třídu s názvem typu zdroje, která obsahuje identifikátory konkrétních zdrojů, např. `R.string.hello_world` nebo `R.drawable.image1`, kde `image1` je současně i jméno obrázku. Fyzická jména zdrojů bez přípony jsou tedy shodná

s názvy svých identifikátorů ve třídě `R.<typ zdroje>`. Tyto identifikátory jsou poté použity v kódu aplikace k přístupu ke zdroji přes třídu `Resources`.

Ke zdrojům lze přistupovat i v XML souborech, např. v layoutech, a to takovouto notací: `@string/hello_world` nebo `@drawable/image1`.

Hustota displeje a jednotky

Různá zařízení často mají displeje o různých velikostech, rozlišeních a tedy i o hustotě pixelů. Aby aplikace vypadala na všech těchto zařízeních stejně a obrázky a ikony byly stejně *kvalitní* a nebyly rozmazané, musí se použít kvalifikátory pro hustotu displeje a v případě návrhu layoutu obrazovky také speciální jednotky.

Nelze uvažovat o známých fyzických pixelech, neboť tlačítko o velikosti 200x300px bude na obrazovce s rozlišením 480x640 zabírat necelou čtvrtinu obrazovky, v případě full-hd displeje by bylo tlačítko znatelně menší. Android z tohoto důvodu zavádí novou jednotku `dp` - *density-independ pixel* neboli virtuální pixel, který nezávisí na hustotě displeje. Použití této jednotky má za následek stejný vzhled uživatelského rozhraní na displejích s různou velikostí a hustotou.

Samotný přepočet na reálnou fyzickou velikost (na fyzické pixely) provádí systém automaticky takto: $px = dp * hustota/160$ s tím, že hustotu lze spočítat jako: $hustota = \sqrt{w^2 + h^2}/d$, kde `w` a `h` je šířka a výška displeje v pixelech a `d` je úhlopříčka v palcích. Podle výsledku systém přiřadí displej do příslušné kategorie (tabulka 4.1) a podle násobícího faktoru přepočítává `dp` na fyzické pixely.

kategorie	**jméno kategorie**	**hustota**	**násobící faktor**
ldpi	low density	120 dpi	0,75
mdpi	medium density	160 dpi	1
hdpi	high density	240 dpi	1,5
xhdpi	extra-high density	320 dpi	2
xxhdpi	extra-extra-high density	480 dpi	3
xxxhdpi	extra-extra-extra-high density	640 dpi	4

Tabulka 4.1: Kategorie displejů podle jejich hustoty

Z tabulky 4.1 a ze vztahu pro výpočet pixelů je zřejmé, že pro displeje spadající do kategorie `mdpi` plaí $1dp = 1px$. Z hustot ostatních kategorií lze odvodit násobící faktor, kterým stačí vynásobit `dp` k získání `px`.

Násobící faktor hraje velmi důležitou roli i při vykreslování bitmap. Například bitmapa, která má být přesně 24x24dp velká. Pokud by byla pouze jedna verze ve složce `drawables` o velikosti např. 24x24px, tedy pro `mdpi` displeje, vypadala by ikona na `xxhdpi` displeji velmi rozmazaně. Nyní se hodí uplatnit kvalifikátor pro hustotu displeje a aplikovat jej na složku `drawables`. Výsledkem budou složky `drawables-ldpi`, `drawables-mdpi`, `drawables-hdpi` atd. Do každé této složky se vloží daná ikona se stejným názvem a s přepočítanou velikostí (za předpokladu, že ji můžeme externě měnit). Aplikováním násobícího faktoru lze vypočítat, že do `drawables-ldpi` vložíme stejnou ikonu o velikosti 18x18px, do `drawables-mdpi` o velikosti 24x24px, a tak dále.

Další virtuální jednotkou jsou `sp` - *scale-independent pixels*. Jedná se o jednotku, která funguje stejně jako `dp`, ale navíc je přepočítávána podle nastavení velikosti písma uživatele. Je tedy doporučeno tuto jednotku používat pro specifikaci velikosti písma.

4.5 Enterprise Mobile Development Kit 4.0

Enterprise Mobile Development Kit 4.0, zkráceně EMDK, je nejnovější verzí EMDK. Jedná se o sadu nástrojů, která umožňuje využívat potenciál mobilních počítačů a zařízení (například skenování čárových kódů) a vytvářet pokročilé podnikové aplikace. EMDK 4.0 je podporováno pro využití ve vývojových prostředích Android Studio a Eclipse s ADT[2] pluginem. V případě Eclipse je EMDK 4.0 poslední podporovanou verzí, dále bude podporováno jen Android Studio. Jelikož EMDK vytvořila společnost Zebra Technologies, je použitelné pouze na zařízeních tohoto výrobce, kde je již standardně předinstalováno.

První část EMDK je nativní knihovna, která je v zařízení již předinstalována, nicméně výrobce poskytuje možnosti a nástroje, jak EMDK aktualizovat, vyjde-li nová verze (tak tomu ostatně bylo i v tomto případě, kdy testovací zařízení původně obsahovalo verzi EMDK 3.1, která byla nahrazena verzí 4.0). Druhou částí je aplikační EMDK, které se instaluje na vývojářský počítač a do vývojového prostředí. Součástí instalace je JAR knihovna, která po přidání do classpath umožňuje ovládat zařízení za pomoci EMDK. Tato knihovna je pouze tzv. *stub* knihovnou, tzn. že její metody neobsahují implementaci a jejich zavolání mimo koncové zařízení vyvolá `RuntimeException`. Proto i při sestavování aplikace musí být ve vývojovém prostředí nastaveno, že EMDK knihovna bude použita při kompilaci, ale nebude součástí

[2]ADT - Android Developer Tools

výsledné aplikace. Podobně to funguje i u Android frameworku (knihovny Android.jar), kdy zavolání jakékoliv její metody mimo reálné zařízení s Androidem způsobí výjimku. Další nezbytnou podmínkou pro používání EMDK v aplikaci je přidání práv v manifestu:

```
1 <uses-permission
2         android:name="com.symbol.emdk.permission.EMDK" />
```

a specifikace, že aplikace bude využívat sdílenou knihovnu:

```
1 <uses-library
2         android:name="com.symbol.emdk"
3         android:required="false" />
```

Tato řádka v manifestu říká systému, že při instalaci musí být knihovna nainstalována v zařízení. Jelikož ale může být aplikace používána i na zařízeních bez EMDK, je nutné přidat atribut `android:requried="false"` (standardně `true`), který zajišťuje, že aplikace bude nainstalována i přes absenci knihovny.

EMDK poskytuje mnoho nástrojů a funkcí, které umožňují využít potenciál daného zařízení. Tato práce se nicméně zabývá pouze jedinou jeho funkcionalitou, a tou je skenování dat, jejich zpracování a dodání cílové aplikaci.

4.5.1 DataWedge aplikace

DataWedge [8] je aplikace, která je součástí EMDK a je opět na zařízeních již předinstalována. Aplikace čte data ze skeneru, zpracovává je a dle nastavení uživatele (aplikace) je předává cílové aplikaci. Nastavení toho, jaké aplikace chtějí odebírat data od jakého skeneru a jakým způsobem, se provádějí nastavuje pomocí tzv. profilů.

4.5.2 DataWedge profily

Profily v aplikaci DataWedge konfigurují to, jak se DataWedge zachová po naskenování dat a jak s naskenovanými daty naloží. Profil se skládá z asociovaných aplikací a jejich konkrétních aktivit a z pluginů rozdělených do 3 kategorií, viz dále. Díky možnosti asociace aplikace s profilem lze definovat různé chování DataWedge pro různé aplikace, protože některé budou chtít dodat data jedním způsobem, jiné druhým atd. Pluginy se dělí na vstupní, výstupní a pluginy zpracovávající data:

- **vstupní pluginy**: pluginy, které komunikují s fyzickými vstupními zařízeními, jako například s vestavěným skenerem nebo s připojenou čtečkou magnetických proužků,
- **pluginy pro zpracování dat**: pluginy, které po obdržení dat nad nimi provedou definované operace a až poté je pomocí výstupních pluginů odešlou cílové aplikaci. Operacemi nad daty může být například připojení prefixu nebo postfixu, připojení klávesy `ENTER` nebo `TAB`, různé pokročilé možnosti zpracování typu doplnění nul do požadovaného počtu před/za data nebo z naskenovaných dat odeslat pouze nějakou jejich část atd.,
- **výstupní pluginy**: pluginy, které se starají o doručení dat cílové aplikaci, a to třemi způsoby:
  - **Keystroke** - způsob dodání dat do aplikace jako stream simulovaných stisků kláves, takže z pohledu aplikace přijdou data tak, jako kdyby byla napsána na klávesnici,
  - **Intent** - způsob dodání dat prostřednictvím intentu, a to buď voláním `startActivity()`, `startService()` nebo broadcastem. V konfiguraci profilu se uvede akce, kategorie a jeden ze zmíněných způsobů doručení,
  - **IP** - způsob dodání dat IP protokolem. V profilu se uvede IP adresa a port cílového zařízení a přenosový protokol (TCP/UDP).

Vytváření a konfiguraci profilů je sice možné provádět přímo v DataWedge, nicméně takový způsob může být značně zdlouhavý a při jakékoliv změně by bylo nutné profil změnit i na všech ostatních zařízeních v podniku. Instalace EMDK na vývojářském počítači obsahuje EMDK plugin, který je také nainstalován do používaného vývojového prostředí (Android Studio a/-nebo Eclipse). Po instalaci a restartování vývojového prostředí je v horním menu nová položka s názvem EMDK, která umožňuje otevřít správce profilů. Správce profilů umožňuje vytvářet a konfigurovat profily stejně jako v DataWedge, nicméně používání je ve srovnání s DataWedge pohodlnější.

Po vytvoření a uložení profilů ve vývojovém prostředí je automaticky vytvořen soubor `EMDKConfig.xml` v adresáři `/assets`. Po prozkoumání tohoto souboru je zřejmé, že se jedná o textový popis vytvořených profilů, který je přibalen do souboru výsledné aplikace. To, že je konfigurační XML součástí aplikace, ještě neznamená, že po jejím nainstalování dojde k automatickému promítnutí profilů do DataWedge. Tato akce, jinak nazývaná aktivace profilů, musí být provedena programově pomocí EMDK knihovny, a to většinou po spuštění aplikace, aby bylo skenování pro aplikaci dostupné.

Třetím způsobem vytváření profilů je čistě programový kód volající API EMDK.

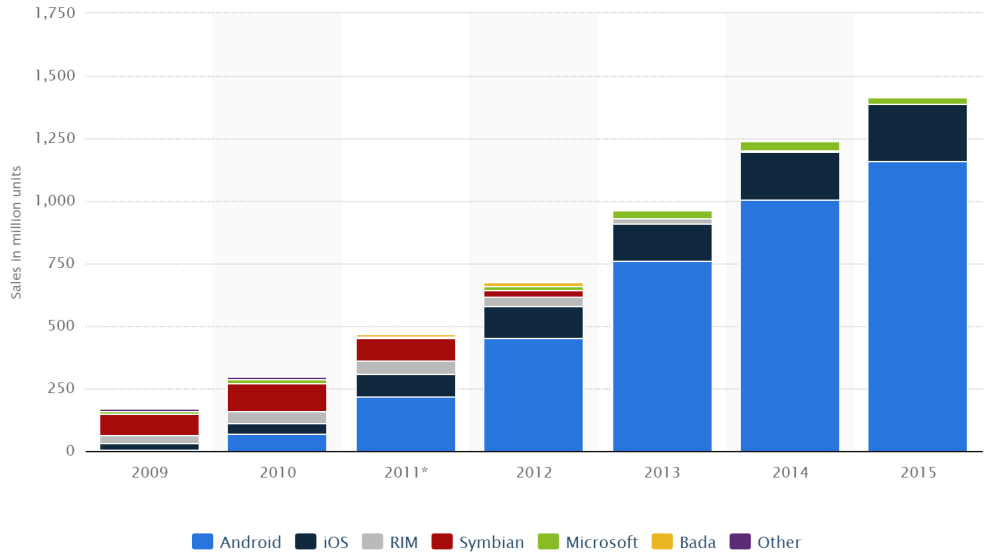
4.6 Konkurenční operační systémy

Android není jediným operačním systémem používaným především na mobilních zařízeních. Jeho největšími konkurenty [22] jsou operační systémy iOS vyvíjený společností Apple a Windows Phone vyvíjený společností Microsoft (srovnání popisuje tabulka 4.2). Dalšími operačními systémy, které bývaly rozšířené, jsou Blackberry OS a Symbian, nicméně uvedeným systémům svou rozšířeností nemohou konkurovat. Ostatní operační systémy jsou používané na tak malém počtu zařízení, že mohou být zanedbány.

	4. kvartál 2015		4. kvartál 2014	
Operační systém	**Počet zařízení**	**Tržní podíl (%)**	**Počet zařízení**	**Tržní podíl (%)**
Android	325 394,4	80,7	279 057,5	76,0
iOS	71 525,9	17,7	74 831,7	20,4
Windows	4 395,0	1,1	10 424,5	2,8
Blackberry	906,9	0,2	1 733,9	0,5
Ostatní	887,3	0,2	1 286,9	0,4
Celkem	**403 109,4**	**100,0**	**367 334,4**	**100,0**

Tabulka 4.2: Srovnání využití mobilních platforem ve 4. kvartálech roků 2014 a 2015 [22] (počty zařízení jsou v tisících)

Graf na obrázku 4.3 potvrzuje fakt, že je Android nejpoužívanějším operačním systémem. Zajímavý je rok 2009, kdy rozšířenost Androidu byla minimální a nejrozšířenějším systémem byl Symbian. Totéž platilo i v roce 2010. Android se stal nejpoužívanějším systémem v roce 2011 a tento stav doposud (2016) trvá.



Obrázek 4.3: Srovnání prodejů zařízení s různými operačními systémy za posledních 7 let (zdroj: *statista.com*)

5 Analýza

Tato kapitola se v první části zabývá studiem a analyzováním různých možností architektury aplikací vhodných pro platformu Android. Další část se týká *dependency injection* a analýzou, jestli je možné principy dependency injection v aplikaci použít. Jsou popsány různé způsoby řešení a jejich silné a slabé stránky. V poslední kapitole analýzy jsou popsány způsoby testování platformy Android, existující testovací frameworky a jejich srovnání.

5.1 Architektura aplikace

Výběr vhodné architektury aplikace je stěžejním bodem před implementací. Není-li aplikace příliš rozsáhlá, určitě šance na implementaci nebo na změny architektury jsou. Pokud se jedná o velký projekt, je změna architektury bez přepsání značného množství kódu prakticky nemožná. Vhodná architektura aplikace zajišťuje přehlednost kódu, usnadňuje její rozšiřitelnost.

Tato sekce popisuje možné a použité architektonické vzory.

5.1.1 MVVM

Model-View-ViewModel je architektura založená na bindování UI prvků s konkrétními objekty modelu. Modelem, jsou stejně jako v případě ostatních architektur, označovány datové třídy a data. View se pak stará o prezentaci dat a interakci s uživatelem. Vrstva View-Model architektury je zodpovědná za stav View. Jejím úkolem je poskytovat data z modelu pro View.

Bindování mezi Modelem a View zprostředkovává bindovací framework. V případě Androidu se nejčastěji jedná o knihovnu AndroidBinding[1]. O jakékoliv změně v modelu je bindovací framework informován a View je automaticky aktualizováno.

5.1.2 MVC

Model-View-Controller je další známou architekturou, kterou lze aplikovat při vývoji pro Android. Skládá se ze třech hlavních komponent. Nejvyšší vrstvou je View, které podobně jako u MVVM zobrazuje data. V kontextu MVC se vrstvu View považuje XML definující vzhled a strukturu uživatelského rozhraní. Model, stejně jako v případě MVVM, poskytuje datové

[1] `https://github.com/gueei/AndroidBinding`

objekty a data, která View zobrazuje. Modelem může být například databáze nebo lokální úložiště. Nejdůležitější částí MVC je Controller. Jeho úkolem je spravovat Model, komunikovat s ním a získávat z něj data. Získaná data potom prostřednictvím View Controller zobrazí. V kontextu Androidu jsou za Controller považovány aktivity a fragmenty, neboť komunikují s nižší vrstvou, ze které získávají data, a ta poté prostřednictvím elementárních UI prvků zobrazují. Controller dále zpracovává události a vstupy od uživatele a příslušně na ně reaguje.

MVC není unikátní architektura a často je spolu s různými platformami vysvětlována odlišně. Základní koncept MVC je každopádně všude stejný a principielně odpovídá předchozímu odstavci. Controller zpracovává události a vstupy. View zobrazuje data a nestará se o to, kde se vzala, a Model poskytuje data a nestará se o to, jak budou zobrazena.

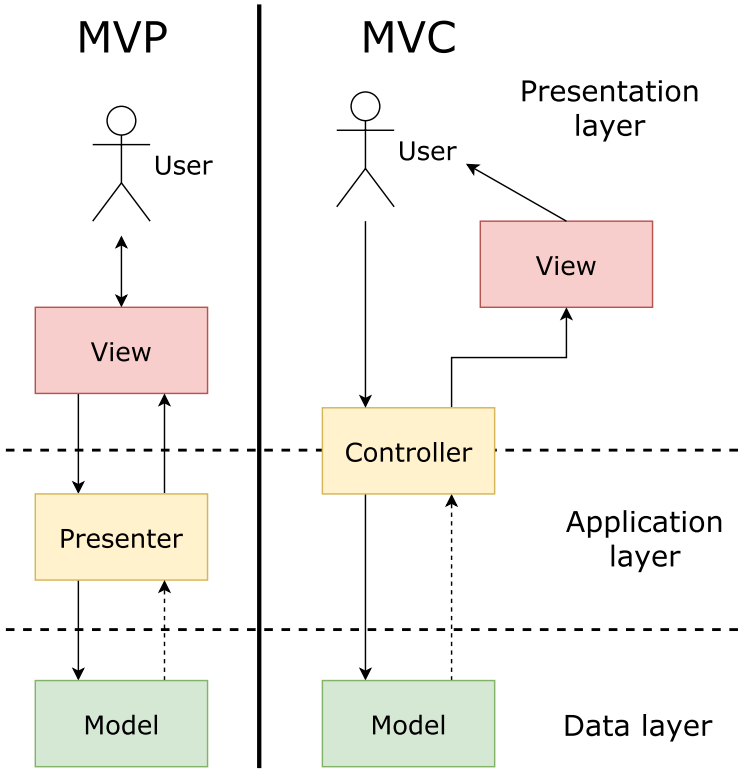
5.1.3 MVP

Cílem Model-View-Presenter architektury je kompletní oddělení View a Modelu pomocí návrhového vzoru Presenter (dále prezentér). Pokud je architektura navržena správně, neví prezentér nic o typu View, se kterým komunikuje. Komunikace probíhá pomocí rozhraní, které prezentér poskytuje a kterým dává najevo, že jakékoliv View, které má být nad ním postaveno, musí danou funkcionalitu poskytovat (implementovat rozhraní). Na konkrétním typu View už prezentéru nezáleží. Správně navržený prezentér by měl od View vyžadovat akce, které nesouvisí s jeho typem. Příkladem nesprávné akce prezentéra může být `zobrazAlertDialogSChybou(<chyba>)`. Taková akce je závislá na konkrétním typu View (Android a jeho `AlertDialog`). Stejná akce nezávislá na typu by se nazývala pouze `zobrazChybu(<chyba>)`, případně vyžaduje-li prezentér dialog, tak `zobrazChybovyDialog(<chyba>)`. Takových situací je ovšem často náročné dosáhnout.

Prezentér tedy slouží jako prostředník mezi View a Modelem. Řídí jejich komunikaci a na základě akcí uživatele říká, co se bude dělat a co se bude zobrazovat. Tím, že řídí akce od uživatele, také zbavuje prezentační vrstvu logiky, která v případě MVC rozhodování prováděla. Díky tomu vypadají aktivity a fragmenty jednodušší a *lehčí*. Za nevýhodu MVP lze považovat to, že implementace MVP vytváří relativně velké množství tříd, které moohu při nesprávné organizaci působit zmatek.

Další výhodou MVP je v případě platformy Android jednodušší testování. V případě MVC by musela být aplikační logika testována instrumentačními testy na reálném zařízení nebo emulátoru. Při použití MVP obsahuje logiku prezentér, který je nezávislý na implementačních detailech View a je

možné jej testovat lokálními JUnit testy, které běží na JVM. Srovnání architektur popisuje obrázek 5.1.



Obrázek 5.1: Srovnání architektur MVC a MVP

5.2 Dependency injection

Dependency injection (dále DI) neboli vkládání závislostí je spíše než architektura konceptem nebo návrhovým vzorem. Cílem DI je strukturovat jednotlivé komponenty (třídy) tak, aby nebyly závislé na jiných. Závislostí se myslí situace, kdy jedna komponenta ve svém kódu vytváří instanci druhé komponenty, a je tedy závislá na její konkrétní implementaci - není možné ji nezávisle testovat. Cílem DI je nechat si všechny závislosti komponenty dodat *zvenku* - komponenta si nesmí závislosti vytvářet, ani se snažit si je nějak získat. Dodání závislostí do komponent má na starosti DI framework. Komponenta definuje závislost jako atribut, který je nastaven externě. Komponenta si nevytváří instanci dané závislosti sama.

Výsledkem DI je větší přehlednost kódu, protože jsou na první pohled zřejmé závislosti dané komponenty. Další výhodou DI je jednoduchá testovatelnost, protože jsou všechny závislosti dodávány externě.

K popisu závislostí se používají anotace definované standardem JSR330[2], například `@Inject` nebo `@Singleton`. Konkrétně `@Inject` slouží k označení závislosti v komponentě, která má být dodána externě. Touto anotací může být označen konstruktor (constructor injection), atribut třídy (field injection) a metoda (method injection).

5.2.1 Dependency injection framework

Dependency injection framework je část aplikace, která je zodpovědná za dosazení závislostí do komponent. Frameworku musí vývojář aplikace specifikovat dvě základní věci: kde framework závislosti vezme a kam je má následně dosadit. Kontejnerem se závislostmi se rozumí komponenta, která vytváří a poskytuje všechny závislosti žádané v aplikaci.

DI frameworků je více a každý je vhodný v různých situacích. Zde jsou popsány frameworky Guice, Dagger 1 a jeho nástupce Dagger 2. Důvodem, proč jsou popsány pouze tyto frameworky, je to, že ostatní frameworky nejsou podle [21] pro vývoj na platformě Android příliš vhodné, a také to, že na základě Guice a Dagger 1 vznikl Dagger 2, který je ostatně i použitým frameworkem v aplikaci.

5.2.2 Guice

Guice je framework pro DI vyvinutý společností Google. Podporuje JSR330 (stejně jako obě verze Daggeru). Pro vkládání závislostí však používá reflexi, což je jeho velkou nevýhodou. Reflexe probíhá za běhu aplikace, což může v rozsáhlé aplikaci způsobit výrazné zpomalení. Tento nedostatek se snaží řešit Dagger 1. Na vývoji Daggeru 1 se podíleli stejní lidé jako v případě Guice [21], kteří věděli, které nedostatky odstranit. Dagger 1 je cílen na prostředí s omezenými zdroji (paměť, výpočetní výkon atd.), kterým je právě například Android.

Na rozdíl od Guice, který provádí konfiguraci a vkládání závislostí za běhu aplikace, Dagger 1 toto řeší statickou analýzou při kompilaci programu, nikoliv při běhu. Při kompilaci aplikace Dagger 1 analyzuje všechny závislosti, které má k dispozici, a všechna místa, do kterých má závislost dosadit. Z těchto informací sestavuje graf a zjišťuje, jestli je dosazení závislosti možné. Pokud nalezne situaci, kdy to možné není, kompilace skončí chybou. Dagger se tímto přístupem snaží nalézt potenciální problémy dříve, než se aplikace spustí. Tento proces však v první verzi Daggeru nebyl ideální a jeho implementace vyžadovala částečné využití reflexe. Statická analýza a tvorba grafu

[2]JSR - Java Specification Request 330

byla pouze částečná a nedokázala odhalit všechny potenciální chyby. Postupem času se při používání Daggeru 1 objevily ještě další problémy, které bylo nutné řešit. V důsledku toho vznikl Dagger 2.

5.2.3 Dagger 2

Dagger 2 (dále označován jen jako Dagger) vznikl eliminací nedostatků jeho předchozí verze. Statická analýza v Daggeru již kompletně analyzuje celou aplikaci a sestavuje kompletní graf závislostí. Stejně jako v případě první verze generuje i nová verze Daggeru zdrojový kód, který slouží ke vkládání závislostí za běhu aplikace. Staticky vygenerovaný kód kompletně eliminuje reflexi, která vkládání prováděla za běhu aplikace. Další jeho výhodou je to, že jej nemusí psát vývojář ručně.

Dagger má tři hlavní části:

- **poskytovatele**: metody označené anotací `@Provides`, které poskytují nějakou konkrétní závislost (pro `@Inject` označená místa). Samy mohou mít další závislosti, které jim jsou při jejich zavolání dodány. Generovaný graf tedy začíná u poskytovatelů, kteří nevyžadují žádné další závislosti. Díky těmto metodám a anotacím Dagger pozná, že odtud může získat požadovanou závislost;
- **moduly**: třída označená anotací `@Module`, která obsahuje poskytovatele - metody označené `@Provides`;
- **komponenty**: rozhraní označená anotací `@Component`, která zpřístupňují závislosti poskytované moduly. Komponenty propojují závislosti s místy, kam mají být dosazeny.

```
1 @Module
2 public class ModuleA {
3     @Provides
4     @Singleton
5     public A provideClassA() {
6         return new A();
7     }
8 }
```

Kód 5.1: Ukázka modulu poskytujícího jednu závislost

Ukázka 5.1 obsahuje modul `ModuleA`, který poskytuje jednu závislost, konkrétně třídu `A`. Poskytovatel má také anotaci `@Singleton`, která značí,

že při dodávání této závislosti bude vždy použita první vytvořená instance a nebude se vytvářet instance jiná. Pokud by byla vyžadována pokaždé nová instance, stačí anotaci smazat.

Poskytování závislostí

Dagger při kompilaci provádí statickou analýzu kódu, ve které prochází všechny moduly a jejich poskytovatele a sestavuje graf závislostí. Touto analýzou Dagger zjišťuje, jestli může závislosti dodat do `@Inject` míst či nikoliv. Pokud ne, kompilace skončí chybou.

```
1 @Singleton
2 @Component(modules = ModuleA.class)
3 public interface CompA {
4     CompB createCompB(ModuleB module);
5     A getA();
6     void inject(MyClass mc); // or activity, service, ...
7 }
```

Kód 5.2: Ukázka komponenty

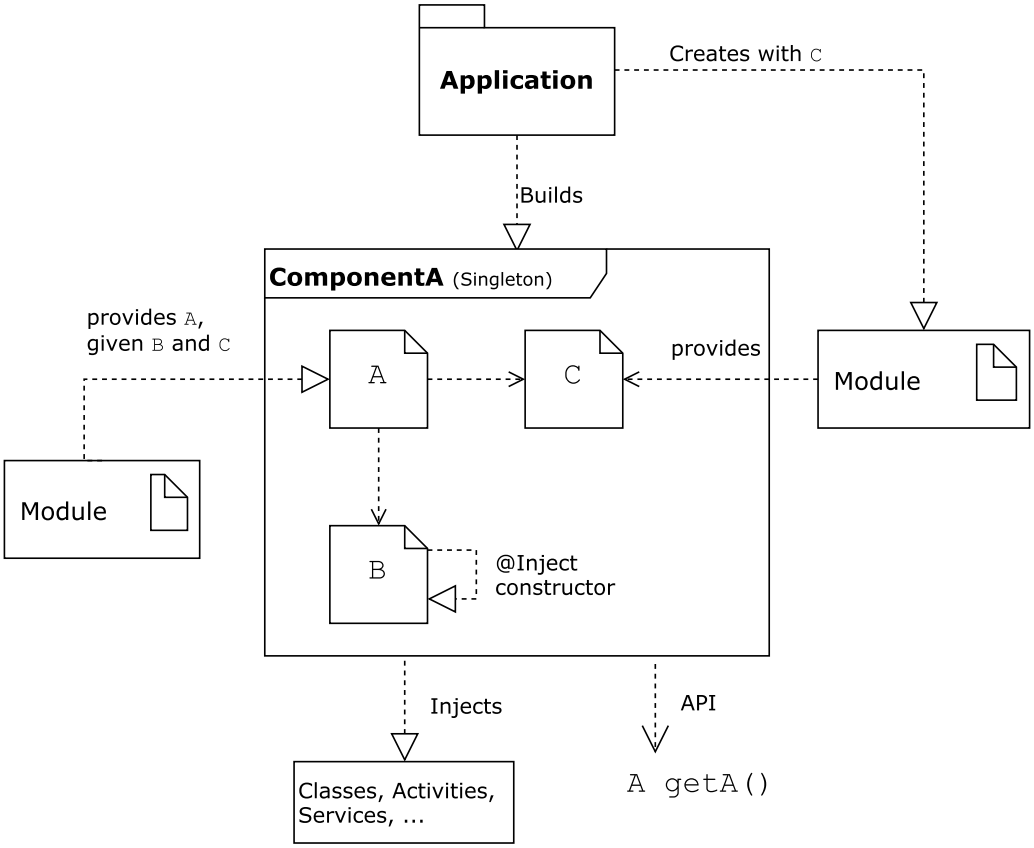
Závislosti jsou poskytovány komponentami. Komponenta je definována jako rozhraní označené anotací `@Component`, jejíž konkrétní implementaci generuje právě Dagger (ukázka 5.2). V anotaci `@Component` je seznam modulů, z nichž komponenta získává závislosti.

Komponenty vkládají závislosti do míst označených anotací `@Inject` (konstruktory, atributy tříd a metody). Injektování závislostí pak probíhá ve stejném pořadí; nejprve konstruktor, poté atributy třídy a až po vytvoření objektu metody.

Anotace `@Inject` u konstruktoru třídy znamená, že parametry konstruktoru jsou závislostmi dané třídy. Takový konstruktor může být právě jeden. Třídy s konstruktorem `@Inject` není nutné uvádět v modulu. Dagger tuto třídu analyzuje a umí ji poskytnout stejně jako v případě poskytovatele v modulu. Důležité je třídu označit anotací `@Singleton`, jinak by Dagger při každém volání závislosti vytvořil novou instanci.

Komponenty mohou své závislosti vystavovat explicitně, jako je tomu v případě ukázky 5.2. Závislost lze potom získat jako `compA.getA()`. Závislosti, které se budou používat na `@Inject` místech, není v komponentě nutné explicitně vystavovat. Nicméně je nutné v komponentě specifikovat třídu, do které se bude vkládání závislostí provádět, a to vytvořením signatury speciální metody. Ta má návratovou hodnotu `void` a jako parametr třídu, do které se budou závislosti vkládat. Injekce samotných závislostí se potom

například v případě `MyClass` provede voláním `compA.inject(myClass1)`, kde `myClass1` je právě instance třídy `MyClass`. Vztah komponent, modulů a poskytovatelů znázorňuje obrázek 5.2



Obrázek 5.2: Diagram komponenty a jejích modulů (zdroj: *github.com/codepath, pozn.: upraveno)*

Diagram na obrázku 5.2 vysvětluje význam komponenty a spolupráci s jejími moduly. Komponenta na obrázku poskytuje celkem 3 závislosti: `A`, `B` a `C`. Závislost `B` na ničem nezávisí a byla dodána Daggerem automaticky, neboť obsahuje `@Inject` konstruktor. Závislost `C` je dodána do svého modulu při jeho vytváření. Modul poté jednoduše instanci předá dál jako poskytovatel. Závislost `A` závisí na `B` a `C`. Jelikož ty na ničem už dále nezávisí, může být A dodáno taktéž. `A` je navíc komponentou explicitně vystaveno.

Takovým způsobem staví Dagger graf závislostí, které pak komponenty poskytují.

Závislost komponent

Samy komponenty mohou být samy na sobě závislé. Závislost komponent se využívá v situaci, kdy životní cyklus závislé komponenty je kratší než

té, na které závisí. Druhým důležitým důvodem je fakt, že držet v paměti všechny komponenty (a instance všech jejich závislostí, které poskytují) není efektivní, když zrovna nejsou potřeba. Závislé komponenty tedy šetří paměť.

Mějme například hlavní komponentu aplikace `CompA` a na ní závislou `CompB`. Konkrétní závislosti poskytované komponentou `CompA` jsou (většinou) jedináčci a jejich životnost je stejná jako životnost aplikace (existují tedy po celou dobu běhu). Závislá komponenta (`CompB`) je taková, která existuje kratší dobu (a její závislosti také). Považujme komponentu `CompB` za komponentu pro nějakou aktivitu. `CompB` pak bude existovat jen tehdy, je-li aktivita otevřená. Na základě způsobu deklarace závislosti může `CompB` využívat závislosti poskytované `CompA`.

Deklarovat závislost mezi 2 komponentami je možné 2 způsoby:

1. explicitním definováním rodičovské komponenty (té, na které závisí),
2. pomocí subkomponent.

V prvním případě (obr. 5.3) se rodičovská komponenta uvádí v poli `dependencies` anotace `@Component` podobně jako moduly, a to následovně:

```
1 @ActivityScope
2 @Component(dependencies = CompA.class, modules =
    ModuleB.class)
3 public interface CompB {
4     void inject(Activity activity);
5 }
```

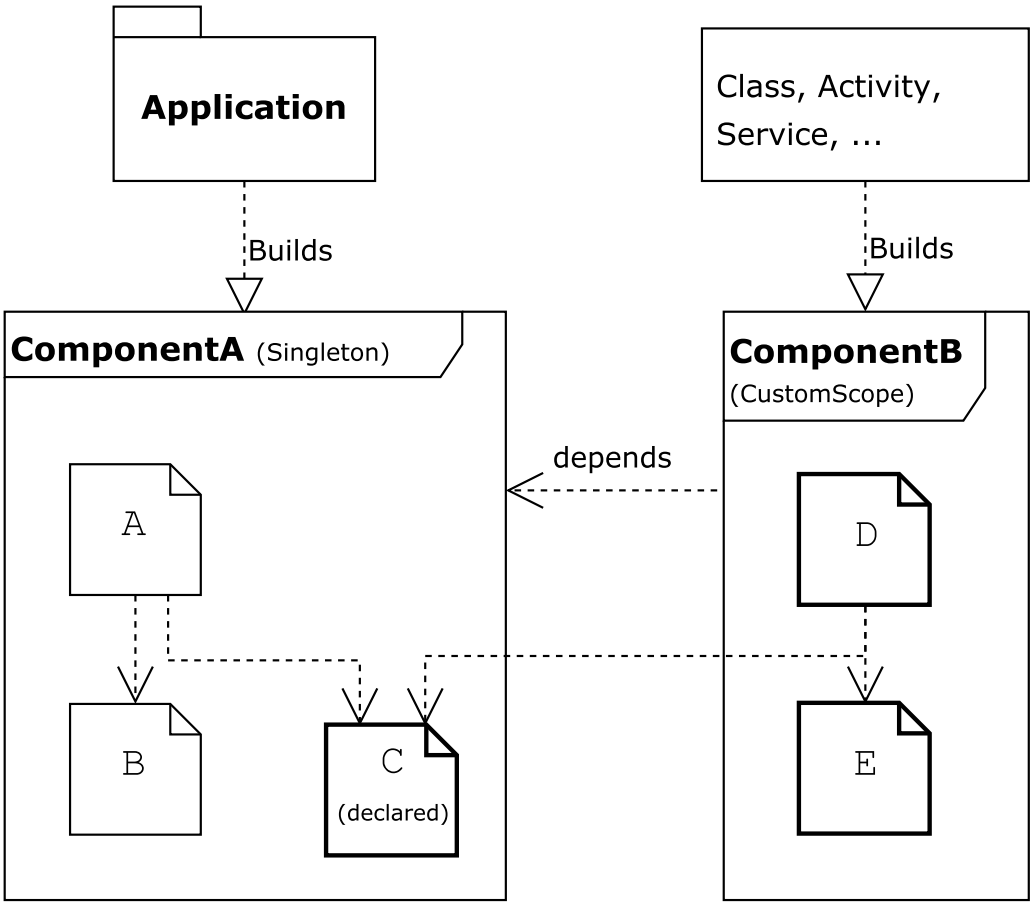
Ve druhém případě (obr. 5.4) se místo anotace `@Component` použije anotace `@Subcomponent`. Komponenta pak vypadá následovně:

```
1 @ActivityScope
2 @Subcomponent(modules = ModuleB.class)
3 public interface CompB {
4     void inject(Activity activity);
5 }
```

Zde je podmínkou deklarovat závislou subkomponentu v rodičovské komponentě, což lze vidět v ukázce 5.2.

Rozdíly mezi těmito 2 způsoby deklarace jsou následující:

- v případě subkomponent musí být subkomponenta v rodičovské komponentě explicitně specifikována,



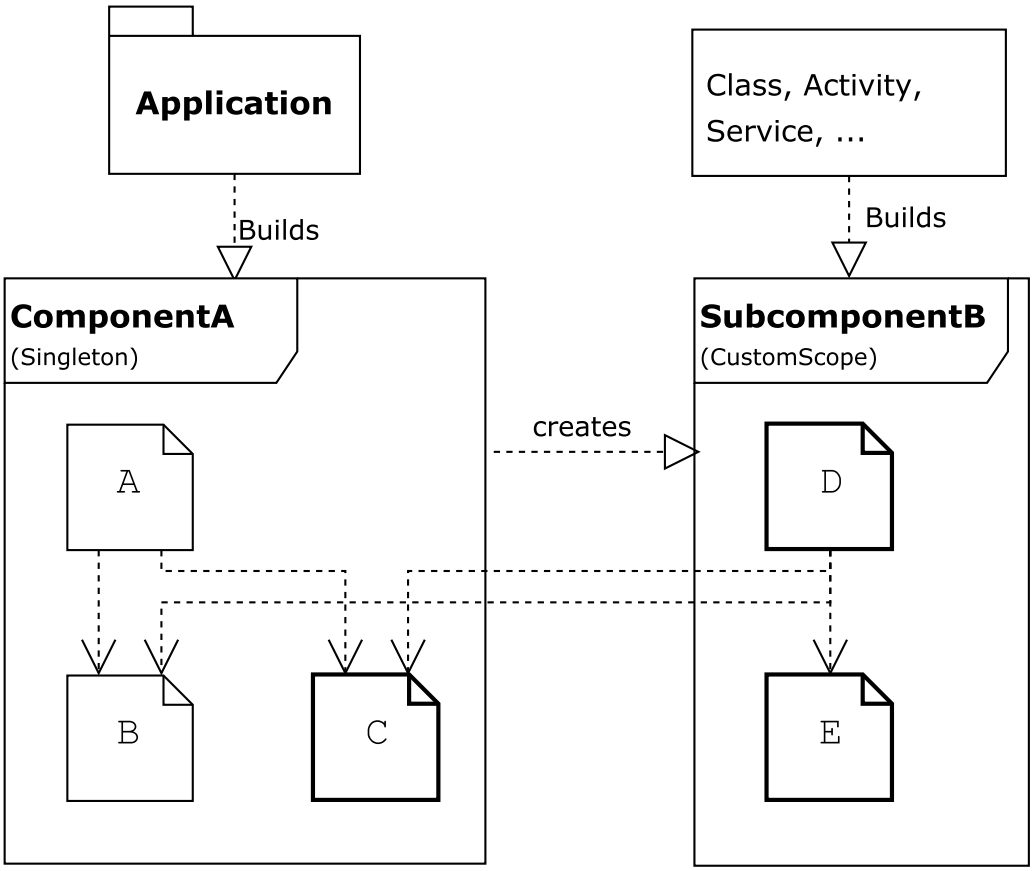
Obrázek 5.3: Závislost komponent pomocí anotací (zdroj: *github.com/codepath, pozn.: upraveno)*

- subkomponenty mají přístup do celého grafu rodičovské komponenty, což v případě prvního způsobu není možné (viz 5.4, kde `D` má přístup k rodičovským `B` a `C`). U anotačního způsobu musí rodičovská komponenta explicitně specifikovat závislosti, které může závisející komponenta využívat (viz 5.3 a explicitně vystavené `C`).

V obou případech každopádně platí, že scope (rozsah/oblast platnosti) musí být jiný, než scope rodičovské komponenty. Dále platí, že závislé komponenty mohou využívat závislosti poskytované komponentami **pouze** na vyšších úrovních. Komponenta nemá přístup k závislostem komponenty na stejné nebo nižší úrovni. Taková závislost se musí přesunout o úroveň výš.

Scopy

Princip scopů v Daggeru určuje počet instancí objektu a souvisí s rozsahem jeho (jejich) platnosti. Jakmile má jakýkoliv poskytovatel (`@Provides`) defi-



Obrázek 5.4: Závislost komponent pomocí subkomponent (zdroj: *github.com/codepath, pozn.: upraveno)*

novaný nějaký scope, Dagger vždy bude udržovat jedinou instanci v daném scopu. Nezáleží na tom, jak se scope nazývá. Název slouží spíše jako informace pro vývojáře. JSR330 standardně definuje jediný scope, a to `Singleton`. Další scopy se vytvářejí ručně, viz následující ukázka 5.3.

```
1 @Scope
2 @Retention(RetentionPolicy.RUNTIME)
3 public @interface UserScope { }
```

Kód 5.3: Definice vlastního scopu

Použití

Jsou-li komponenty, moduly a všechny potřebné závislosti správně definovány a kompilace proběhla úspěšně, zbývá už pouze vytvořit instance jednotlivých komponent a modulů.

Jak již bylo řečeno dříve, Dagger při kompilaci staví graf závislostí a generuje kód, který se později použije k vložení závislostí. Dagger vytváří konkrétní implementace komponent a poskytuje nástroje, jak jejich instance vytvořit získat. Tyto třídy jsou odlišeny prefixem `Dagger`, např. `DaggerCompA`.

Způsob vytváření komponent záleží na použitém způsobu definování závislostí mezi komponentami; buď pomocí parametru anotace `dependencies`, nebo pomocí subkomponent.

V případě definování závislostí v anotaci je postup následující; k vytvoření jakékoliv komponenty je nejprve potřeba vytvořit komponenty, na kterých závisí. Je-li naše komponenta `CompB` závislá na `CompA`, musí se nejprve vytvořit instance `CompA`. K tomu se použije Daggerem vygenerovaná třída `DaggerCompA` takto:

```
1 CompA componentA = DaggerCompA.builder()
2                // instance of class ModuleA.class
3                .moduleA(<instance ModuleA>).build();
```

Při vytváření instance komponenty je nutno předat instance modulů té komponenty, jinak Dagger vrátí chybu. Je-li vytvořena instance rodičovské komponenty, může vytvořit instance závislé komponenty `CompB`:

```
1 CompB componentB = DaggerCompB.builder()
2                .compA(<CompA instance>)
3                .moduleB(<ModuleB instance>).build();
```

Z ukázky je zřejmá závislost `CompB` na `CompA`, proto se při vytváření `CompB` musí použít instance na `CompA`, jinak Dagger vrátí chybu.

Jsou-li instance vytvořené, dají se nad nimi volat metody, které injektují závislosti do specifikovaných tříd, např. `compA.inject(myClass)`, nebo přímo získávat explicitně vystavené závislosti, např. `A instanceA = componentA.getA()` (viz kód 5.2).

V případě použití subkomponent je situace mírně odlišná, stejně ovšem platí, že `CompB` nelze získat bez `CompA`. `CompA` se vytvoří stejně, jak bylo uvedeno výše. K vytvoření `CompB` zde není žádná třída od Daggeru, jako v předchozím případě. Na místo toho byla v `CompA` definována metoda `CompB createCompB(<moduleB>)`, která dělá přesně totéž:

```
1 CompB componentB = componentA.createCompB(<ModuleB instance>);
```

5.3 Kiosk mód

Kiosk mód, mód kiosek nebo jen kiosek je režim, který zajišťuje, že aplikace běžící v tomto režimu bude neustále na popředí. Aplikaci není možné zavřít, vypnout, minimalizovat nebo ji jakkoliv převést do pozadí. Kiosk mód byl spíše výzkumným úkolem než požadavkem, protože Android takovou funkcionalitu pro API level 16 nemá.

Pro zajištění kiosk módu musí být zajištěno následující:

- zakázání tlačítek BACK, HOME, a případně i POWER a tlačítek hlasitosti,
- zakázání notifikační lišty.

Zakázání tlačítka BACK je nejjednodušší. V aktivitě se přepíše metoda `onBackPressed()`, jejíž tělo bude prázdné. Tím se zastaví propagace události stisku a tlačítko bude ignorováno.

Zakázání notifikační lišty je také možné a funguje tak, že se přes nofikační lištu vykreslí speciální view, které *pohltí* veškeré události doteku do této oblasti a nepropustí je níže k notifikační liště, která se díky tomu nerozbalí.

Největší komplikace jsou s tlačítkem HOME. Účelem tlačítka HOME je umožnit uživateli současnou aplikaci opustit, a proto nejsou možná žádná řešení typu někde překrýt metodu nebo nepropagovat událost. První možné řešení je použití knihovny `Android-HomeKey-Locker`[3], která zakazuje HOME tlačítko (dokonce i tlačítka BACK a MENU). Knihovna funguje tak, že zobrazí dialog o nulových rozměrech, který je typu „systémová chyba“. V takové situaci je tlačítko HOME nefunkční. Problémem je to, že toto řešení nefunguje na virtuální tlačítko HOME v dolní liště displeje, které zůstává stále aktivní. A právě testovací podnikové zařízení takovýto typ HOME tlačítka má.

Druhý způsob zakázání tlačítka HOME se od ostatních způsobů liší tím, že tlačítko vyloženě nezakazuje. Naopak, místo zakázání stisku tlačítka se pracuje s událostí a reakcí na jeho stisk. Řešení spočívá v nastavení aplikace jako `launcher`. Launcher je domovská aplikace, která se zobrazí po stisku tlačítka HOME. Jakmile je nějaká aktivita nastavena jako HOME aktivita a uživatel stiskne HOME, tak systém tyto aktivity (jejich aplikace) zobrazí jako výběr, která se má použít. Po zvolení požadované aplikace včetně zapamatování volby dojde vždy při stisku HOME ke spuštění HOME aktivity vybrané aplikace. HOME aktivita se označuje kategorií HOME v manifestu v intent filtru aktivity. Další problém je, že stisk HOME standardně vytváří

[3] `https://github.com/shaobin0604/Android-HomeKey-Locker`

novou instanci nastavené HOME aktivity. To lze změnit nastavením atributu `launchMode` v manifestu na `singleTask`, což znamená, že pokud ještě žádná instance aktivity neexistuje, vytvoří se nová a v novém tasku. Pokud nějaká existuje, je jí předán spouštěcí intent a nová instance se nevytvoří. Bohužel tomu tak není. Systém vytvoří instanci novou a v tu chvíli jsou dvě instance jedné aktivity. Toto chování je známou a dosud nevyřešenou chybou v Androidu[4].

5.4 Logování

Logování tak, jak je známé, není na platformě Android zrovna rozšířené. Vyvíjené aplikace jsou využívány koncovými zákazníky na koncových zařízeních, ke kterým nemá vývojář přístup, a nemůže tak využít potenciál logů. Možným řešením je logování prostřednictvím internetového připojení, kdy se logy odesílají k vývojářům. Nevýhodou je velký přenos dat, což je věc, kterou se snaží vývojáři neustále minimalizovat. Další věcí je, že ne všechna zařízení mohou být připojena k internetu.

V případě DCIx nejsou ovšem podmínky tak svazující. Používaná zařízení se pohybují především ve skladových halách a jsou připojená k podnikové síti. Zařízení jsou tedy i fyzicky dostupná.

K Androidu existují logovací knihovny, které umožňují logování do souboru. Android interně obsahuje `logcat`, což je logovací systém, který umožňuje logovat záznamy z aplikace a následně je procházet. Logcat sbírá logy do interního bufferu zařízení, jehož velikost je omezená. Přístup k logům je možný pouze s připojeným zařízením k počítači, ve kterém jsou nainstalované vývojářské nástroje. Logcat také neumožňuje detailnější konfiguraci typu zápis do souboru, rolování souborů nebo maximální velikost souboru. K tomu slouží logovací frameworky, z nichž ty hlavní mají podporu i pro Android.

Významné logovací frameworky, které lze použít k logování na Androidu:

- **log4j**: logovací framework pro aplikace napsané v Javě,
- **logback**: nástupce log4j, který řeší jeho největší nedostatky,
- **SLF4J**: nejedná se vyloženě o logovací framework. SLF4J funguje jako abstrakce nad nějakým konkrétním logovacím frameworkem. Použití SFL4J jako fasády umožňuje před spuštěním aplikace výměnu logovacího frameworku bez zásahu do kódu. To platí i pro log4j a logback, neboť oba dva frameworky podporují bindování na SLF4J.

[4] `https://code.google.com/p/android/issues/detail?id=26658`

Jak log4j, tak logback umožňují svou konfiguraci pomocí XML souboru. Log4j zároveň také tzv. property souborem a logback ještě navíc pomocí Groovy[5]. Konfigurací se rozumí nastavení chování frameworku při vytvoření záznamu logu. Konfigurace těchto frameworků jsou odlišné a v případě jejich změny je potřeba změnit i jejich konfiguraci.

Použití konfiguračního souboru u frameworku log4j není možné, protože parsování konfiguračního souboru typu properties využívá třídy z balíku `java.beans`, které Android framework neobsahuje. Podobná situace je u XML souboru, kdy opět frameworku chybí třídy vytvářející DOM model. Jediným řešením, jak log4j konfigurovat, je programově.

Na rozdíl od log4j umožňuje logback konfiguraci pomocí XML, a to vytvořením konfiguračního souboru `assets/logback.xml`. Tato konfigurace je po spuštění aplikace automaticky načtena. Nevýhodou je nekompatibilita konfiguračního XML s XML pro log4j.

5.5 Testování

Tato sekce převážně čerpá z `developer.android.com/training/testing`, protože je tento zdroj cílen na testování platformy Android.

Testování zdrojového kódu je kritickou součástí vývoje aplikací a softwarových komponent. Testování pomáhá k dosažení vysoké kvality kódu, odhalení chyb a k ověření, že jednotlivé části kódu vykonávají svou činnost a že ji vykonávají správně. Dalším důvodem testování je fixování funkcionality pro případná pozdější rozšíření. Testování je nutné provést ještě před tím, než se výsledný kód dostane do reálného prostředí, kde by v případě důkladného neotestování mohl mít různé nepříjemné následky, ať už málo kritické nebo fatální.

5.5.1 Základní rozdělení

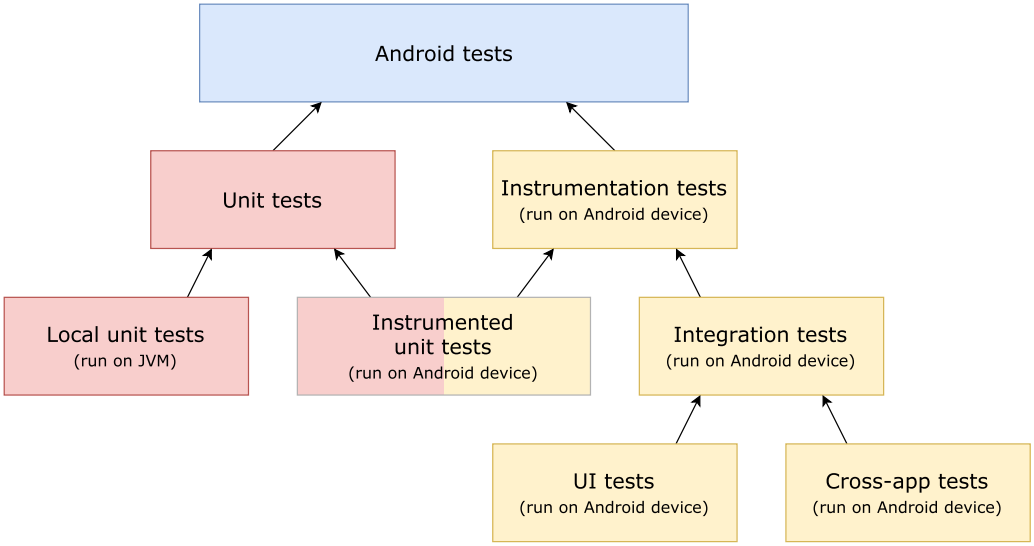
Testy na platformě Android se dělí na dva základní typy (viz obr. 5.5):

- **jednotkové (unit) testy**: testy, které se dále dělí na lokální a instrumentační (viz dále). Tyto testy jsou vhodné především pro testování kódu, který nevyžaduje Android framework. Lze je tedy spouštět na JVM[6]. Příkladem takového kódu je aplikační logika. Ta by neměla mít přístup k Android frameworku a mělo by být možné ji testovat nezávisle;

[5] `http://www.groovy-lang.org/`

[6] JVM - Java Virtual Machine

- **instrumentační testy**: testy, které na rozdíl od lokálních unit testů vyžadují Android framework, a proto je lze spustit jen na zařízeních s Androidem. Unit testy mohou být také instrumentační; jsou to takové testy, které vyžadují Android, ale nijak neinteragují s uživatelským rozhraním (viz `Parcelable`). Další částí instrumentačních testů je testování uživatelského rozhraní aplikace (tlačítek atd.) a testování vzájemné interakce celých aplikací (cross-app testing).



Obrázek 5.5: Základní rozdělení testů platformy Android

Z povahy instrumentačních testů vyplývá, že každé jejich spuštění vyžaduje deploy (nasazení) na zařízení s Androidem, což je relativně pomalý proces. Snahou tedy je, aby bylo co možná nejvíce testovacích případů pokryto lokálními unit testy. Ty běží na JVM, nevyžadují deploy a jejich spuštění je velmi rychlé. Build systém Androidu, Gradle, poskytuje při vývoji speciální verzi Androidu frameworku. Jedná se o knihovnu `android.jar` se všemi třídami a metodami vybraného API levelu. Nicméně tato knihovna obsahuje pouze deklarace hlaviček tříd a metod bez jejich konkrétní implementace, která je až na koncových zařízeních. Jakékoliv volání Android frameworku z `android.jar` **mimo** zařízení s Androidem vyhodí výjimku `RuntimeException` (viz `stub` knihovna v 4.5). Tuto komplikaci lze částečně obejít pomocí tzv. `mock` objektů a mockovacích frameworků, více v sekci 5.5.3.

5.5.2 Instrumentation API

Na způsobu spuštění a běhu unit testů není nic složitého, testy se pouze spustí na lokálním JVM. V případě instrumentačních testů je situace složitější. Ke spuštění instrumentačních testů se využívá tzv. instrumentace a instrumentačního API. Instrumentace je množina speciálních metod v Androidu, které umožňují nezávislé ovládání komponent. Komponentami jsou v tomto kontextu myšleny aktivity, služby, broadcast přijímače a poskytovatelé obsahu.

Standardní průběh vytváření např. aktivity řídí systém tím, že postupně volá metody životního cyklu aktivity. Tyto metody není možné ve zdrojovém kódu volat manuálně. Od toho je zde instrumentace, která poskytuje API, které umožňuje ovládat životní cyklus aktivit. Vytvořením instrumentačních testů se vytvoří vlastně druhá verze aplikace, respektive verze, která obsahuje testy a bude testovat hlavní aplikaci.

Obě aplikace jsou potom nainstalovány a spuštěny ve stejném procesu. Díky společnému procesu je jedné aplikaci umožněno ovládat komponenty druhé. Ke spouštění testů z testovací aplikace slouží speciální třída, tzv. test runner (např. `AndroidJUnitRunner`). Jeho úkolem je spouštět testy a zpracovávat a podávat informace o jejich výsledcích.

Synchronizace

Instrumentace sama o sobě běží v jiném vlákně, než je UI vlákno aplikace. Z toho vyplývá problém synchronizace, kdy instrumentační vlákno může být rychlejší než UI vlákno. Aplikace by potom nestíhala reagovat a instrumentační testy by mohly selhávat. Synchronizaci jde samozřejmě řešit. Prvním řešením je přidání příkazů pro uspání testovacího vlákna, čímž se zajistí, že UI vlákno aplikace bude mít dostatek času na provedení požadovaných akcí. Dalším řešením je prosté opakování akce, která selhala, ať už konkrétní akce nebo celého testu. Obě řešení jsou ovšem značně neefektivní. Řešením je použít nějaký framework, který synchronizaci zajišťuje sám, například Espresso (viz dále v 5.5.4).

Organizace testů

Testy dle svého typu je vhodné v projektu nějakým způsobem organizovat. K organizaci unit a instrumentačních testů je definovaný standard, který Android Studio podporuje a při vytváření nového projektu pomůže s konfigurací:

- `app/src/main/java`: adresář pro zdrojové kódy aplikace, žádné testy,

- **app/src/test/java**: adresář pro jednotkové testy, které budou spouštěny na JVM,
- **app/src/androidTest/java**: adresář pro instrumentační testy, které budou spouštěny na koncovém zařízení s Androidem.

5.5.3 Mock objekty a frameworky

Mock objekty nebo mockovací objekty jsou speciální instance tříd, které simulují chování reálných objektů dané třídy. Vstupem k vytvoření mock objektu je předpis třídy (`class`), výstupem je pak instance tohoto objektu. Mock objekty nemají ve výchozím stavu žádnou funkčnost. Tu lze specifikovat explicitně, což je právě jejich největší výhodou a hlavním účelem.

Mock objekty se používají v případech, kdy je potřeba nahradit nějakou závislost. Této závislosti lze pak explicitně nastavit, jak se bude při nějaké události chovat. Takové vlastnosti mockovacích objektů se přímo hodí v situaci s Android frameworkem, který je poskytován k vývoji jen ve `stub` verzi (hlavičky, žádná implementace). Mockovací objekt by se použil v lokálních unit testech k nahrazení závislostí na Android frameworku (knihovna `android.jar`).

Mockito

Mockito je významným mockovacím frameworkem a je doporučováno přímo stránkami `developer.google.com`. Mockito poskytuje nástroje pro specifikaci chování mock objektů, výjimek, návratových hodnot atd. Ukázka použití:

```
1 List mockedList = mock(List.class); // create mock object
2 mockedList.clear(); // call clear() method on mock object
3 ...
4 verify(mockedList).clear(); // verify that clear() was called
```

Kód 5.4: Ukázka použití mockovacího objektu

Další mockovací frameworky jsou např. JMockit, jMock nebo EasyMock.

5.5.4 Android Testing Support Library

Testing Support Library (TSL) je knihovna poskytovaná společností Google poskytující pokročilé nástroje k testování Android aplikací. TSL má podporu Android Studia, které z něj umožňuje spouštět všechny druhy testů. TSL zahrnuje 3 hlavní části:

- **`AndroidJUnitRunner`**: test runner, který spouští unit testy. Runner podporuje verze unit frameworků JUnit 3 a JUnit 4. Doporučuje se ale, aby byl testovací scénář popsán pouze jedním z nich. Runner se stará o spouštění testů a o zpracování a poskytnutí výsledků testů. Jeho předchůdce, `InstrumentationTestRunner`, podporuje pouze JUnit 3, a proto byl nahrazen tímto runnerem.

- **Espresso**: framework pro testovaní uživatelského rozhraní,

- **UI Automator**: framework také pro testování uživatelského rozhraní, ale vhodný spíše pro testování komunikace a spolupráce více aplikací. Espresso slouží k testování jedné aplikace a jejího uživatelského rozhraní.

Espresso

Espresso je testovací framework uživatelského rozhraní (UI). Je součástí TSL a díky své jednoduchosti je i jedním z nejrozšířenějších testovacích frameworků pro UI.

Hlavní předností Espressa je podpora synchronizace. Problém synchronizace nastává v situaci, kdy je vlákno testovací aplikace vykonáváno rychleji, než UI vlákno testované aplikace. Espresso zbavuje testery nutnosti vkládat do testovacího kódu pozastavovací příkazy. To přináší přehlednost a vyšší rychlost provádění testů.

Další výhodou Espressa je velmi snadná rozšiřitelnost. Pokud nejsou standardně nabízené akce dostačující, Espresso umožňuje velmi jednoduše definovat vlastní. Základní volání Espressa je následující:

```
1 onView(ViewMatcher) // perform where...
2   .perform(ViewAction) // perform what...
3   .check(ViewAssertion); // check that...
```

Význam argumentů:

- **`ViewMatcher`**: třída, která na základě kritérií vybírá view, kterého se bude akce týkat,

- **`ViewAction`**: akce, která se na vybraném view provede,

- **`ViewAssertion`**: tvrzení o view, které se testuje, zda platí (podobně jako `assert*` u JUnit).

Uvedený způsob funguje pro konkrétní view. Pokud ovšem hledané view, na kterém se bude provádět nějaká akce, je v nějakém `AdapterView` (např.

`ListView`), je vhodné místo `onView(Matcher)` použít `onData(Matcher)`. Důvod je ten, že hledané view nemusí být zrovna zobrazeno na displeji a tedy ani v prohledávané hierarchii všech view. `onData(Matcher)` na místo prohledávání view prohledává data daného `AdapterView`, a jakmile najde shodu, použije to view, které data zobrazuje.

Následující ukázka popisuje, jak pomocí Espressa kliknout na tlačítko a ověřit, zda se něco stalo:

```
1 // we know the button ID
2 onView(withId(R.id.my_button)).perform(click());
3 /* OR */
4 // we know the text the button has
5 onView(withText("Login")).perform(click());
6
7 // we know that after click another EditText should appear
8 onView(withId(R.id.my_edittext)).check(matches(isDisplayed()));
```

5.5.5 Ostatní testovací frameworky

K výše uvedeným frameworkům ještě patří i další testovací frameworky, jako například Robotium, Robolectric nebo různé cross-platformní frameworky jako Appium nebo Calabash - ty už ovšem nejsou specifické pro Android a mají úplně odlišnou syntaxi zápisu.

Robotium je předchůdce Espressa, je to tedy framework pro instrumentační testy. Jeho hlavním nedostatkem je synchronizace. Autor testů musel synchronizaci nějakým způsobem řešit sám. Další věcí je nepřehledná syntaxe zápisu Robotia ve srovnání s Espressem.

Robolectric

Naopak Robolectric s instrumentací vůbec nesouvisí. Robolectric umožňuje testovat kód lokálně na JVM i přesto, že obsahuje reference z Android frameworku. Tento problém řeší tak, že se snaží nahradit reálné SDK. Robolectric přepisuje implementaci `android.jar` tak, že jakékoliv volání do `android.jar` je přesměrováno na tzv. `shadow` objekty od Robolectric. Ty obsahují základní implementaci velmi podobnou reálné implementaci. Výhodou je tedy to, že se testy nemusí nahrávat a instalovat na zařízení. To zcela jistě šetří čas. I přesto má ale Robolectric své nevýhody. Ačkoliv se snaží pokrýt veškerou funkčnost Android frameworku, nikdy nepokryje všechno, jako například polohové služby, kamery zařízení a další věci, které může nabídnout pouze reálné zařízení.

6 Implementace

Tato kapitola popisuje implementaci aplikace a vše, co se implementace týká. Kapitola vysvětluje postupy a způsoby, které byly použity k vytvoření cílové aplikace. Nejprve je popsán návrh uživatelského rozhraní, který zohledňuje požadavky zadavatele a splňuje jednoduchou rozšiřitelnost. Dále je popsána a vysvětlena architektura aplikace a implementace jejích hlavních komponent, převážně aktivit. Je vysvětlen způsob jejich komunikace, ať už mezi sebou nebo s DCIx (včetně řízení chyb), způsoby jejich vytváření, zanikání a prezentace uživateli.

Dále je vysvětlen způsob spouštění a průchodu transakcí včetně vysvětlení podpory pro skenování čárových kódů. S tím souvisí implementace skenovacího bufferu, který je použit v situaci, kdy zrovna probíhá výměna transakčních obrazovek a data není kam vložit.

V poslední části kapitoly je vysvětlen způsob logování a použití logovacího frameworku.

6.1 Cílová aplikace

Tato sekce popisuje implementovanou aplikaci, vývojové prostředí, způsob sestavení a také testovací prostředí, jak byla aplikace testována.

6.1.1 Popis

Aplikace je cílená na zařízení s minimálním revizním číslem 16 (Android Jelly Bean). Důvodem tohoto rozhodnutí je stále ještě velký počet zařízení[1] s touto verzí Androidu.

Každá aplikace je v rámci OS identifikována jménem balíku uvedeným v manifestu aplikace. Toto jméno musí být unikátní a z toho důvodu je doporučeno použít stejnou konvenci zápisu jako u jmen klasických balíků v Javě, začínající doménovým jménem autora. V případě této aplikace je jméno balíku aplikace nastaveno na `cz.aimtec.dci.android`.

V manifestu jsou dále uvedena práva, která aplikace potřebuje. Konkrétně se jedná o práva umožňující zjišťovat a měnit stav Wi-Fi adaptéru, přistupovat na internet, zapisovat do externího úložiště a používat EMDK knihovnu.

[1] Statistiky verzí Androidu: `http://developer.android.com/about/dashboards/index.html`

Každá aktivita v manifestu má nastavený atribut `screenOrientation` na hodnotu `portrait`, což znamená, že aktivita bude vždy spuštěna v režimu na výšku. Toto rozhodnutí přišlo od zadavatele práce z toho důvodu, že umožnění otáčení není pro aplikaci nutným požadavkem a jeho implementace je složitější.

6.1.2 Vývojové prostředí

Aplikace byla vyvíjena ve vývojovém prostředí Android Studio (dále AS) postaveném na platformě IntelliJ IDEA[2]. Původně bylo používáno ve verzi 1.5.1, nakonec došlo k aktualizaci na verzi 2.0, která podporuje mnoho nových funkcí (například funkce okamžitého spuštění, kdy provedené změny jsou okamžitě aplikovány do aplikace běžící v zařízení). AS je oficiálním vývojovým prostředím aplikací pro platformu Android. Původním vývojovým prostředím bylo Eclipse spolu s ADT pluginem a sestavovacím systémem Ant, nicméně obojí na konci roku 2015 ztratilo podporu vývojářů Androidu a nedoporučuje se nadále používat [9].

Android Studio obsahuje všechny nástroje potřebné k vývoji aplikací a jejich testování. Jeho součástí je výkonný systém Gradle pro automatické sestavování aplikace, který například usnadňuje používání externích knihoven. AS dále obsahuje flexibilní editor pro tvorbu uživatelského rozhraní, ve kterém na rozdíl od vývojového prostředí Eclipse má mnohem pokročilejší funkce napovídání kódu.

6.1.3 Build systém

Sestavovacím systémem v Android Studiu je Gradle. S ním je standardně používán speciální plugin, který standardní Gradle rozšiřuje o možnost sestavovat Android aplikace.

Gradle je závislý na 2 konfiguračních souborech, na základě kterých sestaví výslednou aplikaci. První je umístěn v `<projekt>/app/build.gradle`, kde `app` je standardním názvem modulu (např. aplikace nebo knihovny). Projekt se může skládat z více (závislých) modulů a aby o všech Gradle věděl, musí být k dispozici i druhý konfigurační soubor `<projekt>/build.gradle`. První soubor je specifický pro daný modul (`app`) a druhý pro celý projekt, kde se mohou konfigurovat závislosti všech modulů v projektu. Projekt s touto aplikací obsahuje právě jeden modul s názvem `app`.

V konfiguračním souboru modulu `app` jsou uvedeny všechny externí závislosti a konfigurace výsledné aplikace, např.:

[2] `https://www.jetbrains.com/idea/`

- `minSdkVersion`: minimální API level, na kterém bude možné aplikaci spustit (nastaveno na 16),
- `compileSdkVersion`: API level, proti kterému se aplikace sestavuje (nastaveno na 23). Zde je doporučováno nastavit vždy nejvyšší možný level, aby aplikace podporovala funkčnosti z nejnovější verze Androidu. V situaci, kdy by byl sestavovací API level nastaven např. na 22, aplikace by byla spuštěna na zařízení s API 23 a použila by funkčnost zavedenou v API 23, mohlo by dojít k chybě.

6.1.4 Testovací prostředí

Testovacím prostředím jsou zde myšlena testovací zařízení a DCIx server, nikoliv testování z pohledu ověřování kvality zdrojového kódu.

Aplikace byla vyvíjena a průběžně testována na dvou fyzických zařízeních:

- **Samsung Galaxy S4**: Android Lollipop 5.0.1 (API level 21), 5" full-hd displej s hustotou ~441 dpi, rozlišením 1080x1920 a kategorií displeje `xxhdpi`,
- **Symbol MC9200**: Android KitKat 4.4.4 (API level 19), 3,7" VGA displej s hustotou ~216 dpi, rozlišením 480x640 a kategorií displeje `hdpi`.

Aplikace byla taktéž průběžně testována i na emulátoru s API level 16 z důvodu ověření kompatibility s nejnižším API levelem.

Od zadavatele byl zapůjčen notebook s nainstalovaným DCIx serverem a databází pro testovací účely.

Zařízení s nainstalovanou aplikací a notebook s DCIx musí být připojeny ke stejné síti. Nejčastěji byla využívána domácí Wi-Fi síť. V případě absence domácí nebo podnikové sítě (především při předvádění vedoucímu této práce) bylo testování prováděno vytvořením Wi-Fi sítě ze zařízení Samsung Galaxy S4. K této síti se notebook a zařízení s aplikací připojilo (Galaxy S4 je standardně již připojené) a aplikace bylo možné testovat.

6.2 Uživatelské rozhraní

Základním požadavkem na uživatelské rozhraní je jeho přehlednost, jednoduchost a zejména rozšiřitelnost z důvodu očekávaného nárůstu množiny podnikových procesů. Aplikace musí být nějakým způsobem konfigurovatelná, např. nastavení údajů pro připojení k DCIx. Na to navazuje uživatelské

rozhraní sloužící k zadání přihlašovacích údajů uživatele k DCIx. Po přihlášení musí být nějakým způsobem zobrazen seznam spustitelných transakcí. Po spuštění transakce musí aplikace vhodně zobrazit obsah přijaté transakční obrazovky a nabídnout uživateli výběr dalšího kroku. Po dokončení transakce se uživateli opět zobrazí seznam transakcí. Tento proces střídání hlavních uživatelských aktivit znázorňuje diagram na obrázku 6.1.

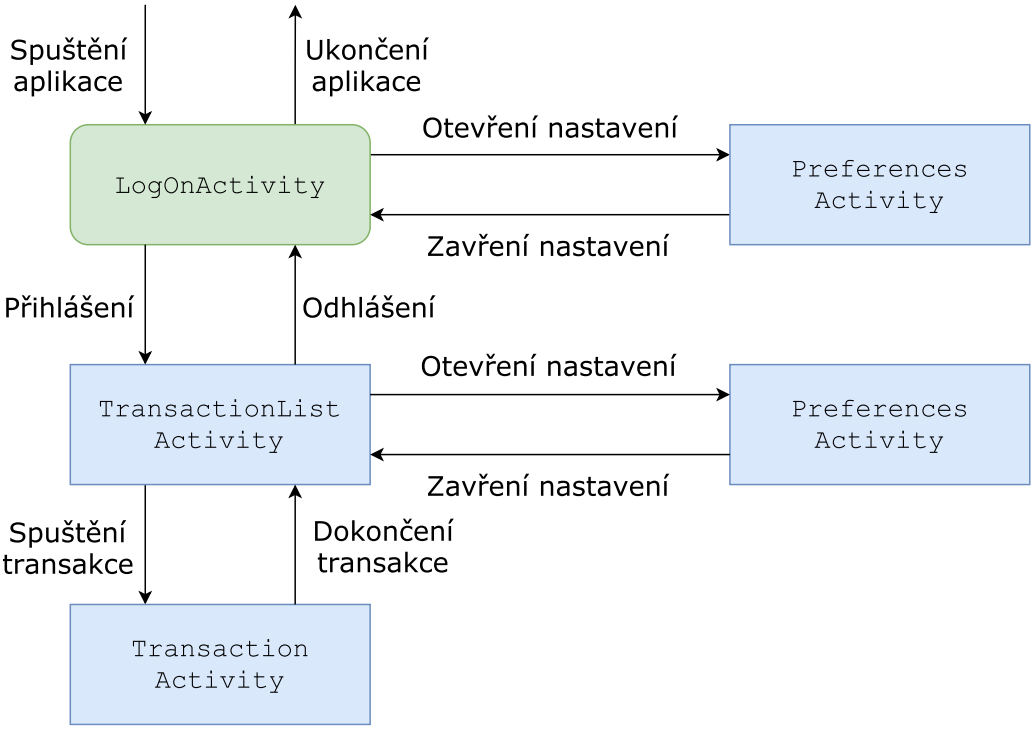
V průběhu transakce musí být zajištěno, aby uživatel neopustil aktivitu s danou transakcí v rámci aplikace. Pochopitelně, že uživatel může tlačítkem HOME na zařízení aplikaci minimalizovat, o tom ale více později.

6.2.1 Uživatelské aktivity

Hlavní uživatelské obrazovky (aktivity) jsou 4. Uživatel mezi nimi může procházet podle toho, na jaké aktivitě zrovna je. Diagram možných přechodů a celkově diagram střídání aktivit znázorňuje obrázek 6.1. Význam aktivit na obrázku je následující:

- `LogonActivity` je vstupní aktivita aplikace a současně aktivita sloužící k přihlášení k DCIx.
- `PreferencesActivity` je aktivita, která umožňuje nastavení aplikace (např. připojení k DCIx, změnu velikosti písma atd.). Tuto aktivitu je možné otevřít jak z přihlašovací aktivity, tak z aktivity se seznamem transakcí. V druhém případě je nastavení připojení k DCIx skryto, neboť je uživatel již připojený. Toto nastavení tedy lze měnit pouze pokud je uživatel odhlášený.
- `TransactionListActivity` je aktivita se seznamem transakcí přihlášeného uživatele.
- `TransactionActivity` je aktivita, ve které uživatel prochází spuštěnou transakci.
- `BaseDrawerActivity` je aktivita v podobě abstraktní třídy, od které všechny ostatní aktivity dědí.

Jedním z dalších požadavků na uživatelské rozhraní je indikace síly signálu připojení a stavu baterie, protože aplikace běží v celoobrazovkovém režimu a tyto informace jsou v notifikační liště zakryty. Zobrazení těchto indikátorů je tedy žádoucí na každé aktivitě aplikace. Aby se zbytečně neopakoval kód, který by danou funkčnost realizoval, byla vytvořena společná aktivita `BaseDrawerActivity` jako rodič všech aktivit, které chtějí indikátory zobrazovat.



Obrázek 6.1: Diagram střídání hlavních uživatelských aktivit

`BaseDrawerActivity` funguje jako wrapper (obal), který rozšiřuje layout ostatních aktivit o 2 UI prvky. Prvním je lišta ve spodní části obrazovky s časem, přihlášeným uživatelem a indikátory stavu signálu a baterie. Druhým prvkem je tzv. `navigation drawer`, což je známé menu, které se rozbalí tažením z levého okraje obrazovky. Každá aktivita, která chce spodní informační lištu mít, musí být oděděna od `BaseDrawerActivity`. Vyjíždějící menu povinné není. `BaseDrawerActivity` obsahuje abstraktní metodu `setUpDrawer()`, kterou všichni potomci musejí implementovat. Ty aktivity, které vyjíždějící menu nemají, jednoduše zděděný layout pro menu uzamknou, viz `PreferencesActivity` a `TransactionActivity`. Ostatní aktivity v překryté metodě specifikují položky svého menu.

Stejné funkcionality by bylo dosaženo použitím jedné aktivity a několika fragmentů. Od tohoto řešení bylo upuštěno, neboť integrovat všechny fragmenty do jedné aktivity by mohlo zcela jistě vést ke zbytečně složitému a nepřehlednému kódu.

Při vytváření obecně jakékoliv aktivity se její layout nastavuje voláním `setContentView(<ID layoutu>)` v metodě `onCreate()` (kód 6.1). Pro koncové aktivity se nic nemění. Situace je jiná u `BaseDrawerActivity`, která v `onCreate()` nic nevolá. Její layout totiž obsahuje pouze společnou dolní lištu, levé vyjíždějící menu a hlavně kontejner pro layout koncové aktivity.

```
1  @Override
2  protected void onCreate(Bundle savedInstanceState) {
3     super.onCreate(savedInstanceState);
4     setContentView(R.layout.activity_transaction);
5  }
```

Kód 6.1: Zkrácená verze `onCreate()` v `TransactionActivity`

Spojení layoutu koncové aktivity s hlavním obalovým layoutem se děje v (překryté) metodě `onCreateView(<ID layoutu>)`. Vstupem této metody je ID layoutu koncové aktivity a jedná se právě o volání `onCreateView(<ID layoutu>)` koncových aktivit. V této metodě načte `BaseDrawerActivity` jak svůj layout, tak layout koncové aktivity podle předaného ID. Tento layout potom vloží do kontejneru pro layout koncové aktivity ve svém layoutu. Jakmile toto udělá, zavolá i ta `super.onCreateView(<layout>)` (kód 6.2). Následně zavolá své 3 abstraktní metody `setUpToolbar()`, `setUpWidgets()` a `setUpDrawer()`. V každé této metodě si potomci definují strukturu horní lišty (toolbaru), levého menu a ostatních UI elementů. Tyto metody nejsou v zásadě nutné. Jejich účel je pouze zpřehlednění kódu. Nutnost jejich implementace předejde situacím „zapomenutí“ na nastavení např. toolbaru.

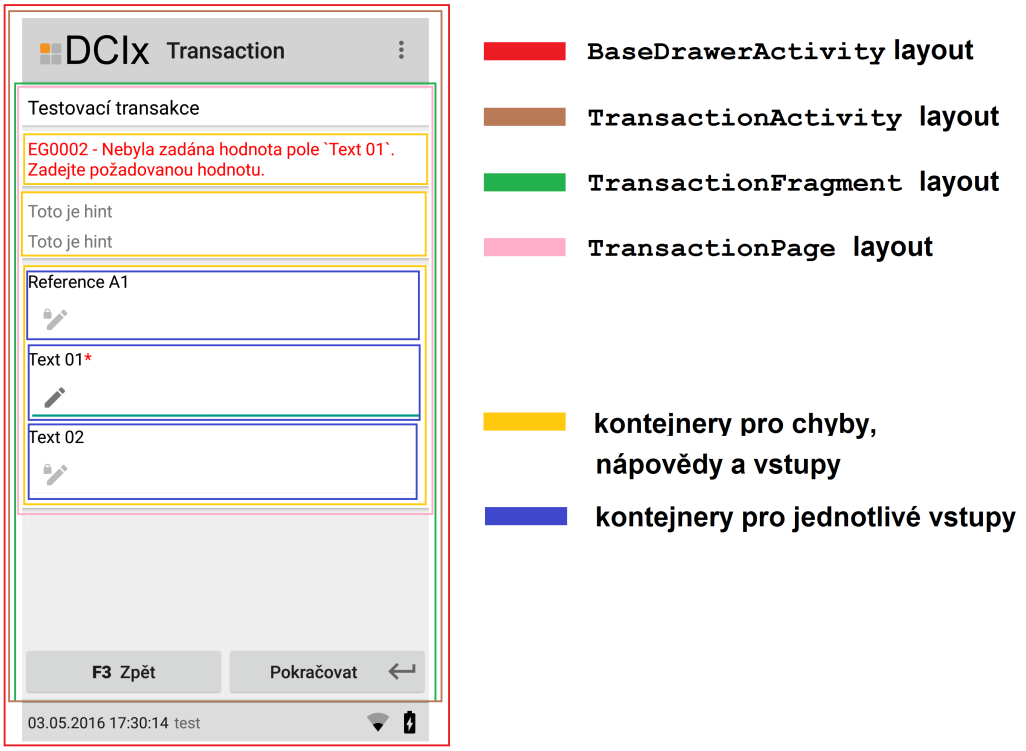
```
1  @Override
2  public void setContentView(int layoutResID) {
3     // base drawer activity layout
4     mDrawerLayout = (DrawerLayout)
         mInflater.inflate(R.layout.activity_base_drawer, null);
5     // inner activity container
6     FrameLayout container = (FrameLayout)
         mDrawerLayout.findViewById(R.id.container);
7     // inflate inner activity into it's container
8     getLayoutInflater().inflate(layoutResID, container, true);
9     // set content view of this activity
10    super.setContentView(mDrawerLayout);
11 }
```

Kód 6.2: Upravená ukázka spojení layoutů v `onCreateView()` v `BaseDrawerActivity`

Levé vyjíždějící menu, tzv. navigation drawer, bylo vybráno z důvodu jednoduché navigaci v aplikaci a současně snadné rozšiřitelnosti. Tento způsob navrhování uživatelského rozhraní je také široce rozšířen v aplikacích pro Android a je společností Google přímo doporučován, více v sekci 6.2.2.

TransactionActivity

Všechny aktivity mají layout relativně jednoduchý. Výjimkou je aktivita `TransactionActivity`, jejíž layout znázorňuje obrázek 6.2. Červený rámeček značí layout rodičovské aktivity (`BaseDrawerActivity`), do něhož byl vložen layout koncové aktivity, která do sebe dále integruje layout svého fragmentu `TransactionFragment`. Ten se skládá ze dvou tlačítek pro navigaci v transakci a z layoutu transakční obrazovky (růžový rámeček, třída `TransactionPage`). Ten dále obsahuje 3 kontejnery: pro chyby, nápovědy a vstupy. Kontejnery pro jednotlivé vstupy (modré rámečky) jsou instance třídy `TransactionInputContainer` a skládají se z titulku vstupu (`TextView`) a pole pro vstup hodnot (`EditText`).



Obrázek 6.2: Rozložení aktivity `TransactionActivity`

6.2.2 Styly a material design

Hlavní styl aplikace dědí od stylu `Theme.AppCompat.Light.NoActionBar` a doplněním atributu `android:windowFullscreen` s hodnotou `true` se aplikace stane celoobrazovkovou. Rodičovský styl dále specifikuje, že aplikace nebude mít standardní lištu (`ActionBar`). Místo toho je použít nový widget `Toolbar`. Tuto změnu přinesl Android Lollipop. Druhou důležitou informací

z názvu stylu (`Light`) je to, že motiv aplikace bude orientován ve světlých barvách. Opakem je styl `Dark`, kdy by aplikace byla ve tmavých barvách.

Volitelná velikost písma

Styly jsou dále využity ve spojení s možností nastavení velikosti písma v aplikaci, což bylo jedním z požadavků zadavatele. Uživatel může v nastavení aplikace zvolit, jakou velikost písma chce (malé, střední, velké). Pod každou touto kategorií jsou ukryté další 4 velikosti, neboť v aplikaci je také nezbytné mít některé písmo větší než jiné.

Byly vytvořeny 3 styly: `FontSizeSmall` pro malé, `FontSizeMedium` pro střední a `FontSizeLarge` pro velké písmo a dále pak 4 atributy: `font_small`, `font_medium`, `font_large` a `font_xlarge`. Každý styl (kategorie) obsahuje všechny 4 atributy s konkrétní hodnotou velikosti písma. Například atribut `font_small` má ve stylu `FontSizeSmall` hodnotu `12sp`, ve stylu `FontSizeLarge` pak hodnotu `20sp` atd.

```
1 <style name="FontStyleLarge">
2   <item name="font_small">@dimen/font_large_small</item>
3   <item name="font_medium">@dimen/font_large_medium</item>
4   <item name="font_large">@dimen/font_large_large</item>
5   <item name="font_xlarge">@dimen/font_large_xlarge</item>
6 </style>
```

Kód 6.3: Ukázka definice stylu `FontStyleLarge`

Ukázka 6.3 obsahuje konkrétní definici stylu `FontSizeLarge`. Konkrétní hodnoty atributů jsou uloženy v souboru `res/values/dimens.xml`, který slouží k ukládání dimenzí, rozměrů, velikostí atd.

Každý jednotlivý element, který nějakým způsobem pracuje s textem, ať už jej zobrazuje nebo slouží ke vstupu, musí mít ve své definici velikosti textu právě jeden ze 4 atributů (např. `android:textSize="?attr/font_medium"`, kde `?attr/` referencuje hodnotu atributu). Jakmile je potom jeden ze 3 stylů aplikovaný, je hodnota atributu tohoto stylu použita všude, kde je daný atribut referencován. Tak dochází ke změnám velikosti textu. Změna velikosti textu musí být napříč spuštěními aplikace perzistentní. Proto při každém spuštění jednotlivých aktivit musí být aplikován naposledy použitý styl s velikosti. Styly jsou reprezentovány čísly 0, 1 a 2 pro malý, střední a velký text. O aplikaci stylu se stará třída `FontSizeTools`, která z úložiště aplikace načte číslo stylu a ten aplikuje (proto změna velikosti písma vyžaduje restart aplikace). Aplikace stylu probíhá v metodě `onCreate()` v `BaseDrawerActivity`, tzn. že ji implicitně zavolají všechny koncové aktivity.

Material design

Aplikace se snaží dodržovat prvky material designu. Material design lze považovat za sadu pravidel, jak by mělo být uživatelské rozhraní vypadat a jak by mělo být rozvrženo (např. velikosti ikony v menu, použití lišty `toolbar` atd.). Material design byl vytvořen společností Google za účelem vytvoření společného vzhledu aplikací pro Android. Pokud budou vývojáři material design dodržovat a používat, zvyknou si uživatelé těchto aplikací na podobný vzhled a intuitivně budou vědět, jak aplikaci ovládat.

Material design byl zaveden ve verzi Android 5.0 a aby byla zachována zpětná kompatibilita i se staršími verzemi, jsou k dispozici tzv. support knihovny. Např. knihovna `AppCompat-v7` zajišťuje kompatibilitu a material design na starších zařízeních až k API level 7 a je závislá na support knihovně `Support-v4`, která umožňuje používat například fragmenty až k API level 4 (fragmenty byly přidány v API level 11).

6.2.3 Indikátory signálu a baterie

Jedním ze základních požadavků na aplikaci je indikace síly signálu a stavu baterie. Oba tyto indikátory jsou umístěny ve spodní liště každé aktivity, aby byly vždy viditelné. Informace o síle signálu a stavu baterie rozesílá systém broadcastem. Každý z indikátorů má svou třídu, konkrétně se jedná o třídy `BatteryImageView` a `WifiSignalImageView`. Obě obsahují broadcast přijímač, pomocí kterého přijímají data pro daný indikátor.

Indikátor baterie definuje celkem 4 úrovně nabití: 0-10%, 11-40%, 41-70% a 71-100%. Na základě přijatého intentu si třída vypočítá úroveň, do které současné nabití spadá, a podle toho zobrazí příslušnou ikonu.

Podobně to funguje v případě indikátoru síly signálu. Z přijatého intentu se získá síla signálu a podle její kategorie se zobrazí příslušná ikona. Ta je rozdělena na čtvrtiny, jejichž zaplnění zespoda reprezentuje sílu signálu.

Indikátor signálu také podporuje stavy, kdy je Wi-Fi vypnutá nebo zapnutá, ale nepřipojená. V případě vypnuté Wi-Fi je přes ikonu zobrazený červený kříž. V druhém případě je v ikoně zobrazen modrý otazník.

6.3 Architektura MVP

Ze 3 představených architektur v sekci 5.1 byla vybrána architektura MVP (Model-View-Presenter). Presenter je dále označován jako „prezentér“. Jejími přednostmi je jednoduchý a přehledný kód, oddělená logika od prezentační vrstvy a snadná testovatelnost pomocí JUnit testů. To byly přední

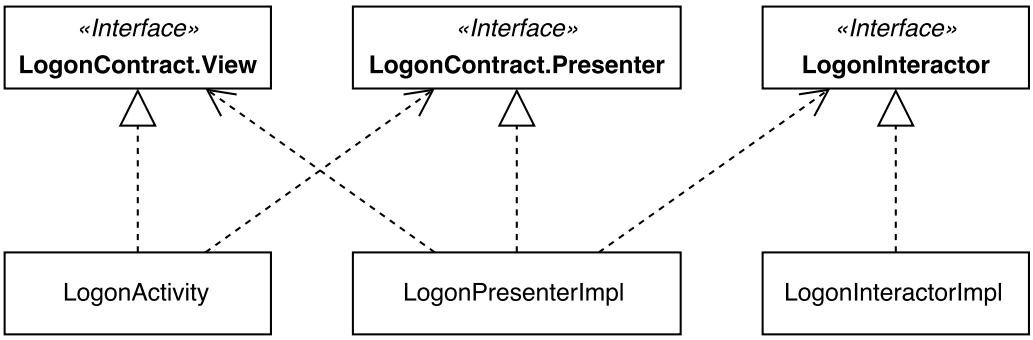
důvody pro volbu MVP.

Tato kapitola popisuje způsob implementace architektury, jejímž výsledkem je přehlednější kód. Aplikační logika je obsažená na jednom místě a prezentéři jsou testovatelní na JVM, neboť neobsahují žádný platformě závislý kód.

6.3.1 Realizace MVP

MVP je realizováno u každého View, které zobrazuje data a vyžaduje nějakou aplikační logiku. Jedná se o aktivity, fragmenty a transakční obrazovku, která je sice uvnitř fragmentu, ale sama obsahuje vlastní logiku, na kterou je vhodné mít vlastního prezentéra.

Každý prezentér a View spolu komunikují pomocí speciální dohody (contract, viz analýza). Tato dohoda obsahuje dvě rozhraní pro obousměrnou komunikaci mezi nimi. Rozhraní pro prezentéra specifikuje akce, které může uživatel provést na View. Naopak rozhraní pro View specifikuje akce, které jsou po View vyžadovány prezentérem. Obě rozhraní jsou vždy ve společné třídě s názvem složeným ze jména konkrétního View a postfixu `Contract` (např. `LogonContract`). Prezentéři, kteří vyžadují komunikaci s DCIx, ještě k tomu využívají třídu zvanou `*Interactor` (dále jen „interaktor“). Ty zařizují dodání dat pro prezentéry, kteří data předají svým View.

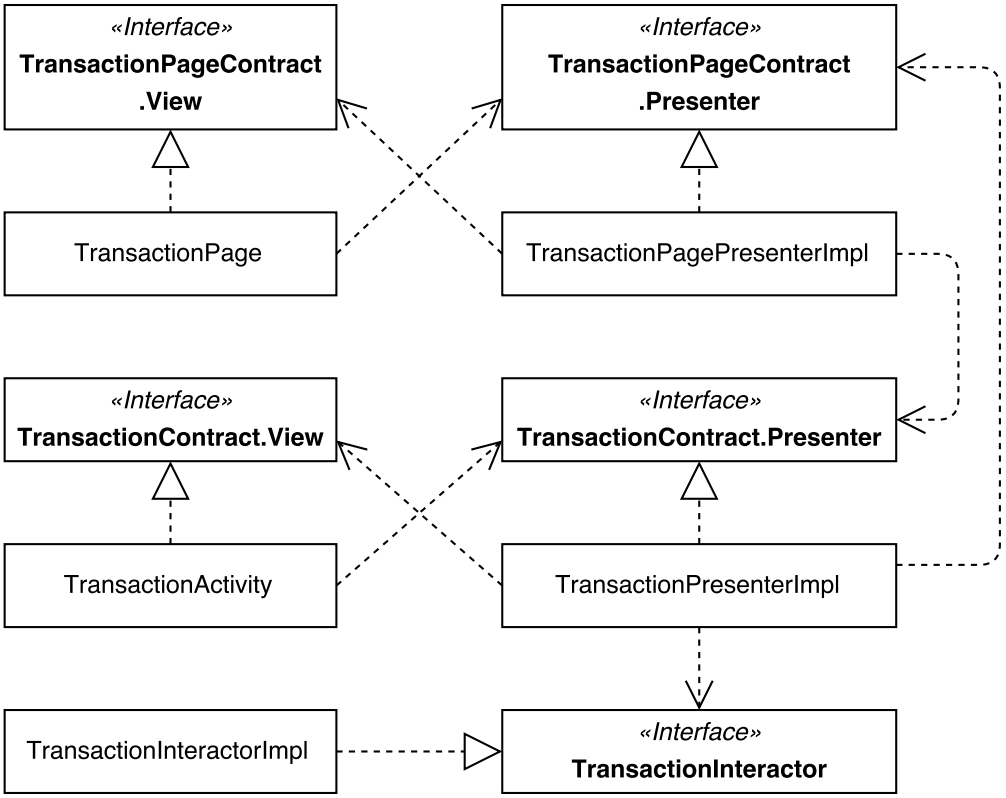


Obrázek 6.3: UML diagram imlementace MVP přihlašovací aktivity

Příkladem lze uvést přihlašovací aktivitu `LogonActivity`. Ta implementuje rozhraní `LogonContract.View` a naopak její prezentér, instance třídy `LogonPresenterImpl`, implementuje rozhraní `LogonContract.Presenter`. UML diagram těchto vztahů popisuje obrázek 6.3. Na obrázku je také zachycen vztah mezi přihlašovacím prezentérem a jeho interaktorem. Interaktoři slouží ke komunikaci s DCIx a jsou celkem 3:

- `LogonInteractor`: využíván `LogonPresenterImpl` (viz obr. 6.3) za účely přihlašování,

- `TransactionInteractor`: slouží k načítání transakčních obrazovek a je využíván třídou `TransactionPresenterImpl`,
- `LogoffInteractor`: využíván třídou `TransactionListPresenterImpl` za účely ohlášení. Tento interaktor je využit také v případě neřízené chyby na DCIx, kdy se její popis zašle do aplikace, zobrazí a uživatel je poté automaticky odhlášen.



Obrázek 6.4: UML diagram MVP transakční aktivity a transakční obrazovky

U všech ostatních View je situace stejná. Výjimku tvoří dvě dvojice View a prezentér transakční obrazovky a View a prezentér transakční aktivity. Ty spolu musí komunikovat, a proto jsou jejich prezentéři propojení, viz obr. 6.4. Důvodů komunikace je několik. Prvním je ten, že dotazy na DCIx iniciuje prezentér transakční aktivity, který musí o spuštění a dokončení dotazu informovat prezentéra transakční obrazovky (viz obr. 6.2 a `TransactionPage` layout). Ten na základě toho ví, že pokud mu zrovna přijdou data od skeneru, tak je uloží do bufferu, neboť se na další transakční obrazovku teprve čeká.

6.4 Dependency injection

Aplikace využívá principy dependency injection (DI), protože zpředhledňují a usnadňují kód a zjednodušují jeho testování. Jako DI framework byl zvolen Dagger 2 (viz analýza v 5.2). Ve srovnání s Guice využívá statické analýzy ke generování kódu, který provádí vkládání závislostí. Tím se zbavuje kompletně reflexe, která je hlavním prvkem Guice a částečně v první verzi Daggeru. Princip fungování DI a konkrétně Daggeru 2 je vysvětlen v sekci 5.2.3.

6.4.1 Komponenty a moduly

Aplikace definuje celkem 6 komponent, 6 modulů (každý po jedné komponentě) a 2 scopy (rozsahy). Každá aktivita v aplikaci má svoji komponentu a své závislosti, které jsou pro ní typické. Nejčastěji se jedná o instance View a prezentérů z MVP architektury. Díky vlastní komponentě každé aktivity existují tyto závislosti pouze tehdy, když je aktivita aktivní (zobrazena). Jakmile se aktivita ukončí, dojde k uvolnění alokovaných závislostí a tím tak k šetření paměti.

Aplikace má jen 4 aktivity, zbylé 2 komponenty slouží k jiným účelům:

- `AppComponent`: hlavní komponenta celé aplikace. Existuje po celou dobu běhu aplikace a současně s ní i instance závislostí, které poskytuje. Komponenta je vytvářena v aplikační třídě `DciApplication`. To je třída, která dědí od `Application` a reprezentuje celou aplikaci. Současně musí být třída uvedena v manifestu aplikace, aby bylo možné ji používat,

- `UserComponent`: komponenta těsně nad `AppComponent`. Drží kontext o přihlášeném uživateli (nepřihlášený uživatel je také stav přihlášení), informace o připojení k DCIx a další. Tato komponenta je vytvářena v `onResume()` přihlašovací aktivity. Důvodem je to, že uživatel může v nastavení změnit přihlašovací údaje k DCIx a tyto přihlašovací údaje musí být vždy těsně před přihlášením aktuální. Tím, že se bude komponenta vytvářet v `onResume()` je zajištěno, že po každém zobrazení přihlašovací aktivity budou závislosti v komponentě poskytovat správné údaje (protože se nové instance vytvoří spolu s komponentou).

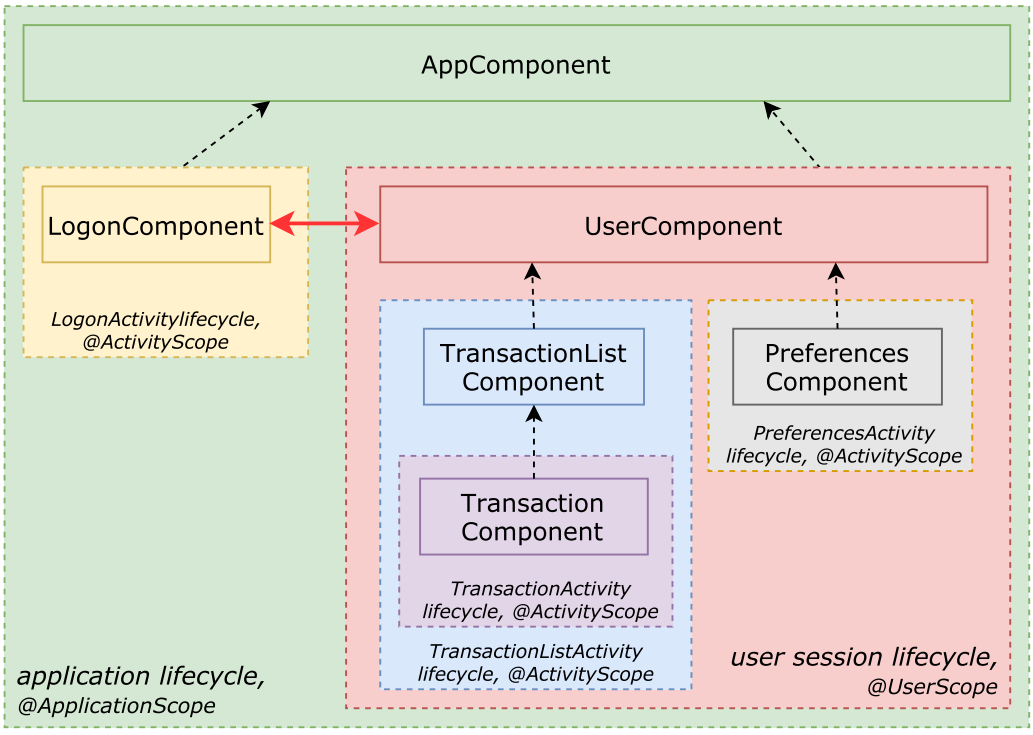
Vytváření komponent pro aktivity se provádí v `onCreate()`, aby v dalších metodách životního cyklu aktivity byly závislosti dostupné.

Aplikace dále definuje 2 scopy (rozsahy):

- **`UserScope`**: scope, který vymezuje životnost závislostí týkajících se přihlášeného uživatele. V principu je tento scope stejný se scopem hlavní komponenty, protože je vytvářen vždy v `LogonActivity`. Od hlavního scopu se liší právě tím, že je vytvořena nová instance vždy při zobrazení přihlašovací aktivity (viz 6.4.1);
- **`ActivityScope`**: scope, který vymezuje životnost závislostí v rámci aktivit. Opět, jméno scopu má pouze informativní charakter. Jeho významu je docíleno časem a místem, kde se vytváří komponenta s tímto scopem, tedy vždy v `onCreate()`.

6.4.2 Závislost komponent

Hlavní a nejvýše položenou komponentou je `AppComponent` (viz předchozí sekce). Tato komponenta poskytuje závislosti, které jsou napříč aplikací sdílené a stačí od nich udržovat pouze jednu instanci (jedináčci). Ostatní komponenty jsou subkomponenty a dědí tedy všechny závislosti definované v modulech hlavní komponenty. Závislosti komponent popisuje diagram na obrázku 6.5. Barevně jsou na diagramu zobrazeny životnosti jednotlivých komponent.



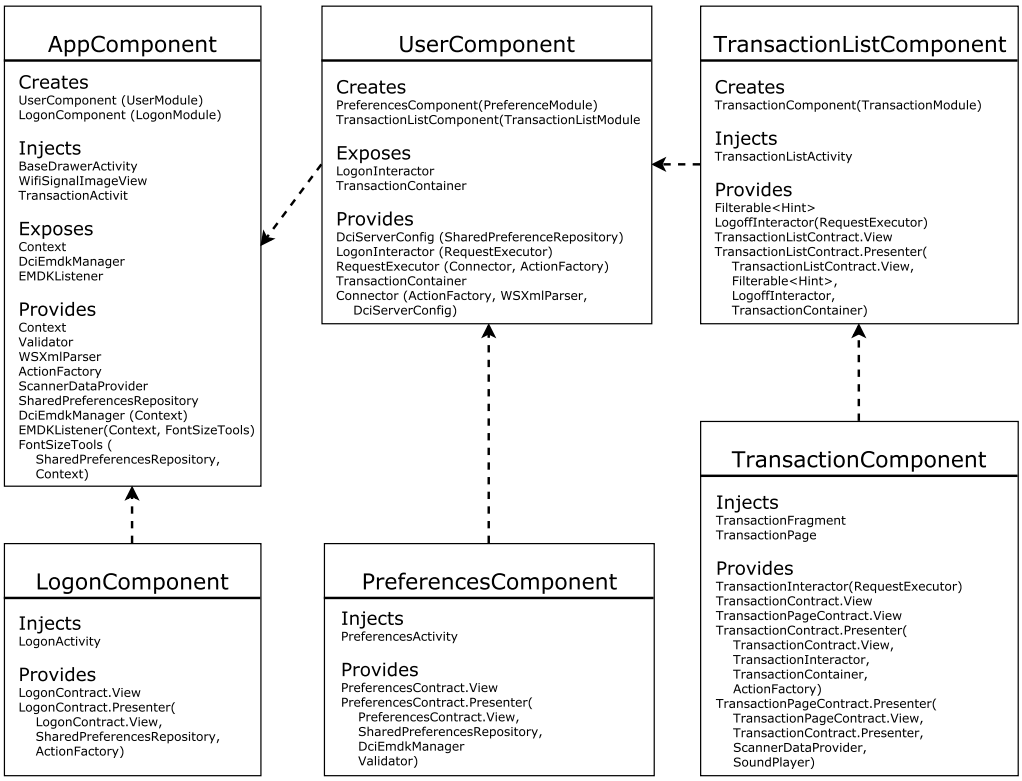
Obrázek 6.5: Závislost komponent v aplikaci

Z diagramu mohou být na první pohled nejasné dvě věci. První je závislost `PreferenceComponent` na `UserComponent`, kde je důvod takový, že v `PreferenceActivity` je vyžadována informace o přihlášeném uživateli. Pokud je uživatel přihlášený, není v aktivitě umožněno měnit nastavení připojení k DCIx. Další otázkou může být, proč `UserComponent` není závislé na `LogonComponent`. Důvod je takový, že `UserComponent` obsahuje nejen kontext přihlášeného uživatele, ale také přihlašovací údaje k DCIx. Pokud by `UserComponent` závisela na `LogonComponent`, pak by se z `LogonComponent` nedalo přihlásit k DCIx, protože by neměla odkud získat přihlašovací údaje. Otázkou může být, proč tedy přihlašovací údaje neposkytuje `AppComponent`. Opět, viz výše. Jedná se o to, že uživatel může před přihlášením měnit nastavení připojení k DCIx, které se vždy při zobrazení přihlašovací aktivity musí načíst (refresh `UserComponent`). Tuto interakci popisuje červená šipka. `LogonComponent` využívá `UserComponent` k získání připojovacích údajů k DCIx a `UserComponent` získává od `LogonComponent` informaci o přihlášeném uživateli, kterou následně udržuje po dobu přihlášení, tj. do té doby, než se opět zobrazí přihlašovací aktivita (odhlášený uživatel).

Poslední obrázek (6.6) této sekce zobrazuje detail všech komponent. Z obrázku je patrný graf závislostí. Například v `AppComponent` je většina závislostí nezávislá na jiných. Ty závislé je komponenta sama schopna dodat (např. `Context`). Jelikož aplikace využívá k realizaci závislostí komponent subkomponenty, jsou všechny závislosti rodičovské komponenty přístupné jejím potomkům. Například `TransactionContract.Presenter` v komponentě `TransactionComponent` závisí na třídě `ActionFactory`, kterou poskytuje sama hlavní komponenta. Závislosti jsou tedy propagovány až na nejnižší úroveň.

6.5 Použití EMDK

Před popsáním samotného průchodu transakcí a komunikace s DCIx je třeba vysvětlit, jak vlastně funguje skenování čárových kódů (ČK). Cílová zařízení této aplikace jsou převážně podnikové mobilní počítače a ruční skenery. Požadavkem je použít softwarové vybavení těchto zařízení a využít jej ke skenování ČK. Softwarovým vybavením je EMDK 4.0 více popsané v sekci 4.5. Nejsou to ovšem jen podniková zařízení, kde bude aplikace používána. Aplikaci bude možné používat i na klasických mobilních telefonech. Z toho vyplývá, že je nutné řešit a rozlišovat situaci, kdy aplikace běží na podnikovém zařízení a kdy na mobilním telefonu. V obou případech musí aplikace podporovat skenování čárových kódů.



Obrázek 6.6: Komponenty a jejich závislosti

K rozlišení těchto situací se používá třída `PackageManager` Android frameworku. Přes tohoto manažera je možné získat detailní informace o nějakém balíku (aplikaci) na zařízení. Jestli je EMDK na zařízení nainstalováno, musí být přístupné tomuto manažerovi a ten o něm může vrátit základní informace. Pokud EMDK na zařízení není, není pak ani správce balíků schopný načíst jeho informace a vyhodí výjimku. Jak tedy zjistit, jestli je EMDK na zařízení nainstalováno, popisuje ukázka 6.4.

```
1 PackageManager pm = mContext.getPackageManager();
2 try {
3     pm.getPackageInfo(EMDK_PACKAGE_NAME, GET_ACTIVITIES);
4     mEMDKInstalled = true;
5 catch (NameNotFoundException exception) {
6     mEMDKInstalled = false;
7 }
```

Kód 6.4: Způsob detekce instalovaného EMDK

Tato detekce se provádí ve třídě `DciEmdkManager`, která slouží i k aktivaci DataWedge profilů (viz dále). Instance tohoto manažera se vytváří ihned po

startu aplikace a je jednou z poskytovaných závislostí hlavní komponenty. Kód v ukázce 6.4 se provádí už v konstruktoru třídy, takže informace o dostupnosti EMDK jsou k dispozici ihned po startu aplikace. Pokud EMDK není při běhu dostupné, používá se místo vestavěného skeneru aplikace Barcode Scanner a kamera zařízení.

6.5.1 Aktivace profilů

Předpokládejme podnikové zařízení s EMDK. Aby bylo EMDK možné v aplikaci využívat a konfigurovat, musí se využít DataWedge profily (viz 4.5.2). Tyto profily slouží k definování způsobu dodání naskenovaných dat do aplikace. Profily umožňují konkrétně specifikovat aktivity, na které se profil uplatní. Dle domluvy se zadavatelem byly vytvořeny 2 profily:

- `DciProfileIntent`: profil, který dodává data v podobě broadcast intentu. Akce intentu je specifikována v konfiguraci profilu a je rovna řetězci `"cz.aimtec.dci.android.scanner"`. Profil je aplikován pouze na aktivitu `TransactionActivity` z důvodu možné konfigurace dodatečného zpracování naskenovaných dat (např. připojení klávesy `ENTER`, která odesílá transakční obrazovku),
- `DciProfileKeystroke`: profil, který dodává nezměněná data v podobě simulovaných stisků kláves z klávesnice. Profil je aplikován na všechny aktivity mimo `TransactionActivity`.

Jsou-li profily nakonfigurované, musí se v zařízení ještě aktivovat. Aktivace profilů probíhá vždy při spuštění aplikace, aby při jejím používání již byly aktivní. K aktivaci profilů se využívá `EMDKManager` z knihovny EMDK. Instanci `EMDKManager` lze získat voláním `EMDKManager.getEMDKManager( context, listener)`, kde `listener` je posluchač se dvěma callbacky. Prvním je `onOpened(EMDKManager)`, který je zavolán, když byla instance manažera úspěšně získána. Druhým je `onClosed()`, který naopak slouží k uvolnění instance manažera získané v `onOpened()`.

V `onOpened()` se z instance EMDK manažera získá manažer profilů, který umí spravovat DataWedge profily a který umí profily v zařízení aktivovat. Bylo vypozorováno, že synchronní způsob aktivace profilů je velice pomalý a prodlužuje spouštění aplikace. Tento problém byl vyřešen asynchronním zpracováním pomocí třídy `AsyncTask`, viz ukázka 6.5. Na ukázce je vidět metoda spouštěná asynchronně, v níž se voláním metody `processProfile(...)` aktivuje daný profil. Prvním parametrem je název profilu, druhým je flag indikující, že se profil vytvoří a aktivuje, případně

jen aktivuje a poslední parametr jsou dodatečná data této akce (nejsou potřeba). Výsledky aktivací jsou ukládány do mapy, kde klíčem je jméno daného profilu a hodnotou výsledek aktivace. Mapa je dále již na hlavním vlákně zpracována a pokud byla aktivace nějakého profilu neúspěšná, je o tom uživatel informován.

```
1  @Override
2  protected Map<String, EMDKResults> doInBackground(String...
       profiles) {
3     int pCount = profiles.length;
4     Map<String, EMDKResults> results = new HashMap<>(pCount);
5     for (String pName : profiles) {
6        results.put(
7           pName,
8           pManager.processProfile(pName, SET, new String[1])
9        );
10    }
11    return results;
12 }
```

Kód 6.5: Asynchronní aktivace EMDK profilů

Způsob skenování dat a jejich příjem je popsán později v sekci 6.8.3.

6.6 Komunikace aplikace s DCIx

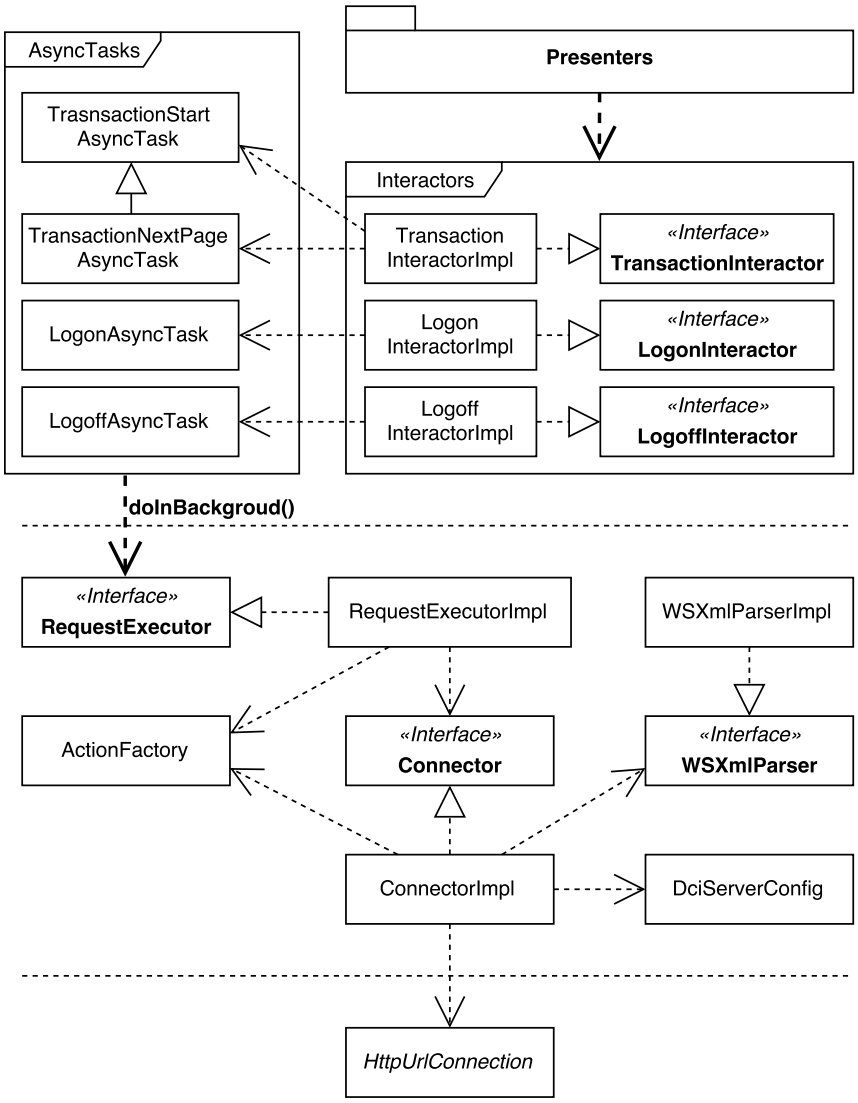
Aplikace vyžaduje komunikaci s DCIx serverem, bez něhož je nepoužitelná. Struktura přenášených dat, jejich popis a způsob komunikace byl vysvětlen v sekci 3.3. Tato sekce se zabývá komunikací s DCIx na nižší úrovni a popisuje významné třídy, které komunikaci zprostředkovávají, viz obr. 6.7. Z obrázku je zřejmé, že veškerá komunikace je iniciována prezentéry z MVP architektury, kteří pomocí svých interaktorů odesílají dotazy (requesty) na server.

Aby veškeré requesty na server byly prováděny asynchronně, jsou ještě před jejich samotným odesláním předstrčeny instance `AsyncTask`, které zajistí provedení requestu ve vedlejším vlákně. Tyto instance využívají služeb třídy `RequestExecutor`, která připravuje requesty pro `Connector`, který je pak vyloženě odesílá na DCIx. `RequestExecutor` má 4 hlavní metody, z jejichž názvu je patrný jejich účel:

- **`doUserLogon()`**: sestaví request pro přihlášení uživatele,

- `doTransactionStart()`: sestaví request pro spuštění transakce,
- `doTransactionNextPage()`: sestaví request pro získání další transakční obrazovky,
- `doUserLogoff()`: sestaví request pro odhlášení uživatele.

Request následně obdrží `Connector`, který na základě `DciServerConfig` vytvoří spojení se serverem a request odešle. Návratovou hodnotou requestu je vždy `Action`. Ta obsahuje buď úspěšně zpracovanou odpověď ze serveru, nebo případnou chybu. Odesílání na nižší úrovni popisuje následující sekce.



Obrázek 6.7: UML diagram tříd zprostředkujících komunikaci s DCIx

6.6.1 Odesílání dotazů

K odesílání requestů na server je použita třída `HttpURLConnection` z balíku `java.net`, který je součástí Android frameworku (proto není potřeba jiné externí knihovny). Dalším důvodem je to, že knihovna umožňuje přístup k hlavičkám HTTP requestu a odpovědi, což aplikace ve specifických situacích vyžaduje.

Bezchybný průchod

Předpokládejme bezchybné vykonání requestu a úspěšné získání odpovědi. `Connector` z instance `URL` vytvoří `HttpURLConnection`. Tomu následně nastaví maximální časové limity (timeout) pro připojení a pro čtení odpovědi. Poté, vyžaduje-li typ requestu ID relace, je toto ID přidáno do hlavičky requestu. Dále se do těla requestu zapíší jeho parametry, pokud nějaké jsou. Request se odešle voláním `connection.getResponseCode()`, které vrací HTTP kód odpovědi. Jelikož předpokládáme bezchybné vykonání, je následně odpověď v podobě XML přečtena a předána parseru, který ji rozparsuje a vytvoří instanci třídy `Action`. Tato akce je vrácena zpět přes `RequestExecutor` až do třídy `AsyncTask`, která callbackem informuje prezentéra, že request skončil. Všichni prezentéři, kteří vyžadují interakci s DCIx, totiž implementují rozhraní, které z nich udělá posluchače průběhu daného requestu.

6.6.2 Řízení chyb

Řízení chyb je v aplikaci řešeno na dvou místech podle úrovně chyby. Prvním místem je přímo `Connector`. Tyto chyby nejčastěji souvisí s připojením a přenosem dat mezi DCIx a jsou následující:

- **chyby připojení**: veškeré chyby související s připojením, například nedostupnost DCIx a/nebo překročení časových limitů;
- **vypršení relace**: jakmile na DCIx buď vyprší, nebo je ukončena relace daného uživatele, skončí request s HTTP kódem 302 (přesměrování). Cíl přesměrování je přečten z hlavičky odpovědi a pokud se jedná o URL vedoucí na přihlašovací stránku DCIx, je i v aplikaci zobrazena přihlašovací aktivita a všechny ostatní ukončeny;
- **chyby parsování**: `Connector` k parsování přijaté odpovědi používá `WSXmlParser`, který může také vrátit chybu při parsování odpovědi;

- **neřízené chyby v DCIx**: v DCIx mohou nastat neřízené chyby. V takovém případě končí request s HTTP kódem 500 a výš. Nicméně i přesto lze získat odpověď, konkrétně text chyby a výpis zásobníku. V aplikaci je pak chyba včetně výpisu zásobníku zobrazena a uživatel vrácen na přihlašovací obrazovku (definováno zadavatelem).

- **řízené chyby v DCIx**: řízenou chybou v DCIx je taková chyba, která nereprezentuje chybu ve vykonávání nějakého kódu. Jedná se o chyby typu nevyplněné povinné pole atd. Tato kategorie chyb tedy jediná nevyžaduje speciální podporu ve třídě `Connector`.

Ve všech uvedených případech se vytvoří akce s příslušnou chybou (a jejím kódem) a propaguje se zpět do aplikační logiky, konkrétně prezentéra. Prezentér rozlišuje všechny neřízené chyby a podle toho dává příkazy svému View, aby zobrazilo příslušný dialog. Je-li nějaká chyba řízená, pokračuje se ve zpracování dané transakční obrazovky dál. Tato chyba se nijak nezpracovává a je zobrazena v transakční obrazovce. Na řízené chyby aplikace reaguje přehráním zvukového efektu chyby.

6.7 Nastavení aplikace

Nastavení celé aplikace se provádí v aktivitě `PreferencesActivity`, konkrétně v jejím (jediném) fragmentu `TransactionFragment`. Aktivita obsahuje 3 kategorie nastavení:

- **připojení k DCIx**: umožňuje nastavit IP adresu, port, kontext DCIx serveru a časový limit připojení. Kategorie ještě umožňuje skenovat nastavení připojení, o tom ale samostatně v sekci 6.7.1,

- **vzhled aplikace**: současně obsahuje pouze nastavení velikosti písma,

- **ostatní**: kategorie obsahuje nastavení, která nejsou typická pro žádnou jinou kategorii. V současnosti kategorie obsahuje pouze možnost nastavení počtu pípnutí při řízené chybě.

Kategorie a jejich jednotlivé položky se nastavují v XML souboru na cestě `res/values/xml/preferences.xml`. Tato konfigurace popisuje jednotlivé položky od titulku, přes datový typ až po výchozí hodnotu.

Standardními typy jednotlivých položek jsou `EditTextPreference` nebo `ListPreference` a jejich společným předkem je třída `DialogPreference`. V prvním případě je vstupem nějaký text, který je uložen ve formátu `String`. `ListPreference` zobrazuje seznam, buď jednovýběrový, nebo vícevýběrový.

Pokud jsou nabízené typy nedostačující nebo je třeba jejich chování mírně modifikovat, lze danou třídu děděním rozšířit, následovně:

- **`IntegerEditTextPreference`** umožňuje ukládat pouze celá čísla, jelikož standardní `EditTextPreference` ukládá řetězce. Tato třída dědí od `EditTextPreference` a přepisuje některé její metody tak, aby ukládaná hodnota byla ve formátu celého čísla.

- **`FontSizeListPreference`** reprezentuje dialog s výběrem jedné velikosti písma. Protože se jedná o výběr z hodnot, dědí tato třída od `ListPreference`. Výstupem dialogu a současně uloženou hodnotou je index volby písma (0 - nejmenší, 1 - střední, 2 - největší);

- **`ErrorBeepCountPreference`**: zobrazuje speciální jezdec (`SeekBar`), kterým je možno změnit počet pípnutí při řízené chybě. Tato třída je ukázkou dialogu, který má kompletně uživatelsky definovaný layout. Třída dědí od obecného dialogu pro nastavení, `DialogPreference`.

6.7.1 Skenování informací pro připojení k DCIx

Připojovací údaje k DCIx se konfigurují v `PreferencesActivity`. Jednotlivé parametry připojení lze měnit buď manuálně, nebo je lze skenovat z QR kódu. Tento QR kód je umístěný na DCIx, konkrétně na stránce *O DCIx* (angl. *About*). Formát QR kódu je `server=<IP>;port=<port>;context=<kontext>`. Generování QR kódu bylo také součástí této práce.

Skenování QR kódu je umožněno speciální položkou v aktivitě. Po kliknutí se rozlišuje, jestli se jedná o podnikové zařízení se skenerem nebo o mobilní telefon s kamerou. V prvním případě se uživateli zobrazí dialog s jedním polem pro skenování. Po naskenování kódu se dialog potvrdí a parametry se z kódu rozparsují a uloží. V případě mobilního telefonu se opět spustí aplikace Barcode Scanner, která kód naskenuje.

Spuštění aplikace Barcode Scanner se provádí vytvořením intentu s akcí `"com.google.zxing.client.android.SCAN"`, která identifikuje skenovací aktivitu v Barcode Scanneru. Tato skenovací aktivita se spustí voláním `startActivityForResult(<intent>, <kód requestu>)` (viz 6.6). Prvním parametrem je zmiňovaný intent a druhým je kód requestu, viz dále.

```
1 @Override
2 public void startBarcodeScannerApplication() {
3    if (/* camera installed */) {
4       String action = "com.google.zxing.client.android.SCAN";
5       Intent intent = new Intent(action);
```

```
6         startActivityForResult(intent, SCAN_REQUEST_CODE);
7      } else {
8         // error shown
9      }
10  }
```

Kód 6.6: Ukázka spuštění skenovací aktivity z Barcode Scanneru

Po spuštění a naskenování kódu se vrací řízení zpět této aplikaci voláním metody `onActivityResult()`, která má 3 parametry (viz 6.7). Prvním je kód requestu, se kterým byl Barcode Scanner spuštěn a díky kterému lze identifikovat spouštěnou aktivitu (kterých může být více). Druhým parametrem je kód výsledku operace, jestli se spuštění a ukončení cizí aktivity provedlo v pořádku. Třetím parametrem je intent, který obsahuje naskenovaná data. Tento intent vytvoří právě aktivita Barcode Scanneru, vyplní naskenovaná data a odešle je volající aktivitě.

```
1  @Override
2  public void onActivityResult(int requestCode, int resultCode,
       Intent intent) {
3     super.onActivityResult(requestCode, resultCode, intent);
4     if (requestCode == SCAN_REQUEST_CODE) {
5        if (resultCode == Activity.RESULT_OK) {
6           String qrContent =
                intent.getStringExtra("SCAN_RESULT");
7           mPresenter.parseAndSaveDciConfig(qrContent);
8        } else if (resultCode == Activity.RESULT_CANCELED) {
9           // show error
10       }
11    }
12 }
```

Kód 6.7: Ukázka zpracování skenování QR kódu aplikací Barcode Scanner

6.8 Provádění transakcí

Tato sekce popisuje, jak je implementováno provádění samotných transakcí.

6.8.1 Načtení seznamu transakcí

Odpovědí od DCIx na úspěšné přihlášení je ID relace přihlášeného uživatele a seznam transakcí. Transakcí se v datovém modelu rozumí instance

třídy `Hint`, která v kontextu transakční obrazovky znamená také nápovědu pro uživatele. Informace o přihlášeném uživateli se udržují ve třídě `TransactionContainer`, která patří do komponenty `UserComponent`. Pro rekapitulaci, tato komponenta je vytvářena v metodě `onResume()` přihlašovací aktivity, což znamená, že pokaždé, když se přihlašovací aktivita stane aktivní, dojde k opětovnému vytvoření uživatelské komponenty. Tím současně dojde k vytvoření nové instance `TransactionContainer` a tím se znehodnotí veškeré údaje předchozího uživatele.

Jelikož přihlašovací aktivita vytváří `UserComponent`, má tedy i přístup k jejím explicitně vystaveným závislostem, které jsou dvě a slouží právě k těmto účelům. První je `TransactionContainer` a druhá `LogonInteractor`. Důvod, proč si komponenta přihlašovací aktivity sama nedrží instanci přihlašovacího interaktoru je opět ten, že v případě změny připojovacích údajů by byla použita předchozí instance se starými a již neplatnými údaji. Po úspěšném přihlášení je instanci `TransactionContainer` předán uživatel a seznam transakcí. Prezentér následně zajistí, aby přihlašovací aktivita spustila aktivitu `TransactionListActivity`.

Komponenta aktivity se seznamem transakcí závisí na uživatelské komponentě a tudíž může využít všechny její závislosti. V metodě `onCreate()` aktivity je zavolán její prezentér, aby zajistil zobrazení transakcí v aktivitě. Ten k tomu použije právě instanci `TransactionContainer`, ze které získá seznam transakcí. Pokud je seznam prázdný, je o tom uživatel příslušně informován. V opačném případě jsou transakce zobrazeny.

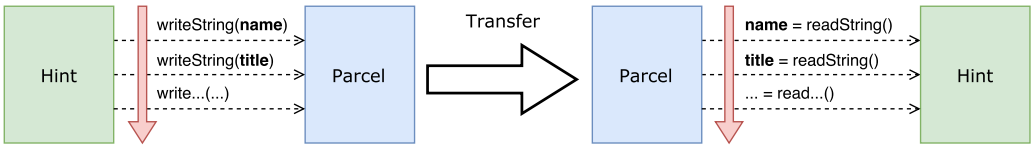
6.8.2 Spuštění transakce

Spuštění transakce iniciuje uživatel kliknutím na zvolenou transakci. V tu chvíli se ze seznamu transakcí podle pozice kliknutí vybere zvolená transakce a vloží se do intentu, kterým se následně spustí `TransactionActivity`.

Serializace

Intenty mohou standardně obsahovat primitivní typy. K přenosu nějakého objektu neprimitivního typu slouží rozhraní `Parcelable` (alternativa rozhraní `Serializable`). Oboje slouží k serializaci objektů, nicméně rozhraní `Parcelable` je specifické pro platformu Android.

Objekty tříd implementující rozhraní `Parcelable` je možné serializovat a předávat např. pomocí intentu. Serializace funguje na principu zapisování do *balíku* (`Parcel`) a deserializace čtení z balíku. Nezbytné je, aby čtení z balíku probíhalo ve stejném pořadí jako zápis do balíku (viz obr. 6.8).



Obrázek 6.8: Princip serializace pomocí `Parcelable`

Serializovaná transakce se tedy intentem přenese z aktivity se seznamem transakcí do `TransactionActivity`.

Vykonání requestu

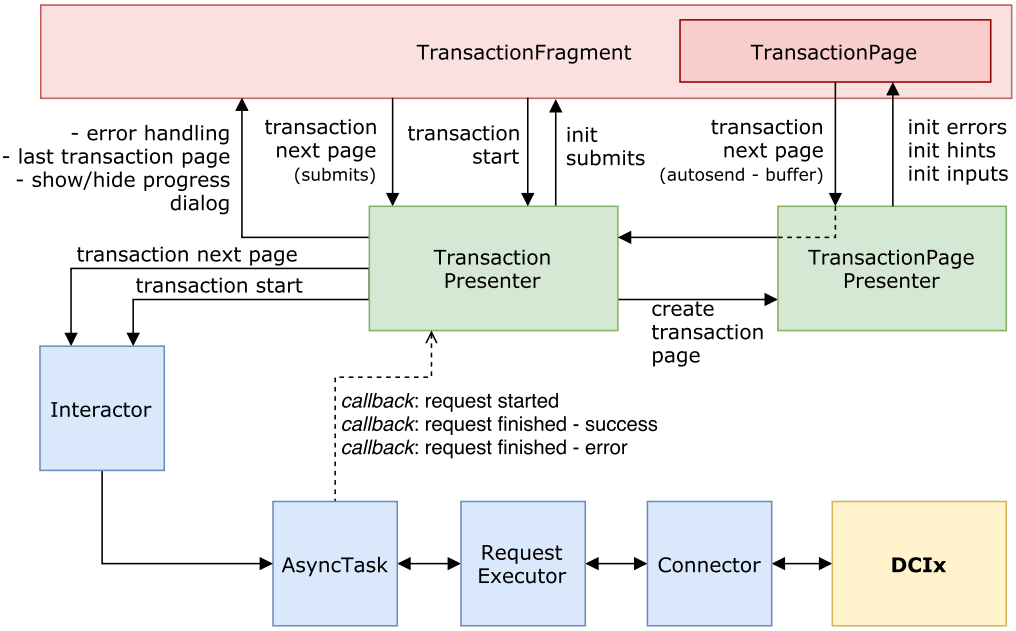
Requesty na server (start nebo další krok transakce) spouští přes svého prezentéra `TransactionFragment`. Aby mohl fragment transakci spustit a získat první transakční obrazovku, musí nejprve nějak získat ID transakce, kterou chce uživatel spustit. Jak uvádí předchozí sekce, zvolená transakce je předána intentem z `TransactionListActivity` do `TransactionActivity`. Transakční aktivita tedy musí transakci nějak dodat fragmentu. To se děje v metodě `onAttachFragment(Fragment)` aktivity. Tato metoda je zavolána jako callback, když se k aktivitě připojil nějaký její fragment (parametrem je fragment). Operátorem `instanceof` aktivita zjistí, jestli se jedná právě o `TransactionFragment` (protože jich může být více) a pokud ano, získá transakci z intentu, který jej spustil a vloží jej do argumentů fragmentu.

Nyní zbývá spouštěnou transakci získat z argumentů fragmentu a poté odeslat request na DCIx. Získání spouštěné transakce se děje v `onCreate()` fragmentu. Do atributu fragmentu se z jeho argumentů získá spouštěná transakce a v metodě `onActivityCreated()`, která indikuje, že hostující aktivita byla kompletně vytvořena, se už konečně zavolá prezentér fragmentu, který request odešle.

6.8.3 Průchod transakcí

Odpovědí na request, který spouští transakci, je první transakční obrazovka. Princip načítání transakčních obrazovek je vysvětlen na obrázku 6.9.

Spouštění transakce a odeslání transakční obrazovky má společnou část označenou modře. Zřejmá je interakce transakčního prezentéra se třídou `AsyncTask`, která ho pomocí callback metod informuje o průběhu a výsledku requestu. Pokud proběhl request v pořádku a je získána akce, tedy další transakční obrazovka, je přeposlána prezenetéru transakční obrazovky. Ten zajistí zobrazení všech řízených chyb, nápověd a vstupů. Naopak transakční prezentér zobrazí tlačítka a vyplní dodatečné menu v horní liště aktivity,



Obrázek 6.9: Princip načítání transakčních obrazovek

neboť on je ten, kdo interaguje s uživatelem při odesílání transakční obrazovky.

Pokud request skončil nějakou chybou nebo se jedná o poslední akci v transakci, řeší takovou situaci sám transakční prezentér. Ten se dále stará o odeslání transakční obrazovky buď z menu, nebo jedním ze dvou tlačítek v dolní části fragmentu. Další situací, kdy lze transakční obrazovku odeslat, je stav, kdy byl například naskenován kód, který měl na konci znak `ENTER`, který slouží k odeslání obrazovky. Proto má `TransactionPagePresenter` referenci na `TransactionPresenter`, aby ho o této události informoval.

6.8.4 Skenování čárových kódů

Při průchodu transakcí uživatel skenuje čárové kódy do transakčních polí. Během skenování kódů a střídání transakčních obrazovek může nastat situace, kdy na chvíli vypadne nebo se zpomalí spojení s DCIx. Taková situace nesmí uživatele od skenování zdržovat. Nějakým způsobem je tedy potřeba řešit, jak naložit s daty, která přijdou při výměně transakčních obrazovek.

Příjem naskenovaných dat

Příjem dat od skeneru se na transakční aktivitě od ostatních aktivit liší v tom, že data jsou dodána broadcastem (viz 6.5.1). K příjmu broadcastů slouží třída `ScannerDataProvider` (dále SDP), která má dva hlavní úkoly:

přijmout broadcasty a distribuovat je po aplikaci. SDP dále definuje rozhraní `OnScannerDataListener`, které implementují třídy, které chtějí data od skeneru dostávat.

Distribuce dat po aplikaci spočívá v přihlášení posluchačů (tříd implementujících `OnScannerDataListener`) k SDP. SDP si udržuje seznam těchto posluchačů a jakmile přijdou nějaká data, SDP je všem rozešle. Z příjmu dat od SDP se lze odhlásit. K oběma akcím slouží dvě metody ze SDP: `registerListener(...)` a `unregisterListener(...)`. Příjem a distribuci dat znázorňuje následující ukázka 6.8.

```
1  @Override
2  public void onReceive(Context context, Intent intent) {
3     if (intent.getAction().equals(ACTION_SCANNER_DATA)) {
4        String data = intent.getStringExtra(EXTRA_EMDK_DW_DATA);
5        String type = intent.getStringExtra(
6           EXTRA_EMDK_DW_LABEL_TYPE);
7        boolean fromCamera = intent.getBooleanExtra(
8           EXTRA_DCIx_DATA_FROM_CAMERA, false);
9
10       ScannedData scannedData = new ScannedData(data, type,
             fromCamera);
11
12       for (OnScannerDataListener listener : mListeners) {
13          listener.onScannerData(scannedData);
14       }
15    }
16 }
```

Kód 6.8: Distribuce naskenovaných dat posluchačům

Třída `ScannedData` nese mimo dat a typu kódu také informaci o tom, odkud data pocházejí. Zdrojem dat může být skener na podnikovém zařízení nebo kamera mobilního zařízení. Vstupní pole v transakční obrazovce obsahující ikonu čárového kódu znamenají, že data se mají naskenovat. Je-li zařízením podnikový skener, může se rovnou skenovat. Je-li zařízením např. mobilní telefon, je nutné klepnout na ikonu čárového kódu. Ta spustí Barcode Scanner jako v předchozích případech, nejedná-li se o zařízení s EMDK. Výsledek skenování se vrací do `TransactionActivity`.

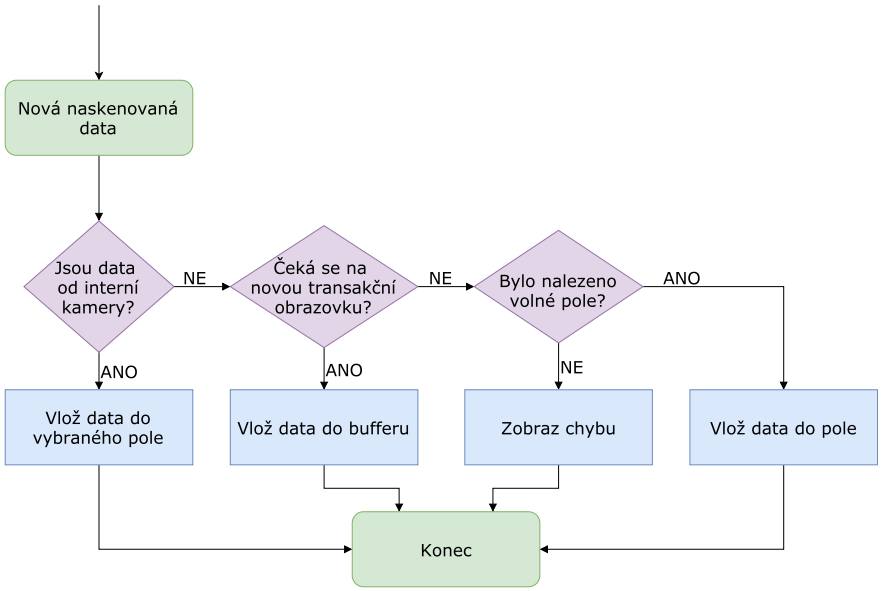
V transakční aktivitě se z přijatého intentu vytvoří nový, který má stejnou akci jako intenty od DataWedge. Tento intent je následně rozeslán broadcastem úplně stejně, jako kdyby jej rozeslal DataWedge. SDP tedy vůbec neví o tom, že data přicházejí ze dvou zdrojů. Výhodou je to, že SDP je

jediné místo, které přijímá naskenovaná data, ať už jsou odkudkoliv.

K odlišení dat od skeneru a od kamery mobilního telefonu se používá `boolean` flag v intentu s klíčem `EXTRA_DCIx_DATA_FROM_CAMERA`. Tento flag je do intentu přidán v `onActivityResult()`, tedy ve chvíli přijetí dat od kamery. V případě dat od skeneru flag v intentu není. To nevadí, neboť při získávání hodnoty tohoto flagu lze specifikovat výchozí hodnotu, která se použije, pokud intent flag neobsahuje. Výchozí hodnota je `false`, to znamená, že pokud není explicitně stanoveno jinak (viz `onActivityResult()`), považuje se za zdroj dat skener podnikového zařízení.

Skenovací buffer

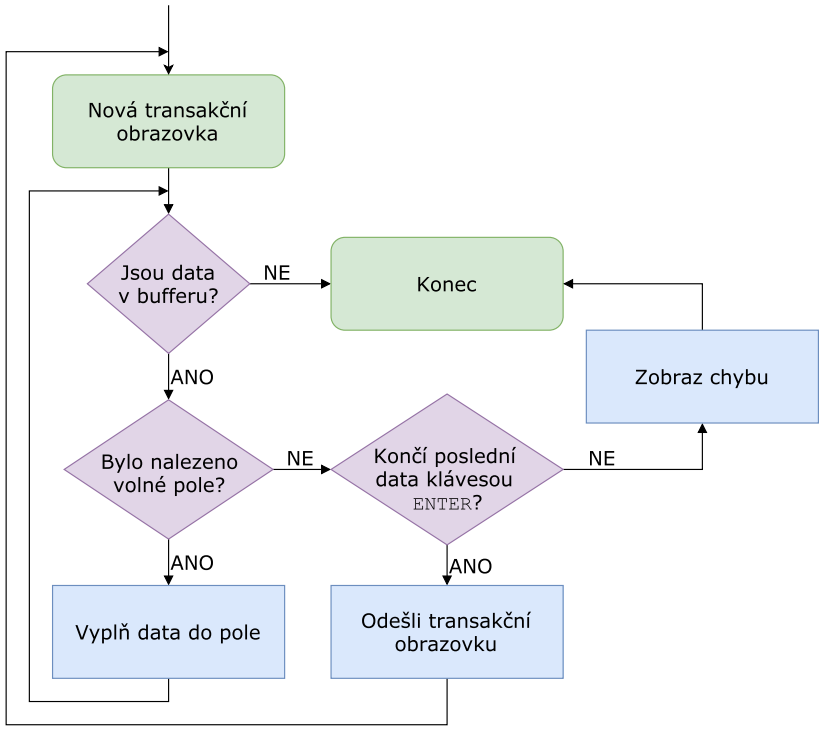
Skenovací buffer slouží k bufferování naskenovaných dat při výměně transakčních obrazovek. Buffer je implementován v prezentéru transakční obrazovky (`TransactionPagePresenterImpl`), který současně implementuje `OnScannerDataListener` a je registrovaný u SDP k odběru dat od skeneru.



Obrázek 6.10: Diagram zpracování dat od skeneru

Prezentér si drží informaci o tom, jestli zrovna probíhá výměna transakčních obrazovek. Touto informací je pouhý `boolean` flag jako atribut třídy. Jakmile je iniciována výměna transakčních obrazovek, je tento flag nastaven na `true`. Po dokončení requestu je tento flag nastaven zpět na `false`. Přijdou-li nějaká data ve chvíli, kdy je flag `true`, uloží se do bufferu. Buffer je implementován jako klasická fronta (`Queue<ScannedData>`).

Po přijetí dat se rozhoduje, co se s nimi provede. Prvním rozhodnutím



Obrázek 6.11: Diagram používání bufferu

je, jestli jsou data od kamery nebo od skeneru. Pokud jsou data od kamery, jejich skenování muselo být tedy iniciováno kliknutím na ikonu čárového kódu v poli. V této situaci jsou naskenovaná data vložena do pole nebo připojena za existující. V případě dat od skeneru nemá skener informaci o tom, na jaké pole se kliklo. Na žádné se totiž nekliká a pro data musí být nalezeno vhodné vstupní pole. Dle dohody se zadavatelem byl způsob vkládání dat implementován následovně (viz obr. 6.10).

Aplikace prochází všechna vstupní pole od prvního, dokud nenarazí na prázdné. Během hledání přeskakuje ta pole, která jsou buď zakázaná (jsou jen pro čtení), nebo nejsou prázdná. Jakmile se najde prázdné pole, data se do něj vloží. Pokud se žádné takové nenajde, data jsou zahozena a uživatel je o tom informován. Pokud nastala situace, že nalezené vstupní pole je poslední v transakční obrazovce a data jsou zároveň ukončena znakem `ENTER`, jsou data do pole vložena a transakční obrazovka automaticky odeslána.

Po přijetí další transakční obrazovky aplikace zkontroluje buffer, jestli v něm nejsou nějaká data (obr 6.11). Pokud ano, aplikace se opět pokusí nalézt volné vstupní pole a data do pole vložit. Tento proces se opakuje buď do doby, kdy byl buffer dat vyprázdněn, nebo do doby, kdy byla všechna vstupní pole naplněna. Pokud byla poslední vkládaná data ukončena klávesou `ENTER`,

je transakční obrazovka stejně jako v předchozím případě odeslána.

Oba dva tyto procesy fungují současně a neustále se opakují.

6.9 Logování

K logování byl na základě analýzy (5.4) a domluvy se zadavatelem použit log4j z těchto důvodů:

- stejný logovací framework jako v DCIx,
- stejná konfigurace jako v případě DCIx, takže osoby, které znají logy v DCIx, se vyznají v logách aplikace,
- programová konfigurace umožňuje získat cestu k externímu úložišti (na rozdíl od XML konfigurace, kde musí být cesta uvedena napevno).

K použití log4j se musí do projektu přidat 2 knihovny:

- **log4j-1.2.15.jar**: klasická log4j knihovna,
- **android-logging-log4j-1.0.3**: knihovna, která závisí na log4j a umožňuje napojit logcat na logovací záznamy log4j.

Významnou roli hraje třída `Log4jConfigurator`, která specifikuje a konfiguruje, jak bude logování probíhat. Framework loguje jak do souboru, tak i na logcat. V případě souborů se bude zapisovat do externího úložiště, do složky `DCIx_logs`. V ní se bude udržovat historie posledních 10 souborů s logy. Jméno souboru s logy je `DCIx.log` a v případě překročení jeho velikosti (1MB) je vytvořen nový a předchozímu je přidána přípona `.1` atd. Toto nastavení bylo zvoleno z důvodu omezené velikosti paměti cílových zařízení.

Konfigurace loggeru probíhá při spuštění aplikace ve statickém bloku třídy `DciApplication`.

7 Automatické testování

Tato kapitola se týká testování a popisuje, jakým způsobem byla aplikace testována a jak byl využit potenciál mockovacích testovacích frameworků. Při testování aplikace byly často odhaleny a opraveny chyby, které se během vývoje a ručního testování neprojevily.

K lokálním unit testům byl použit standardní JUnit framework ve verzi 4. Instrumentační testy jsou rozdělené na instrumentační unit testy a testy uživatelského rozhraní, které bylo testováno pomocí frameworku Espresso. V obou typech testů byla vyžadována podpora ze strany mockovacích frameworků. Zvoleným zástupcem se stalo Mockito.

Veškerý výběr použitých nástrojů k testování probíhal převážně na základě podpory a doporučení ze stránek s programátorskou dokumentací Androidu - `developer.android.com`. Espresso bylo zvoleno z důvodu snadného používání, jednoduchého rozšíření a podpory synchronizace (viz 5.5).

Dodržování konvencí použitých při psaní testů zvyšuje jejich přehlednost a orientaci v testovací třídě. Konvence používané v této aplikaci jsou následující:

- testovací třídy končí suffixem `Test`, případně dalším suffixem, který popisuje konkrétní testovanou situaci,
- metoda zavolaná před spuštěním všech testů má anotaci `@BeforeClass` a název `setUpClass()`,
- metoda zavolaná před spuštěním každého testu má anotaci `@Before` a název `setUp()`,
- testovací metody jsou označené anotací `@Test` a mají prefix `test`. Pokud je více testovacích metod, které testují stejný případ, akorát nějakou jinou variantu, je tato varianta označena postfixem `_varianta`,
- metoda zavolaná po dokončení všech testů má anotaci `@After` a název `afterClass()`.

7.1 Používání mock objektů

Před vytvářením samotných testů je nejprve potřeba rozumět tomu, jak se mockovací objekty vytvářejí a konfigurují. K mockování byl použit mockovací framework Mockito. Mockito se snadno používá, je přehledné a poskytuje

dostatečnou konfiguraci mock objektů. Mockito je také uváděno v programátorské dokumentaci pro Android jako framework kompatibilní s Android unit testy.

K vytvoření mock objektu stačí pouze třída, ze které se mock objekt vytvoří:

```
1 Hint hintMock = mock(Hint.class);
2 // now hintMock.getName() would return nothing
```

Spolu s mockovacími objekty umožňuje Mockito vytvářet tzv. spy objekty. Spy objekt se vytváří z **reálné instance** nějakého objektu vytvořením tzv. wrapperu, který je možné konfigurovat. Fungování spy objektů popisuje následující ukázka:

```
1 Hint hintSpy = spy(new Hint("foo", "bar"));
2 // now hintSpy.getName() would return "foo"
3 when(hintSpy.getName()).thenReturn("bar");
4 // now hintSpy.getName() would return "bar"
```

Mockovací objekty umožňují specifikovat návratovou hodnotu. V následující ukázce jsou uvedeny dva způsoby, jak lze toto provést.

```
1 // one way
2 doReturn("bar").when(hintSpy).getName();
3 // anoher way
4 when(hintSpy.getName()).thenReturn("bar");
```

Druhý způsob na rozdíl od prvního je typově hlídaný. Dalším rozdílem je to, že pokud daný mock objekt je ve skutečnosti spy objekt, tak ve druhém případě se provede volání reálného (obaleného) objektu. V prvním případě, jde-li o spy, tak se reálný objekt vůbec nezavolá.

Mockito umožňuje simulovat vyhození výjimky při zavolání nějaké metody. Následující příklad je i konkrétním případem použití této konstrukce v testech aplikace.

```
1 when(mConnectionMock.getResponseCode())
2         .thenThrow(new SocketTimeoutException());
```

Další konstrukcí používanou převážně v instrumentačních testech je konstrukce `thenAnswer(Answer)`. Rozhraní `Answer` obsahuje jedinou metodu `answer(InvocationOnMock)`, která se používá v situaci, kdy je třeba kontrolovat parametry, s jakými byla metoda mock případně spy objektu zavolána. Dalším použitím je změna návratové hodnoty nebo vůbec přidání návratové hodnoty `void` metodě:

```
1 Hint hintMock = mock(Hint.class);
2 when(hintMock.setName(anyString())).thenAnswer(new Answer() {
3    Object answer(InvocationOnMock invocation) {
4       Object[] args = invocation.getArguments();
5       // assert args here, e.g. assertEquals(args[0],"foo");
6       return "bar";
7    }
8 });
9 String fakeName = hintMock.setName("foo"); // returns "bar"
```

To vše se týkalo konfigurace mock objektu. Druhou částí použití mock objektu je ověření, že opravdu došlo k zavolání nějakých metod, dokonce i k počtu jejich volání. Následující ukázka ověřuje zavolání metod mock objektu.

```
1 verify(profileManagerMock, times(2))
2    .processProfile(anyString(), eq(SET), any(String[].class));
3
4 // now we verify no more interactions with mConnectionMock
5 verifyNoMoreInteractions(profileManagerMock);
```

V druhé části ukázky lze vidět, že Mockito umožňuje zkontrolovat, že na mock objektu nebyly zavolány žádné další metody.

7.2 Lokální unit testy

Lokální unit testy jsou takové testy, které lze spustit na JVM. Testovaný kód by měl být co nejméně platformě závislý, aby byla nutnost použití mock objektů co nejmenší. K unit testům byl použit framework JUnit 4.

Testy je pokryta celá aplikační logika. Unit testy jsou vhodné místo, kde se projeví výhoda oddělené aplikační logiky od prezentační vrstvy. Díky architektuře MVP je možné testovat aplikační logiku na JVM bez nutnosti nahrání aplikaci na zařízení. Veškeré externí závislosti, typicky View u každého prezentéra, jsou mockovány. Na každou akci prezentéra, která mohla v reálném prostředí od View přijít, jsou vytvořeny testovací případy, které se snaží pokrýt maximální možnou množinu průchodů a variant, jak může volání prezentéra skončit.

7.3 Instrumentační unit testy

Malou kategorií jsou instrumentační unit testy. Jsou to takové unit testy, které vyžadují Android framework, neboť testují kód, který je sice platformě závislý, ale nejedná se o UI test. Tyto třídy jsou pouze 3:

- **`SharedPreferencesRepositoryTest`**: testuje třídu, která umožňuje přístup k `SharedPreferences`, což je součást Android frameworku,
- **`HintTest`**: testuje serializaci a deserializaci třídy `Hint`, která implementuje `Parcelable`, které je opět součástí Android frameworku,
- **`WSXmlParserTest`**: testuje parsování XML odpovědí.

7.4 Testy uživatelského rozhraní

Testy uživatelského rozhraní je skupina instrumentačních testů, které lze spustit pouze na zařízeních s Android. Testy spouštějí celé aktivity a simulují akce uživatele. Při těchto testech se musí ověřit, že například stisk přihlašovacího tlačítka skutečně provede to, že informuje prezentéra, který následně iniciuje request na server. Jelikož se ale jedná o testy, tak by neměla být potřeba skutečného DCIx a veškeré requesty by měly být zastaveny před jejich odesláním. Tato situace je opět řešena pomocí mock objektů. Typickým případem použití mock objektu je třída `RequestExecutor`. Tato třída je mockována, aby za prvé nedošlo ke skutečnému odeslání requestu a za druhé aby na ní mohlo být ověřeno, že se request skutečně dostal např. od stisku tlačítka až k samotnému odeslání.

7.4.1 Dependency injection v testech

Dependency injection je použito ke vkládání závislostí jak do produkčního kódu, tak do testovacího kódu. Musí být zajištěno, že v případě produkční verze bude použita reálná instance třídy `RequestExecutor` a v případě testů její mock. Závislosti vkládané do tříd jsou vytvářeny v modulech (viz 5.2.3). Musí být tedy vytvořen nějaký modul, který v případě testovací verze aplikace na místo reálné instance dodá mock objekt.

Vše začíná ve třídě `DciApplication`. Tato třída obsahuje metody, které vytvářejí komponenty `AppComponent` (viz ukázka) a `UserComponent`, obě se svými moduly, tj. `AppModule` a `UserModule`.

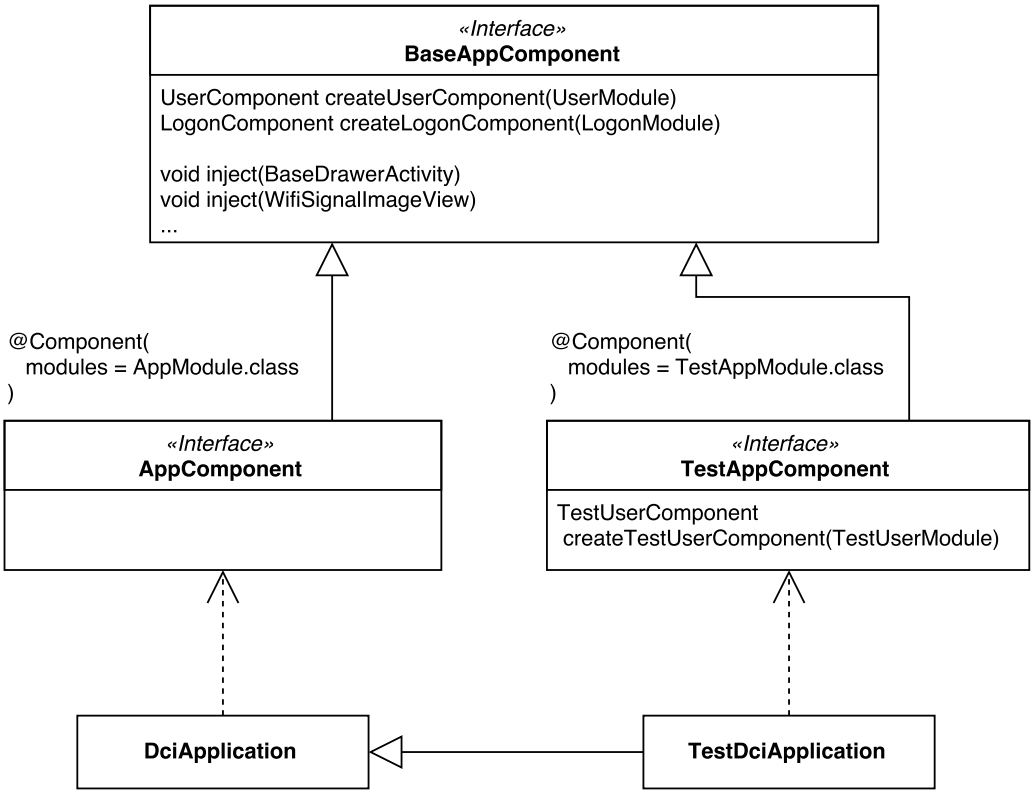
```
1 public BaseAppComponent createApplicationComponent() {
2    return DaggerAppComponent.builder()
```

```
3           .appModule(new AppModule(this))
4           .build();
5   }
```

Kód 7.1: Vytváření `AppComponent` v `DciApplication`

Tyto metody jsou právě místem, kde se musí v případě testovací verze aplikace změnit, že místo modulu s reálnými instancemi závislostí se použije modul s mock objekty.

Tyto mockované závislosti musí být injektovány do testovacích tříd, aby bylo možné, je konfigurovat. Injekce do tříd, jak je známo, se provádí v definici komponenty přidáním metody `void inject(<kam>)` (jméno `inject` je konvencí). Nyní se musí vyřešit, do jaké komponenty metodu přidat, aby se nemíchal testovací kód s produkčním.



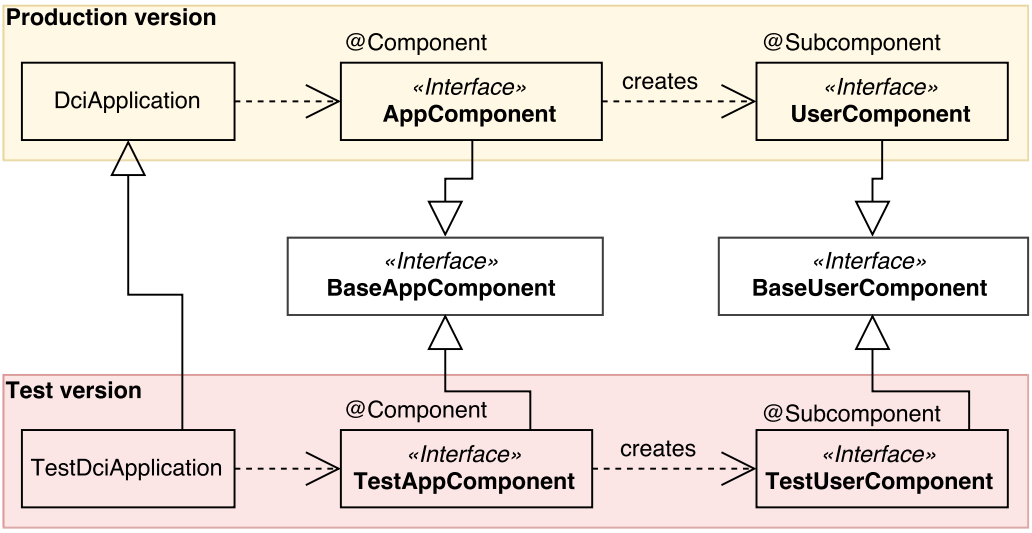
Obrázek 7.1: Diagram hlavní komponenty po úpravě pro testování

K vysvětlení řešení DI je lepší začít od začátku. V testovacích třídách je potřeba mock objektu třídy `RequestExecutor`. Ta je poskytována modulem `UserModule`. Musí být tedy vytvořen modul, který místo reálné instance vrátí mock (odtud `TestUserModule`). Aby mohl být použit testovací uživatelský modul, musí se vytvořit i testovací uživatelská komponenta, respek-

tive subkomponenta (odtud `TestUserComponent`). Jelikož je to subkomponenta, musí mít nějakou rodičovskou komponentu, která ji vytvoří (odtud `TestAppComponent`). Aby tato komponenta byla hlavní komponentou testovací verze aplikace, byla vytvořena `TestDciAppComponent`, která dědí od `DciAppComponent` a překrývá metody, které vytvářejí právě `AppComponent` (viz následující ukázka) a `UserComponent` (viz obr. 7.1).

```
1 @Override
2 public BaseAppComponent createApplicationComponent() {
3    return DaggerTestAppComponent.builder()
4       .testAppModule(new TestAppModule(this))
5       .build();
6 }
```

Kód 7.2: Vytváření `TestAppComponent` v `TestDciApplication`



Obrázek 7.2: Diagram `AppComponent` a `UserComponent` po úpravě pro testování

Aby se zbytečně neopakoval kód, byla vytvořena dvě speciální rozhraní: `BaseAppComponent` a `BaseUserComponent` (viz návratové hodnoty ukázek 7.1, 7.2 a obr. 7.2). Tato rozhraní ještě nereprezentují komponenty. Obsahují pouze hlavičky metod, které jsou společné pro produkční a testovací verzi aplikace. V produkční verzi tato rozhraní rozšiřují rozhraní `AppComponent` respektive `UserComponent` a v testovací verzi `TestAppComponent` respektive `TestUserComponent`. Tato 4 rozhraní mají již anotace `@Component` respektive `@Subcomponent` a reprezentují komponenty produkční repektive testovací verze aplikace. Na obrázku 7.1 si lze všimnout, že `TestAppComponent`

dostává závislosti z `TestAppModule`, který obsahuje mock verze objektů, jsou-li třeba. Stejně funguje i `TestUserComponent`, která závislosti získává z `TestUserModule`, který poskytuje například mock `RequestExecutor`. Jelikož tyto testovací komponenty používají moduly, které kompletně nahrazují moduly v produkční verzi, musí i tyto testovací moduly poskytovat všechny závislosti jako v produkční verzi, jinak bude Dagger hlásit chybu.

`TestUserComponent` potom dále obsahuje metody pro injektování závislostí do testovacích tříd. Po spuštění testů bude potom stejný mock objekt použit jak v testované aplikaci, tak v testující třídě, která může ověřit, jak byl mock objekt používán.

Test Runner

Ve výše uvedených sekcích byl vysvětlen význam třídy `TestDciApplication`, která zajišťuje použití testovacích komponent v testovací verziaplikace. Musí se už jen nějak určit, aby se při testování použita právě tato třída.

Instrumentační testy jsou prováděny runnerem, který vždy celou aplikaci před testy nejprve spustí. Runner k tomu potřebuje vědět, jaká třída reprezentuje instanci aplikace. V produkční verzi je to `DciApplication`, protože je uvedena v manifestu aplikace. Pokud se má použít jiná hlavní třída, existuje řešení, jak ji změnit. To spočívá ve vytvoření nového test runneru (odtud `MockTestRunner`), který dědí od výchozího `AndroidJUnitRunner` a přepisuje jeho metodu `Application newApplication(...)` (viz ukázka 7.3). Tato metoda zajišťuje vytvoření instance aplikační třídy, kterou je možné díky jejímu přepsání změnit.

```
1 public class MockTestRunner extends AndroidJUnitRunner {
2
3   @Override
4   public Application newApplication(ClassLoader cl, String
        className, Context context) throws
        InstantiationException, IllegalAccessException,
        ClassNotFoundException {
5     // use TestDciApplication instead
6     return super.newApplication(cl,
          TestDciApplication.class.getName(), context);
7   }
8
9 }
```

Kód 7.3: Ukázka změny hlavní aplikační třídy na `TestDciApplication`

7.4.2 Espresso

Testování uživatelského rozhraní (UI) pomocí Espressa umožňuje simulovat akce uživatele a ověřovat, zda to, co se mělo provést se provedlo. Espresso pracuje jen a pouze s UI, na kterém provádí akce. Odpovědí na akci je buď nějaká změna UI, kterou ověří opět Espresso, nebo např. simulace odeslání requestu na server, který ověří Mockito.

Tato sekce dále popisuje konkrétní případy a ukázky, jak bylo Espresso použito a jak bylo využito jeho snadné rozšiřitelnosti. Vysvětlení, co je to matcher, akce nebo aserce, je v kapitole 5.5.4.

Příprava testů

UI testy provádějí akce na aktivitách. Ke specifikaci aktivity, která se bude testovat, slouží testovací pravidlo `ActivityTestRule<T>`, kde `T` je testovaná aktivita. Instance tohoto pravidla se uloží do atributu testované třídy, který se označí anotací `@Rule`, viz následující ukázka.

```
1 @Rule
2 public ActivityTestRule<TransactionActivity> mActivityRule =
3         new ActivityTestRule<>(
4           TransactionActivity.class, // what activity
5           true, // initial touch mode
6           false); // should activity be launched?
```

Kód 7.4: Ukázka použití `ActivityTestRule` pro testování aktivity

Ve výchozím nastavení je aktivita spuštěna před každou `@Before` a `@Test` metodou a ukončena po dokončení testu a každé `@After` metodě. V ukázce 7.4 je použit způsob, kdy není aktivita spuštěna automaticky. Pokud by měla být spuštěna automaticky, stačí poslední dva parametry konstruktoru smazat.

Aktivity lze spustit manuálně voláním `launchActivity(Intent)` nad daným pravidlem. Například u `TransactionActivity` je zpožděné spuštění žádoucí ze dvou důvodů. Prvním důvodem je to, že každé spuštění této aktivity v aplikaci znamená, že se spouští nová transakce a odesílá se tedy request na server, který transakci spouští. Proto je nejprve potřeba získat mock `RequestExecutor` a nastavit na něm návratovou hodnotu pro volání metody `doTransactionStart(...)`. Druhým důvodem je to, že ve chvíli spouštění produkční verze této aktivity je uživatel už přihlášen. Před samotným spuštěním aktivity se tedy musí nastavit, že je uživatel přihlášený a až poté lze aktivitu spustit:

```
1 // user must be logged
2 mTransactionContainer.onUserLogon(createLogonAction());
3 // transaction start request must be mocked
4 doReturn(firstTransactionPageAction)
5         .when(mRequestExecutor)
6         .doTransactionStart(anyString(), any(Action.class));
7 // now launch activity
8 mActivityRule.launchActivity(intent);
```

Kód 7.5: Manuální spuštění aktivity

Některé testy díky tomu, že jsou spuštěny na reálných zařízeních, tak využívají i reálné úložiště `SharedPreferences`. Veškeré změny provedené testy jsou tedy perzistentní. Jako řešení téhle situaci byla vytvořena třída `PreferencesBackupTool`, která vytvoří kopii `SharedPreferences` a později ji zase nahraje zpět. Třída má dvě metody:

- `store(SharedPreferences)`: volána z metod s anotací `@BeforeClass`, čili před spuštěním všech testů v testovací třídě,
- `restore()`: volána z metod s anotací `@AfterClass`, čili po dokončení všech testů v testovací třídě.

Parametrem volání `store()` je reference na objekt `SharedPreferences`. Zálohovací třída si poté vytvoří kopii všech dat a referenci uloží. Volání `restore()` potom vymaže všechna data z `SharedPreferences` a nahraje uloženou kopii. To zajišťuje stejný stav zařízení po dokončení všech testů.

Testování sběru dat z UI

Chování prezentera po provedení nějaké akce je ověřeno unit testy. Tato krátká sekce popisuje testování případů, že prezentační vrstva sbírá data z jednotlivých elementů tak, jak by měla. K tomu je opět použit mock třídy `RequestExecutor` a konstrukce s `Answer` (použití `Answer` viz 7.1).

Jako příklad lze uvést situaci, kdy se pomocí Espressa odešle transakční obrazovka kliknutím na tlačítko `Continue`. Prezentační vrstva by měla *posbírat* data ze zadaných polí, vytvořit akci a předat ji prezentéru, přes kterého se dostane až k mock objektu, kde se ověří, že byla data korektně posbírána:

```
1 when(mRequestExecutorMock.doTransactionNextPage(anyString(),
    any(Action.class))).thenAnswer(new Answer<Action>() {
2       @Override
```

```
3       public Action answer(InvocationOnMock invocation) throws
            Throwable {
4           Action inputAction = (Action)
                invocation.getArguments()[1];
5           // assert submit
6           assertEquals(1, inputAction.getSubmits().size());
7           assertEquals("continue",
                inputAction.getSubmits().get(0).getName());
8           // assert inputs
9           List<Input> inputs = inputAction.getInputs();
10          assertEquals(3, inputs.size());
11          assertEquals("input1", inputs.get(0).getName());
12          // more assertions here
13          return createInputsSubmitsAction_ModuleScan();
14      }
15  }
16 );
```

Kód 7.6: Ukázka ověření sběru dat z polí transakční obrazovky

Testování vstupu na transakční obrazovce

Tato ukázka popisuje, jak lze pomocí Espressa otestovat správnou konfiguraci vstupního pole na transakční obrazovce.

Mock třídy `RequestExecutor` je nakonfigurován tak, aby po zavolání `doStartRequest(...)` vrátil akci, která bude mít 3 vstupy. V ukázce je popsáno otestování pouze prvního, jinak by byla příliš dlouhá. Vstupy jsou zobrazeny v kontejneru s ID `frg_transaction_tp_input_container`. První vstup má titulek „Input 1“ a hodnotu „Input's 1 value“ (kontejner pro vstup má 1 `TextView` pro titulek a 1 `EditText` pro hodnotu). Vstup je pouze pro čtení, musí být tedy zakázaný a nesmí mít focus. Jedná se o modul `scan`, takže musí mít ikonu čárového kódu a jelikož je jen pro čtení, tak ještě se zámkem. Vstup už má hodnotu, takže musí být vidět tlačítko pro vymazání obsahu i přes to, že je zakázané (layout aktivity viz 6.2.1).

```
1 doReturn(createInputsAction_ModuleScan())
2     .when(mRequestExecutor).doTransactionStart(anyString(),
          any(Action.class));
3 mActivityRule.launchActivity(getStartIntent());
4
5 // there has to be 3 TransactionInputContainers
```

```
6  onView(withId(R.id.frg_transaction_tp_input_container))
7        .check(matches(isDisplayed()))
8        .check(hasChildrenOfType(3,
           TransactionInputContainer.class));
9
10 onView(withChildAtIndexByParent(
11       0, // on view that is a first child within this parent:
12       withId(R.id.frg_transaction_tp_input_container)))
13   .check(matches(allOf( // check that
14        // the view's first child contains text "Input 1" AND
15        withChildAtIndex(0, withText("Input 1")),
16        // and the view's second child:
17        withChildAtIndex(1,
18            allOf(
19                // contains text "Input's 1 value" AND
20                withText("Input's 1 value"),
21                not(isEnabled()), // is not enabled AND
22                not(hasFocus()), // doesn't have focus AND
23                // has locked scan left ico AND
24                withLeftDrawable(R.drawable.ic_scan_lock_24dp),
25                // has clear button right ico
26                withRightDrawable(R.drawable.ic_clear_14dp)
27            )
28       ))
29   ));
```

Kód 7.7: Ukázka testu správné konfigurace vstupu na transakční obrazovce

Ukázka obsahuje 3 významné metody:

- `hasChildrenOfType(X, Y)`: aserce, která ověřuje, že view má přesně `X` potomků třídy `Y`,
- `withChildAtIndexByParent(X, Y)`: matcher, který vrací view, které je `X`-tým potomkem rodiče, který vrací matcher `Y`,
- `withChildAtIndex(X, Y)`: matcher, který vrací view, které je `X`-tým potomkem nalezeného view a současně odpovídá matcheru `Y`. Nalezeným view je myšleno view, které nalezl matcher v `onView()`.

Matcher de facto view nevrací, matcher dostane na vstupu nějaké view a porovnává se s ním, jestli mu odpovídá. Výstupem matcheru tedy není

view, ale `true` nebo `false`. Pokud se tedy řekne, že matcher vrací view, je to ve skutečnosti tak, že bylo nalezeno view, které tomu matcheru odpovídá.

Ukázka metody matcheru `withChildAtIndexByParent(childPosition, parentMatcher)`, která provádí samotné matchování:

```
1  @Override
2  public boolean matchesSafely(View view) {
3     if (!(view.getParent() instanceof ViewGroup)) {
4        return false;
5     }
6
7     ViewGroup parent = (ViewGroup) view.getParent();
8     return parentMatcher.matches(parent) &&
          parent.getChildAt(childPosition).equals(view);
9     }
10 }
```

Všechny vytvořené vlastní akce, aserce a matchery, jsou obsaženy v příslušných třídách v balíku `cz.aimtec.dci.android.espresso`.

Testování intentů

Espresso poskytuje nástroje k testování intentů. K tomu namísto pravidla `ActivityTestRule` slouží pravidlo `IntentsTestRule`. K testování intentů se používají dvě metody:

- `intending(Matcher)`: slouží k informování Espressa, že bude vyvolán intent, který bude odpovídat intent matcheru v parametru. Espresso umožňuje na intent odpovědět jiným skrze `onActivityResult()` voláním `intending(Matcher).respondWith(activityResult)`;
- `intended(Matcher)`: umožňuje ověřit, že intent odpovídající matcheru v parametru byl skutečně vyvolán.

Testování intentů je použito například v situaci, kdy je třeba otestovat skenování QR kódu pomocí vestavěné kamery zařízení (viz 6.7.1). Aby testy nebyly závislé na aplikaci Barcode Scanner, je použita konstrukce `intending(...)`, která funguje podobně jako `when(...)` u Mockita, tzn. že odchytí volání intentu, nepustí jej dál a odpoví tak, jak bylo nastaveno:

```
1 ActivityResult actRes = /* create activity result */;
2 // tell espresso to expect an intent to Barcode Scanner
3 // tell espresso to respond to that intent with ActivityResult
```

```
4  intending(hasAction(ACTION_ZXING_SCAN)).respondWith(actRes);
5  // click on preference that should open Barcode Scanner
6  onData(withKey(preferenceKey)) // preference's key
7     // in what adapter view (another one: navigation drawer)
8     .inAdapterView(withId(android.R.id.list))
9     .check(matches(isDisplayed())) // pref must be displayed
10    .perform(click()); // CLICK!
11 // capture intent
12 intended(hasAction(ACTION_ZXING_SCAN));
```

Testování stavu sítě

Tato ukázka popisuje, jak je testována situace, kdy se ztratí připojení k síti. Testuje se, jestli na to aplikace zareagovala a jak. Účelem ukázky je spíše předvedení, jak lze vytvořit vlastní akci pomocí Espressa. Testovacím případem je nedostupné připojení.

```
1  @Test
2  public void testWifiDisabled() throws Exception {
3     if (isWifiEnabled()) {
4        // disable wifi if enabled
5        sWifiManager.setWifiEnabled(false);
6        // wait for wifi disable state
7        onView(withId(act_base_iv_wifi_signal_indicator))
8           .perform(waitForWifiState(WIFI_STATE_DISABLED));
9     }
10    // verify that signal indicator shows disabled wifi icon
11    onView(withId(act_base_iv_wifi_signal_indicator))
12       .check(matches(
13          withDrawableOnImageView(ic_wifi_disabled_24dp)));
14    // verify that No connection bottom bar is displayed
15    onView(withId(R.id.act_base_tv_no_connection))
16       .check(matches(isDisplayed()));
17    // click on signal indicator
18    onView(withId(act_base_iv_wifi_signal_indicator))
19       .perform(click());
20    // verify that dialog that enables Wi-Fi was displayed
21    onView(withText(act_base_enable_wifi))
22       .check(matches(isDisplayed()));
23 }
```

Důležitou roli v testu hraje akce `waitForWifiState(...)`, která realizuje čekání na vypnutou Wi-Fi. Jedná se o instanci rozhraní `ViewAction()`, jejíž hlavní metoda `perform(...)`, která akci provádí, vypadá následovně:

```
1  @Override
2  public void perform(UiController uiController, View view) {
3     WifiManager wifiMngr = (WifiManager)
          view.getContext().getSystemService(Context.WIFI_SERVICE);
4     if (wifiMngr.getWifiState() == wifiState) {
5        return;
6     }
7     while (true) {
8        if (wifiMngr.getWifiState() == wifiState) {
9           return;
10       }
11       // simulating pause for 50ms
12       uiController.loopMainThreadForAtLeast(50);
13    }
14 }
```

Testování kliknutí na drawable

Ukázka testuje kliknutí na pravé drawable pole `ClearableEditText`, které by mělo vymazat jeho obsah. Jelikož drawable není tlačítko ani view, na které by se mohla uplatnit klasická akce `click()`, musí se vytvořit speciální akce, která kliknutí nasimuluje. Vstupem této akce je pouze `DrawablePosition`.

```
1  @Override
2  public void perform(UiController uiController, View view) {
3    float[] precision = Press.FINGER.describePrecision();
4    switch(drawablePosition) {
5      case RIGHT: coords = GeneralLocation.CENTER_RIGHT
6           .calculateCoordinates(view); break;
7      case LEFT: coords = GeneralLocation.CENTER_LEFT
8           .calculateCoordinates(view); break;
9      default: throw new AssertionError(
10          "Drawable position not implemented.");
11   }
12   // send DOWN event
13   MotionEvent down = MotionEvents.sendDown(uiController,
        coords, precision).down;
```

```
14    // send UP event
15    MotionEvents.sendUp(uiController, down, coords);
16  }
```

Princip spočívá v tom, že podle pozice drawable (zatím jen buď levá, nebo pravá) se pomocí třídy `GeneralLocation` získají souřadnice, na kterých leží dané drawable. Na tyto souřadnice se poté odešle událost `DOWN`, která reprezentuje dotyk prstu na displeji a hned poté `UP`, která reprezentuje zvednutí prstu.

7.4.3 Shrnutí testů

Tato aplikace byla otestována dvěma typy testů: lokálními unit testy a instrumentačními testy převážně uživatelského rozhraní. Unit testy byly použity k testování platformě nezávislého kódu, kterým je díky architektuře MVP především aplikační logika. Unit testy pokryly všechny testovatelné třídy a jejich metody. Snahou bylo otestovat i co největší množství různých variant testovacích případů.

Instrumentační testy byly použity k testování platformě závislého kódu, především uživatelského rozhraní. Díky mock objektům lze simulovat odpověď od DCIx a lze tak testovat veškeré scénáře, které mohou při reálném používání aplikace nastat (např. výpadek spojení, vypršení relace atd.). Instrumentační testy simulují chování reálného uživatele a testují velké množství akcí, které může uživatel provést.

8 Závěr

Hlavním cílem této práce bylo vytvořit mobilní aplikaci pro platformu Android, která bude komunikovat s informačním systémem DCIx. DCIx je produkt společnosti Aimtec a. s. poskytující řešení pro logistické, distribuční a výrobní společnosti. Aplikace je cílená na koncová podniková zařízení používaná nejčastěji ve skladech a skladových prostorách.

První část diplomové práce se týká především DCIx a platformy Android. Jsou vysvětleny základní principy a způsoby komunikace s DCIx. Následně je stručně představena platforma Android, zejména její části, které byly v této práci použity. Druhá část diplomové práce je zaměřena více na vývoj cílové aplikace. Byly analyzovány a srovnány různé architektury a návrhové vzory použitelné na této platformě. Součástí druhé kapitoly je také rozebrání možností, jak testovat aplikace pro Android. Následuje kapitola týkající se samotné implementace. Ta obsahuje popis, jak byla aplikace implementována, jaké architektonické prvky byly použity a proč. Další významnou kapitolou této práce je testování. Tato kapitola popisuje způsoby, jak byla aplikace testována a jak testování přispělo ke zvýšení kvality aplikace.

Architektura aplikace byla zvolena a implementována tak, že zvyšuje přehlednost a udržovatelnost kódu. Toho je docíleno především díky čistému oddělení prezentační, aplikační a datové vrstvy. Díky dependency injection je správa a distribuce závislostí jednoduchá. Dependency injection je také velmi nápomocné v případě testů uživatelského rozhraní. Platformě nezávislá aplikační logika umožňuje rychlé testování jednotkovými testy na JVM. Datová vrstva je testována za pomoci mockovacích objektů, které zbavují testy externích závislostí. Uživatelské rozhraní je testováno moderním testovacím frameworkem a poskytuje kontrolu, že se prezentační vrstva chová tak, jak by měla.

Základními požadavky na aplikaci byly snadná rozšiřitelnost, přehledné a jednoduché uživatelské rozhraní a celkově snadné používání aplikace nejen na podnikových zařízeních. Všechny tyto a další body zadání byly splněny. Aplikace se umí přihlašovat k DCIx, umí načíst a zobrazit seznam transakcí a nějakou transakci spustit. Následně aplikace umožňuje skenovat a vkládat data z integrovaného skeneru i z vestavěné kamery a navigovat mezi transakčními obrazovkami. Součástí aplikace je také nastavení, které umožňuje konfigurovat například připojovací údaje k DCIx. Aplikace byla otestována jednotkovými lokálními testy a instrumentačními testy převážně uživatelského rozhraní. Aplikace funguje a její chování bylo ověřeno a otestováno

i na reálném podnikovém zařízení.

Obtížnou částí při vývoji aplikace byla především integrace frameworku Dagger 2 použitého pro dependency injection. Pochopení vnitřní stavby závislostního grafu, poskytování závislostí a samotná závislost komponent, to vše bylo při vývoji velkou výzvou, kterou se i přesto podařilo splnit. Dependency injection spolu s vrstvenou architekturou pomohly k lepšímu pochopení, jak celkově strukturovat a produkovat lepší a přehlednější kód.

Diplomová práce celkově přinesla mnoho nových poznatků a zkušeností týkajících se především vývoje a testování aplikací pro Android. Veškeré tyto přínosy budou zcela jistě uplatnitelné kdykoliv později.

8.1 Další rozšíření

Cílové prostředí, ve kterém se bude aplikace používat, je typické pro další rozšíření. Z důvodu vysoké poptávky po mobilních zařízeních se dá očekávat, že budou narůstat případy využití aplikace k přístupu k DCIx. Co se týká možných vylepšení, tak lze vyjmenovat následující:

- **mód kiosek + heslo**: mód aplikace již představený v analýze v sekci 5.3. V kiosk módu není možné převést aplikaci jakkoli do pozadí. Uživatel by měl možnost po zadání hesla nebo po přihlášení jako administrátor kiosk mód vypnout,
- **více modulů**: v současném stavu podporuje aplikace nejpoužívanější DCIx moduly `type` a `scan`. DCIx transakce obsahují ale více modulů, které by bylo vhodné do budoucna implementovat,
- **prohlížení dat**: uživatel by měl možnost prohlížení dat, ať už jde o grafy nebo o různé tabulky,
- **tiskárny a skenery**: aplikace by pomocí Bluetooth nebo sítě měla přístup k externím skenerům a tiskárnám,
- **prohlížení dat**: aplikace by umožnila prohlížení dat ze skladu (seznamy balení, pozice, ...),
- **zprávy**: aplikace by umožnila zasílání zpráv mezi uživateli,
- **asynchronní dependency injection**: Dagger 2 lze rozšířit o Dagger Producers, což je modul, který podporuje asynchronní vkládání závislostí.

Přehled zkratek

ADT	Android Development Tools
AOT	Ahead of Time (compilation)
API	Application Programming Interface
APS	Advanced Planning and Scheduling
ART	Android Runtime
AS	Android Studio
CRM	Customer Relationship management
DI	Dependency Injection
DOM	Document Object Model
DVM	Dalvik Virtual Machine
EAM	Enterprise Asset Management
EDI	Electronic Data Interchange
EMDK	Enterprise Mobile Development Kit
ERP	Enterprise Resource Planning
HTTP	Hypertext Transfer Protocol
ISO	Internet Organization for Standardization
JIS	Just in Sequence
JIT	Just in Time (ERP)
JIT	Just in Time (compilation)
JSR	Java Specification Request
JVM	Java Virtual Machine
MES	Manufacturing Execution System
MIME	Multipurpose Internet Mail Extensions
MOM	Manufacturing Operations Management
MRP	Material Requirements Planning
MRP	Manufacturing Resource Planning
MVC	Model-View-Controller
MVP	Model-View-Presenter
MVVM	Model-View-ViewModel
Mx	Mobility Extensions
QMS	Quality Management System
RFID	Radio Frequency Identification
SCM	Supply Chain management
SDP	ScannerDataProvider
SLF	Simple Logging Facade
SSL	Secure Socket Layer

TCP	Transmission Control Protocol
TSL	Testing Support Library
UDP	User Datagram Protocol
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WLAN	Wireless Local Area Network
WMS	Warehouse Management System
XML	Extensible Markup Language

Literatura

[1] *Just in Time & Just in Sequence* [online]. [cit. 22.4.2016]. Dostupné z: http://www.cie-plzen.cz/index.php/cz/lexikon-metod/just-in-time-just-in-sequence.

[2] *Co je MES - Výrobní informační systém* [online]. 2012. [cit. 21.4.2016]. Dostupné z: http://mescentrum.cz/o-projektu/co-mes.

[3] *Just-in-time* [online]. 2013. [cit. 22.4.20146]. Dostupné z: https://managementmania.com/cs/just-in-time.

[4] *Quality Management System* [online]. Department of Trand and Industry, UK. [cit. 21.4.2016]. Dostupné z: http://www.businessballs.com/dtiresources/quality_management_systems_QMS.pdf.

[5] *Just-in-Time, Just-in-Sequence* [online]. Siemens AG. Dostupné z: http://w3.siemens.com/mcms/mes/en/industry/discretemanufacturing/automotiveoem/pages/just-in-time-just-in-sequence.aspx.

[6] Basl, J. – Blažíček, R. *Podnikové informační systémy: Podnik v informační společnosti - 3., aktualizované a doplněné vydání.* Grada Publishing a.s., 2012. Dostupné z: https://books.google.cz/books?id=SqtgAgAAQBAJ. ISBN 9788024775944.

[7] Browne, J. *Production Management Systems: An Integrated Perspective.* Addison-Wesley Publisher, 1996. ISBN 0201422972.

[8] Crawford, B. *DataWedge API for Android* [online]. 2013. [cit. 4.5.2016]. Dostupné z: https://developer.zebra.com/docs/DOC-1962.

[9] Eason, J. *An update on Eclipse Android Developer Tools* [online]. 2015. [cit. 3.5.2016]. Dostupné z: http://android-developers.blogspot.cz/2015/06/an-update-on-eclipse-android-developer.html.

[10] Friedman, D. *WMS Pros and Cons* [online]. 2006. [cit. 21.4.2016]. Dostupné z: http://ewweb.com/archive/wms-pros-and-cons.

[11] Furtado, P. H. *RFID vs. Barcodes: Mistake-proofing the warehouse* [online]. 2015. [cit. 23.4.2016]. Dostupné z: http://www.logasiamag.com/2015/07/rfid-vs-barcodes-mistake-proofing-the-warehouse/.

[12] Glockner, H. et al. *Augmented Reality in Logistics* [online]. DHL Customer Solutions & Innovation, 2014. [cit. 24.4.2016]. Dostupné z: http://www.dhl.com/content/dam/downloads/g0/about_us/logistics_insights/csi_augmented_reality_report_290414.pdf.

[13] Žák, R. *Manufacturing Operations Management* [online]. 2015. [cit. 21.4.2016]. Dostupné z: http://www.systemonline.cz/rizeni-vyroby/mom-manufacturing-operations-management.htm.

[14] Kjellsdotter, L. *Use of Advanced Planning and Scheduling (APS) systems to support manufacturing planning and control processes.* PhD thesis, Chalmers University of Technology, 2012. Dostupné z: http://publications.lib.chalmers.se/records/fulltext/162616/162616.pdf.

[15] MacLean, D. – Komatineni, S. – Allen, G. *Pro Android 5.* Apress, 2015. ISBN 1430246804.

[16] Maříк, V. *Je Industry 4.0 opravdu revolucí?* [online]. 2015. [cit. 21.4.2016]. Dostupné z: http://www.stech.cz/Portals/0/Konference/2015/03%20Industry/PDF/01_marik.pdf.

[17] Morrill, D. *Announcing the Android 1.0 SDK, release 1* [online]. 2008. [cit. 2.5.2016]. Dostupné z: http://android-developers.blogspot.cz/2008/09/announcing-android-10-sdk-release-1.html.

[18] Příhoda, J. *Jak efektivně řídit sklad* [online]. 2015. [cit. 23.4.2016]. Dostupné z: http://www.systemonline.cz/it-pro-logistiku/jak-efektivne-ridit-sklad.htm.

[19] Sodomka, P. – Klčová, H. *Řízení majetku a správa kritických aktiv v ERP systémech* [online]. 2011. [cit. 22.4.2016]. Dostupné z: http://www.systemonline.cz/it-asset-management/rizeni-majetku-a-sprava-kritickych-aktiv.htm.

[20] Vollmann, T. *Manufacturing Planning and Control for Supply Chain Management.* McGraw-Hill, 2005. ISBN 0071121331.

[21] Wharton, J. *Dependency Injection with Dagger 2* [online]. 2014. [cit. 4.5.2016]. Dostupné z: https://speakerdeck.com/jakewharton/dependency-injection-with-dagger-2-devoxx-2014.

[22] Woods, V. – Meulen, R. *Gartner Says Worldwide Smartphone Sales Grew 9.7 Percent in Fourth Quarter of 2015* [online]. 2016. [cit. 3.5.2016]. Dostupné z: http://www.gartner.com/newsroom/id/3215217.

[23] Zahálka, L. – Schwob, R. *Warehouse Management* [online]. 2009. [cit. 21.4.2016]. Dostupné z: http://www.systemonline.cz/erp/warehouse-management.htm.

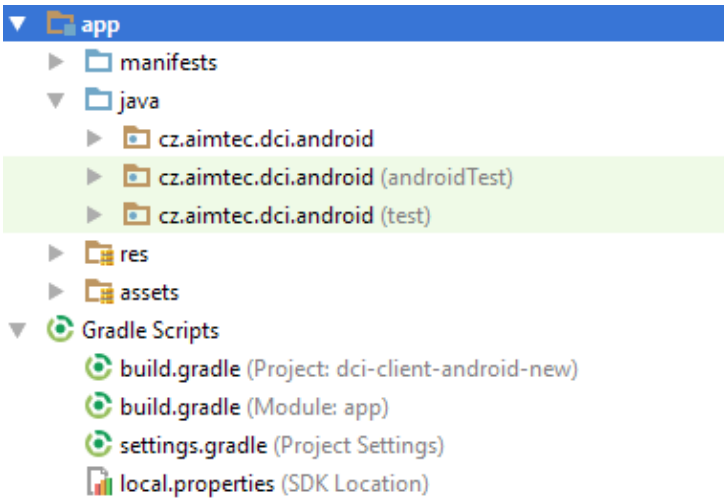
Přílohy

Uživatelská příručka

Tato uživatelská příručka popisuje, jak lze aplikaci sestavit a spustit. K vyžití hlavní funkčnosti aplikace je nezbytný DCIx server.

Import, sestavení a spuštění

Nejjednodušším způsobem, jak aplikaci sestavit a spustit, je využít služeb Android Studia. Projekt se do Androi Studia dá jednoduše naimportovat pomocí `File/New/Import project` a vybrat soubor `build.gradle` nebo `settings.gradle` z kořenového adresáře projektu. Jakmile se projekt naimportuje, měla by základní souborová struktura vypadat jako na obrázku 8.1.

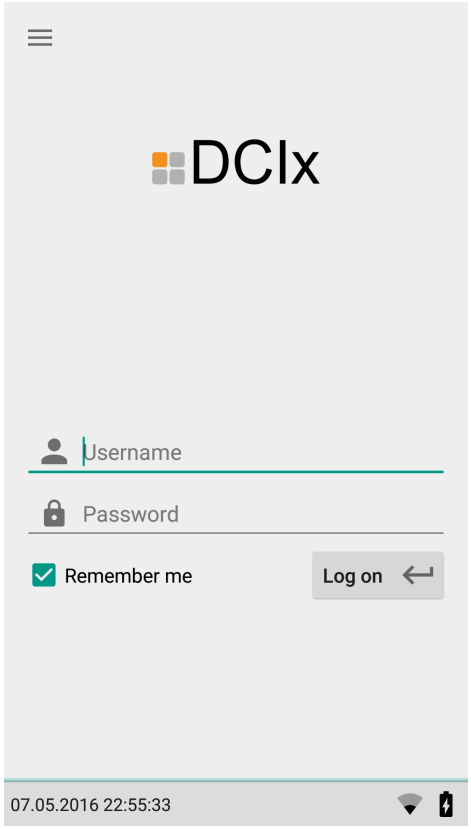


Obrázek 8.1: Struktura projektu

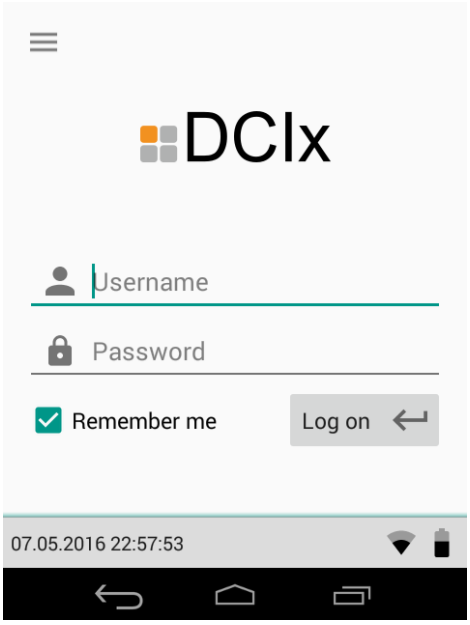
Jakmile je projekt úspěšně naimportovaný a vytvořený, je potřeba jej kompletně sestavit. To se provede klepnutím na `Build/Rebuild Project`. Tato akce odstraní všechny předchozí soubory sestavení a kompletně sestaví novou aplikaci. Po dokončení stačí klepnout na ikonu ▶ , která zobrazí dialog s výběrem zařízení, na kterém se má aplikace spustit. Po výběru zařízení a potvrzení dialogu dojde k instalaci a následnému spuštění aplikace.

Přihlášení, výchozí obrazovka

Po spuštění aplikace se zobrazí přihlašovací obrazovka s poli pro přihlášení, viz obrázky 8.2a a 8.2b.



(a) Displej 1080x1920 5"



(b) Displej 480x640 3,7"

Obrázek 8.2: Ukázka přihlašovacích obrazovek na různých zařízeních

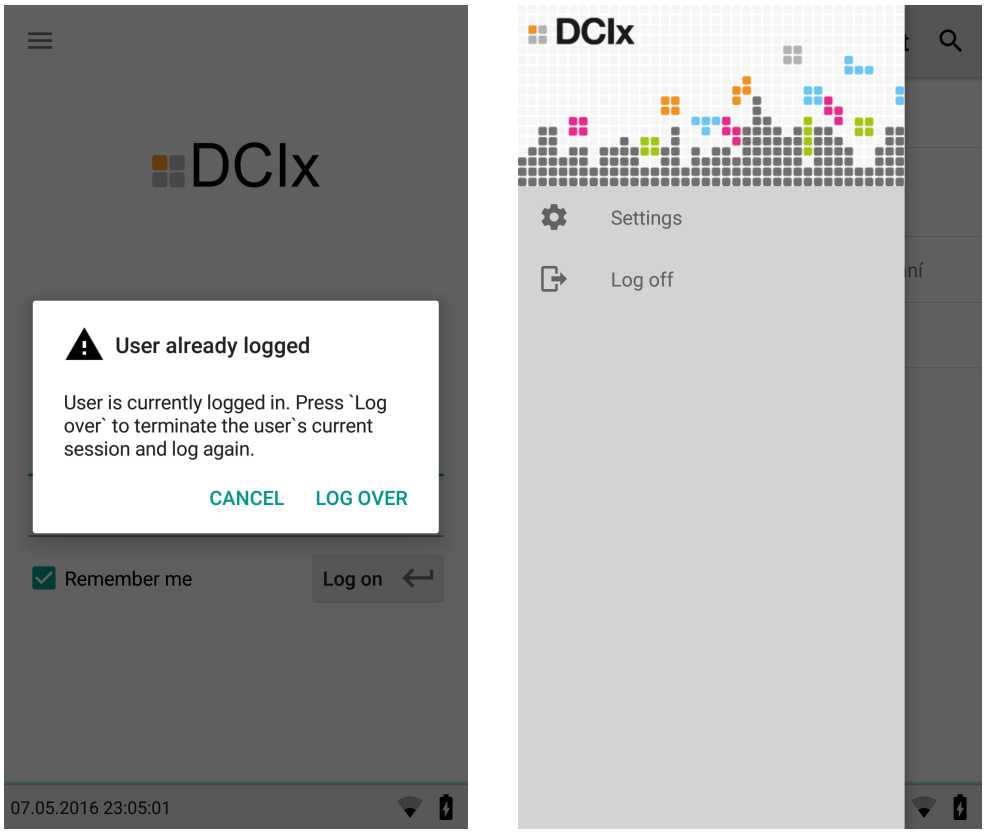
Pro přihlášení je nutné vyplnit uživatelské jméno, heslo a potvrdit tlačítkem `Log on`. Pokud si uživatel nepřeje uložení jeho osobních údajů, odškrtne tlačítko `Remember me`. Potvrzení přihlášení lze také provést klávesou `ENTER`. Na obrazovce je umožněno přejít na následující pole klávesou `ENTER`.

Opětovné přihlášení

Po přihlášení se může zobrazit dialog s informací, že je uživatel již přihlášen (viz obr. 8.3a). Stiskem tlačítka `LOG OVER` se existující relace přepíše novou a uživatel bude přihlášen.

Horní lišta

Každá obrazovka má horní navigační lištu, většinou s logem a jménem obrazovky. Pokud má obrazovka vyjížděcí menu, je v levé části lišty ikona ☰ , která menu otevírá (viz obr. 8.3b). V případě transakční obrazovky je v pravé



(a) Dialog, když je uživatel již přihlášen (b) Navigační menu seznamu transakcí

Obrázek 8.3: Ukázka obrazovek aplikace

části menu ikona ⋮ , která otevírá menu s dodatečnými položkami pro odeslání transakční obrazovky (viz obr. 8.6b).

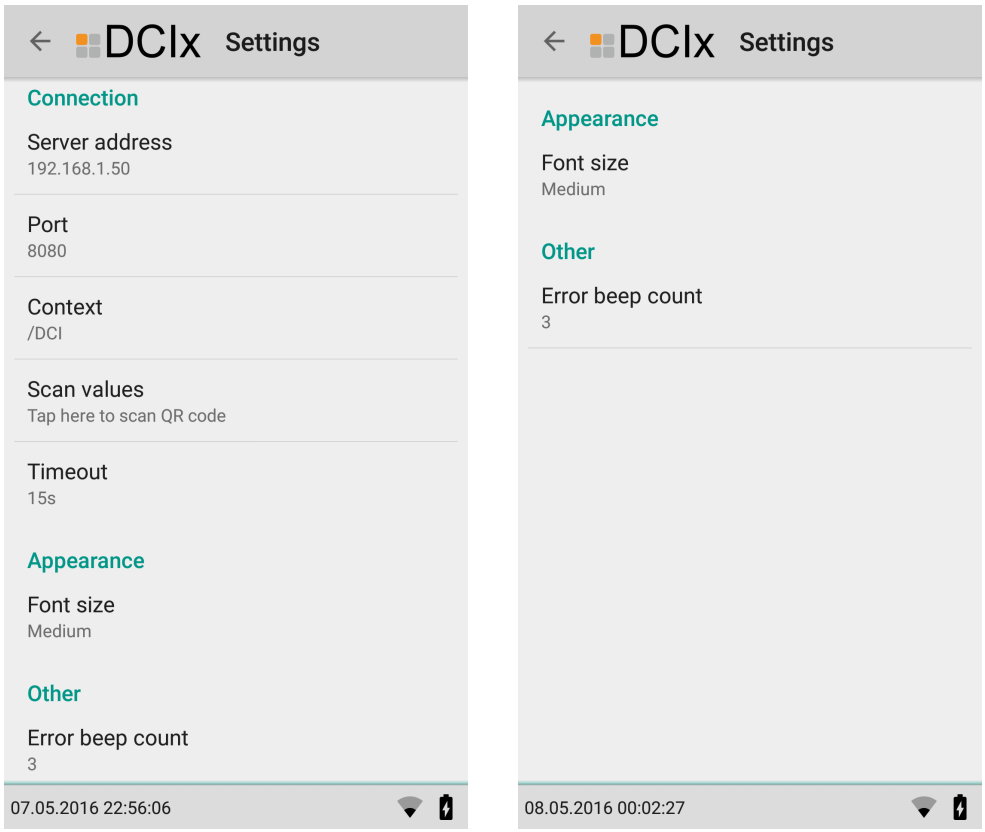
Notifikační lišta

V dolní části obrazovky je notifikační lišta. Tato lišta je vidět na všech obrazovkách, protože aplikace běží v celoobrazovkovém režimu. Vlevo v liště je zobrazen datum a čas, vpravo jsou indikátory síly signálu a stavu baterie. Po klepnutí na indikátor baterie se zobrazí informace o stavu nabití baterie.

Nastavení

Obrazovku s nastavením lze zobrazit ze dvou míst; prvním je navigační menu přihlašovací obrazovky, druhým navigační menu obrazovky se seznamem transakcí. Rozdíl mezi nimi je patrný z obrázků 8.4a a 8.4b.

Obrazovka s nastavením obsahuje ve výchozím nastavení 3 kategorie:



(a) Bez přihlášeného uživatele (b) S přihlášeným uživatelem

Obrázek 8.4: Obrazovka s nastavením

- **`Connection`**: kategorie s nastavením týkajícím se připojení k DCIx,
  - **`Server address`**: IP adresa DCIx serveru,
  - **`Port`**: port DCIx serveru,
  - **`Context`**: kontext, na kterém běží,
  - **`Scan values`**: umožňuje skenovat QR kód s nastavením připojení k DCIx. Pokud se jedná o podnikové zařízení s vestavěným skenerem, otevře se dialog, do kterého se kód naskenuje. V případě jiného zařízení s vestavěnou kamerou se spustí aplikace Barcode Scanner, která kód naskenuje. V obou případech po naskenování dojde k uložení naskenovaných hodnot,
  - **`Timeout`**: maximální čas připojení k DCIx a čtení odpovědi.
- **`Appearance`**: kategorie týkající se vzhledu aplikace,
  - **`Font size`**: umožňuje nastavit velikost písma. Na výběr je: `small` pro malé, `medium` pro střední (výchozí hodnota) a `large` pro

velké. Pokud se nově zvolená velikost písma liší od předchozí, kvůli projevení změn dojde k restartování aplikace,

- `Other`: ostatní nikam nezařazené položky,
  - **`Error beep count`**: počet pípnutí při chybě v transakční obrazovce.

Pokud se nastavovací obrazovka zobrazí z obrazovky se seznamem transakcí, tj v situaci, kdy je uživatel přihlášen, je kategorie `Connection` skryta.

Seznam transakcí

Po úspěšném přihlášení je uživateli zobrazena obrazovka se seznamem transakcí, viz obr. 8.5a. Na obrázku 8.5b je vidět situace, kdy se ztratilo připojení k síti, např. kvůli slabému signálu. Pokud by byla Wi-Fi vypnutá úplně, bude indikátor červeně přeškrtnutý. Na obrázcích si lze všimnout, že v dolní liště je možné vidět přihlašovací jméno aktuálně přihlášeného uživatele.

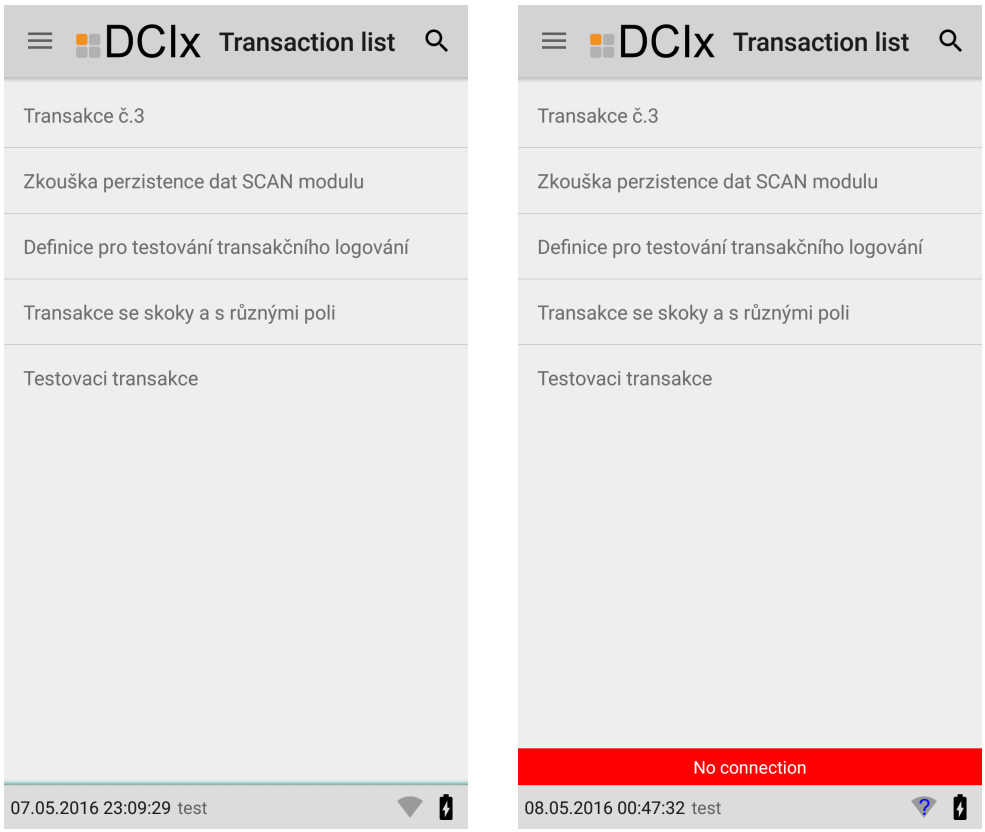
Seznam transakcí umožňuje mezi transakcemi vyhledávat. V pravé horní části obrazovky, v horní navigační liště, je tlačítko 🔍 , které otevírá pole pro zadání jakékoliv části jména transakce (viz obr. 8.6a).

Po nalezení hledané transakce na ni stačí klepnout a transakce se spustí. Na této obrazovce rovněž funguje klávesa `F9` k odhlášení.

Průchod transakcí

Poslední obrazovka v aplikaci slouží k průchodu transakcí. K odeslání aktuální transakční obrazovky slouží 2 hlavní tlačítka v dolní části: `F3 Zpět` a `Pokračovat`. Pokud má obrazovka nějaká další, jsou zobrazena v menu v horní liště (viz obr. 8.6b), které lze mj. otevřít klávesou `F1`. Odesílání obrazovky lze také provádět pomocí funkčních kláves `F2` až `F9`. Tlačítko `Continue` slouží vždy k navigaci dopředu. Klávesa `F3` naopak slouží naopak k navigaci zpět. Jsou-li v pravém horním menu nějaké další položky, které odesílají transakci, jsou vždy označeny nějakou funkční klávesou, která provede klepnutí na danou položku. **Pozor:** pokud je menu otevřené, funkční klávesy nefungují. Funkční klávesy fungují pouze tehdy, je-li menu zavřené.

Transakční obrazovka se dále skládá ze jména transakce a z elementů, které obsahuje. Odshora jsou seřazeny v tomto pořadí: chyby, nápovědy a vstupy. Na obrázku 8.7a si lze všimnout, že přijatá transakční obrazovka



(a) S připojením (b) Bez připojení

Obrázek 8.5: Seznam transakcí

obsahuje 1 chybu a 3 vstupy. Obsahuje-li transakční obrazovka chybu, zařízení přehraje chybový zvuk tolikrát, kolikrát je specifikováno v nastavení. První vstup má titulek `Reference A1` a měl již předvyplněnou hodnotu `hodnota ref A1`. Toto pole je jen pro čtení a je zakázané, proto jej nelze vymazat. V levé ikoně pole je malý zámek, který značí uzamčené pole.

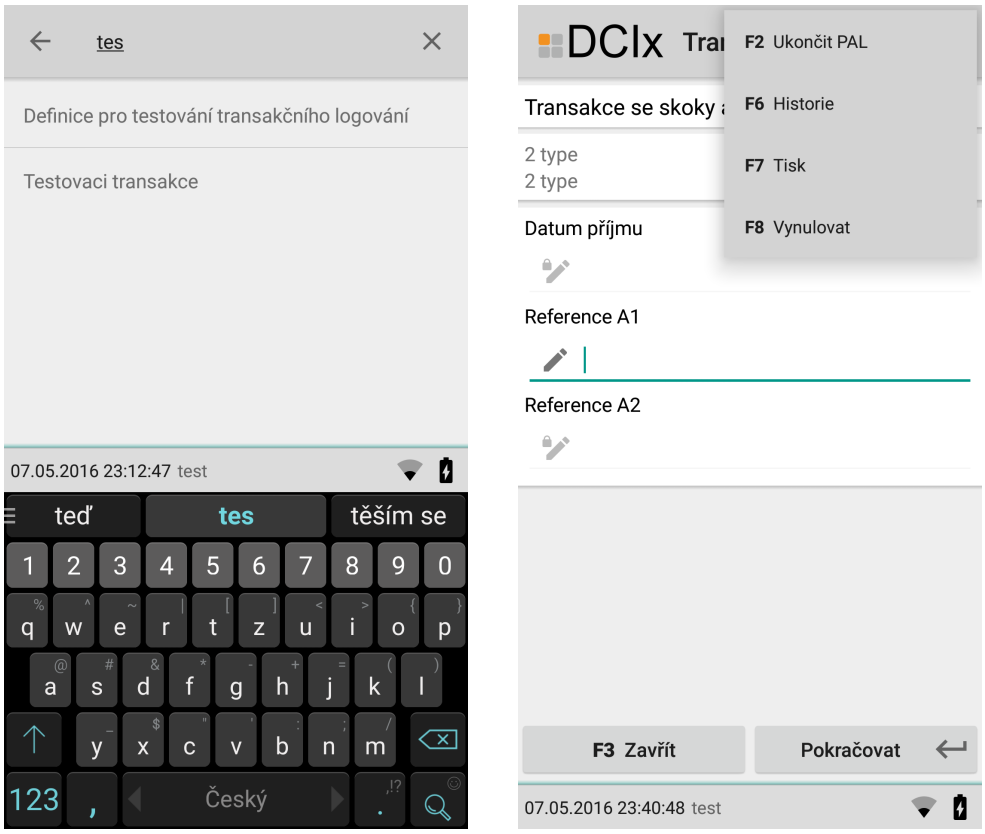
Druhé pole má titulek `Text 01`*. Červená hvězdička značí, že je pole povinné. Třetí pole je `Text 02` a je zakázané stejně jako první.

Na obrázku 8.7b je transakční obrazovka s jednou nápovědou s textem `4 scan`. Dále obrazovka obsahuje 3 pole, z nichž poslední je jen pro čtení.

Moduly

Z ikon vstupních polí na obrázcích 8.7a a 8.7b si lze všimnout, že v prvním případě se jedná o modul `type`, což značí i ikona ve tvaru psací tužky. Druhý případem je modul `scan`, ten je značen ikonou v podobě čárového kódu.

Je-li aplikace spuštěna na jiném než podnikovém zařízení a nemá vesta-



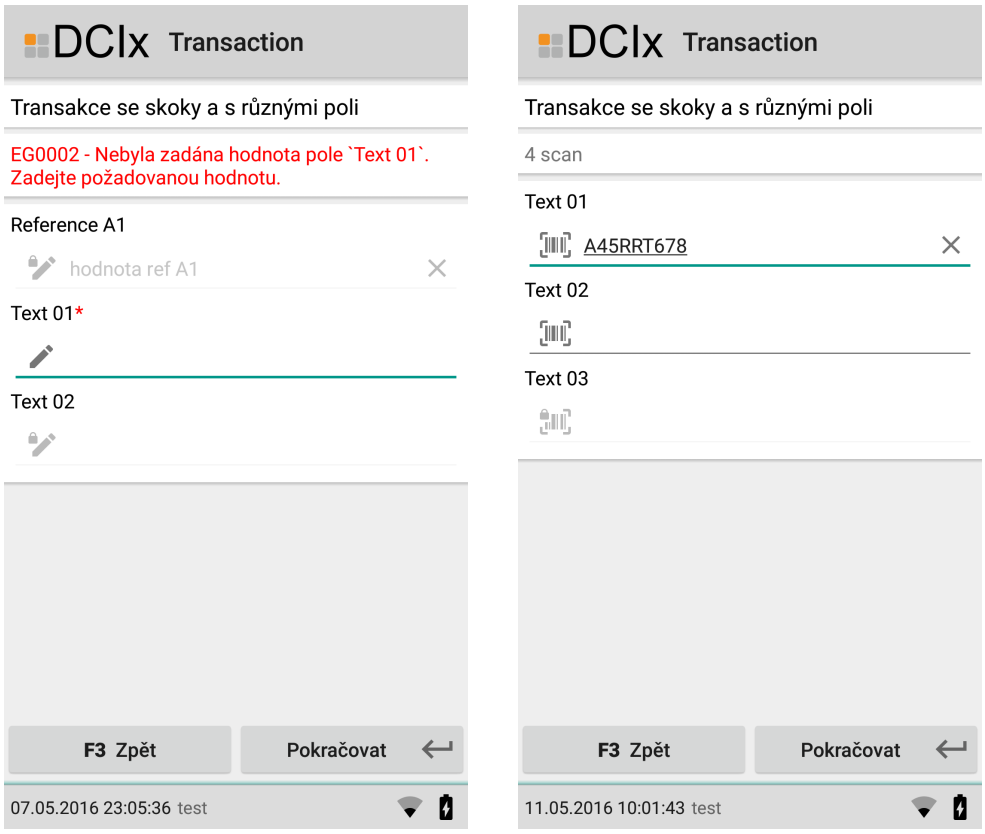
(a) Vyhledávání transakcí (b) Menu pro odesílání obrazovky

Obrázek 8.6: Ukázky obrazovek aplikace

věný skener čárových kódů, je využita ke skenování kódů vestavěná kamera (pokud zařízení nějakou má). Ke skenování kódů slouží aplikace Barcode Scanner. Skenování se spouští klepnutím na ikonu čárového kódu pole, do kterého se má skenovat. Při klepnutí na ikonu na podnikovém zařízení s integrovaným skenerem k žádné akci nedojde.

Skenovací buffer

Aplikace disponuje funkcí rychlého skenování. To je taková funkce, která v případě pomalého připojení k síti umožní skenovat čárové kódy dál. Naskenované čárové kódy jsou ukládány do paměti a jakmile přijde od DCIx nová obrazovka s nějakými volnými vstupy, jsou uložená data vložena do těchto vstupů. Pokud se data vyčerpala z paměti všechna a nějaký vstup zbývá, nic se neděje. Pokud ovšem byly zaplněny všechny vstupy a poslední naskenovaná data končila znakem `ENTER` (výchozí nastavení), dojde k automatickému odeslání transakční obrazovky.



(a) Modul `type` (b) Modul `scan`

Obrázek 8.7: Detail transakční obrazovky

V situaci, kdy jsou všechny vstupy plné a uživatel naskenuje data, jsou tato data zahozena, neboť pro ně není k dispozici žádný volný vstup. Data se tedy ukládají do paměti pouze tehdy, pokud byla zrovna odeslána transakční obrazovka a čeká se na přijetí další.

Reakce na chyby

Během používání aplikace může dojít k následujícím chybám:

- **nedostupné připojení**: akci lze buď přerušit, nebo zkusit znovu,
- **vypršení relace**: jakmile relace přihlášeného uživatele vyprší nebo je ukončena, je o tom uživatel při jakékoliv akci na DCIx informován dialogem, který ho po potvrzení vrátí na přihlašovací obrazovku,
- **neřízené chyby v DCIx**: v DCIx může nastat neřízená chyba, která pokud nastane, je uživateli v dialogu zobrazen její popis a po potvrzení dialogu je uživatel vrácen zpět na přihlašovací obrazovku.