

# **Tvorba klientské části a komunikačního rozhraní mobilní aplikace na platformě Android**

The Creation of a Client-specific and Communication Interface  
Mobile Applications on the Android Platform

Bc. Dušan Šild

---

Diplomová práce  
2013



Univerzita Tomáše Bati ve Zlíně  
Fakulta aplikované informatiky

---

Univerzita Tomáše Bati ve Zlíně  
Fakulta aplikované informatiky  
akademický rok: 2012/2013

## ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Bc. Dušan Šild**  
Osobní číslo: **A11505**  
Studijní program: **N3902 Inženýrská informatika**  
Studijní obor: **Informační technologie**  
Forma studia: **kombinovaná**

Téma práce: **Tvorba klientské části a komunikačního rozhraní mobilní aplikace na platformě Android**

Zásady pro vypracování:

1. Zrealizujte literární rešerši daného tématu.
2. Analyzujte možnosti dostupných aplikací.
3. Formou projektu navrhnete řešení komunikačního a klientského rozhraní aplikace.
4. Realizujte vytvoření mobilní aplikace dle navrženého projektu.
5. Otestujte funkčnost a vyhodnoťte efektivnost projektu.
6. Věnujte pozornost zabezpečení aplikace.

Rozsah diplomové práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. MURPHY, Mark L. **Android 2: průvodce programováním mobilních aplikací**. Vyd. 1. Brno: Computer Press, 2011, 375 s. ISBN 978-80-251-3194-7.
2. GOOGLE INC. **Android Developers** [online]. 2013 [cit. 2013-01-08]. Dostupné z: <http://developer.android.com/index.html>
3. MEIER, Reto. **Professional Android application development**. Indianapolis, IN: Wiley, 2009. ISBN 978-047-0344-712.
4. HOLZNER, Steven. **Android application development for dummies**. Hoboken, N.J.: Wiley, 2010. ISBN 978-047-0770-184.
5. MURPHY, Mark L. **Begging Android**, New York: Apress, 2009, 384s. ISBN 978-1-4302-2419-8.
6. SMITH, David; FRIESEN, Jeff: **Android Recipes: A Problem-Solution Approach**. 2nd ed. New York: Apress, 2011, 550s. ISBN 978-1-4302-3414-2.

Vedoucí diplomové práce: **Ing. Michal Krbeček**

Ústav matematiky

Datum zadání diplomové práce: **22. února 2013**

Termín odevzdání diplomové práce: **22. května 2013**

Ve Zlíně dne 22. února 2013

  
prof. Ing. Vladimír Vašek, CSc.  
*děkan*



  
doc. Mgr. Roman Jašek, Ph.D.  
*ředitel ústavu*

## **ABSTRAKT**

Diplomová práce se zabývá návrhem a vývojem mobilní aplikace na platformě Android v jazyce Java. Cílem práce je vytvořit mobilní aplikaci pro správu a komunikační rozhraní pro synchronizaci kontaktů mezi více lidmi či zařízeními. Věnuje se dále zabezpečení, efektivnosti aplikace, využitím a budoucím potenciálem aplikace na trhu.

Klíčová slova: Platforma Android, mobilní aplikace, webová služba, formát JSON, databáze SQL, synchronizace kontaktů.

## **ABSTRACT**

Diploma thesis describes the design and development of mobile applications on the Android platform in Java. The aim is to create a mobile application for managing bussiness contacts and communication interface for synchronizing them between multiple people or devices. Application security, efficiency, utilization and future potential on the market are also discussed.

Keywords: Android platform, mobile application, web service, JSON format, SQL database, synchronization of contacts.

### Poděkování

Chtěl bych poděkovat rodině a nejbližším za neocenitelnou podporu během celého studia, přátelům za objektivní kritiku a pomoc s uspořádáním myšlenek a v neposlední řadě také vedoucímu práce Ing. Michalu Krbečkovi za rady a nápady poskytnuté při konzultacích.

„Nejste-li schopni vysvětlit problém šestiletému dítěti, ve skutečnosti tomu sami příliš nerozumíte.“

Albert Einstein

**Prohlašuji, že**

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

**Prohlašuji,**

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně

.....  
podpis diplomanta

**OBSAH**

<b>ÚVOD.....</b>	<b>9</b>
<b>I TEORETICKÁ ČÁST .....</b>	<b>10</b>
<b>1 PLATFORMA ANDROID .....</b>	<b>11</b>
1.1 HISTORIE A VÝVOJ.....	11
1.2 ARCHITEKTURA.....	12
1.3 AKTUÁLNÍ VERZE .....	15
1.4 NEDOSTATKY PLATFORMY .....	15
1.5 JAZYK JAVA .....	16
1.5.1 Historie.....	16
1.5.2 Rozšíření Javy .....	17
1.5.3 Vlastnosti jazyka .....	17
1.6 NÁVRH SOFTWARE .....	18
1.7 NÁVRHOVÉ PRINCIPY .....	20
1.7.1 Principy SOLID.....	21
1.7.2 Deméteřin zákon (Law of Demeter – LoD) .....	21
1.7.3 Principy DRY.....	22
1.7.4 Principy GRASP (General Responsibility Assignment Software Patterns).....	22
<b>2 APLIKACE DOSTUPNÉ NA TRHU .....</b>	<b>24</b>
<b>3 NÁVRH MOBILNÍ APLIKACE A WEBOVÉ SLUŽBY .....</b>	<b>27</b>
3.1 NÁVRH STRUKTURY .....	28
3.2 NÁVRH MOBILNÍ APLIKACE.....	29
3.2.1 Návrh datové vrstvy .....	29
3.2.2 Návrh uživatelského rozhraní (UI) a aplikační vrstvy .....	32
3.2.3 Synchronizační služba.....	36
3.3 NÁVRH WEBOVÉHO ROZHŘANÍ.....	37
3.3.1 Návrh datové vrstvy .....	38
3.3.2 Návrh komunikačního rozhraní služby .....	38
<b>II II. PRAKTICKÁ ČÁST.....</b>	<b>39</b>
<b>4 IMPLEMENTACE .....</b>	<b>40</b>
4.1 IMPLEMENTACE MOBILNÍ APLIKACE.....	40
4.1.1 Datová vrstva.....	40
4.1.2 Uživatelské rozhraní – prezentační vrstva .....	45
4.1.3 Aplikační vrstva .....	46
4.1.4 Synchronizační služba.....	51
4.2 IMPLEMENTACE WEBOVÉ SLUŽBY.....	52
4.2.1 Datová vrstva.....	53

---

4.2.2	Komunikační rozhraní služby .....	55
<b>5</b>	<b>ZABEZPEČENÍ APLIKACE .....</b>	<b>57</b>
<b>6</b>	<b>TESTOVÁNÍ A VYHODNOCENÍ EFEKTIVNOSTI VYTVOŘENÉ APLIKACE .....</b>	<b>59</b>
	<b>ZÁVĚR .....</b>	<b>60</b>
	<b>ZÁVĚR V ANGLIČTINĚ .....</b>	<b>62</b>
	<b>SEZNAM POUŽITÉ LITERATURY .....</b>	<b>64</b>
	<b>SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK .....</b>	<b>66</b>
	<b>SEZNAM OBRÁZKŮ .....</b>	<b>67</b>
	<b>SEZNAM PŘÍLOH.....</b>	<b>68</b>

## ÚVOD

Celosvětově nejpoužívanější mobilní operační systém Android je vyvíjen jako otevřená platforma, za jejíž použití nemusejí výrobci telefonů platit. Tyto přístroje tak přicházejí na trh za mnohem nižší ceny a jsou dostupné milionům majitelů chytrých zařízení, od jednoduchých přes pokročilé až po náročné uživatele.

V Evropě tvoří zařízení se systémem Android drtivou většinu, ať už to jsou chytré mobily či tablety, které postupně nahrazují běžné osobní počítače a notebooky. Android platforma předčila dokonce Apple, Windows a jiné operační systémy, jež se donedávna držely v jasné špičce mezi mobilními technologiemi zejména ve Spojených Státech Amerických. Největším výrobcem chytrých telefonů a zároveň i telefonů obecně je jihokorejský Samsung následovaný americkým Apple. Samsung navíc svůj náskok neustále zvyšuje.

Chytré telefony jsou stále více součástí každodenního života a čím dál častěji si nachází cestu do společností jako služební telefony. Mobilní aplikace jsou jejich nedílnou součástí a poskytují majitelům chytrých telefonů mnoho užitečných funkcí. Jednou z takových funkcí může být firemní telefonní seznam, který obsahuje důležité kontakty a umožňuje jejich sdílení mezi spolupracovníky ve společnosti, případně v oddělení.

Cílem práce je vytvořit mobilní aplikaci, která umožní správu obchodních kontaktů, představující kontaktní informace na společnosti a jejich zástupce (jednatele) a komunikačního rozhraní poskytujícího možnosti synchronizace. Ta je výhodou zejména při využití těchto kontaktů více lidmi, kdy dochází ke změnám či přidání nových údajů automaticky, kdykoli je zařízení připojeno k Internetu, bez nutnosti uživatele cokoli udělat, kdekoli se právě nachází.

Práce se zabývá analýzou zadání, jednotlivými požadavky a možnostmi jejich splnění. Následně návrhem mobilní aplikace s využitím ověřených postupů a metod. Dále stavbou aplikace podle návrhu, kde jsou rozebrány případné problémy, výhody či nevýhody řešení, omezení, které se vyskytla a vypořádání se s nimi. Práce se věnuje také zabezpečení aplikace i komunikace mezi aplikací a službou, i případným bezpečnostním rizikům, které připadají v úvahu. Nakonec se probírá otestování a vyhodnocení vytvořené aplikace i synchronizační služby a hodnotí, jsou-li splněny kladené požadavky.

## I. TEORETICKÁ ČÁST

## 1 PLATFORMA ANDROID

Android je rozsáhlá open source platforma, která vznikla zejména pro mobilní zařízení (chytré telefony, PDA, navigace a tablety). Zahrnuje v sobě operační systém (založený na jádru Linux), middleware, uživatelské rozhraní aplikace. Vyvíjí ho konsorcium Open Handset Alliance. Při vývoji systému byla brána v úvahu omezení, kterými disponují klasické mobilní zařízení jako výdrž baterie, menší výkonnost a málo dostupné paměti. Zároveň bylo jádro Androidu navrženo pro běh na různém hardwaru. Systém tak může být použit bez ohledu na použitý hardware, velikost či rozlišení obrazovky. [1]

Samotná platforma Android dává k dispozici nejen operační systém s uživatelským prostředím pro koncové uživatele, ale i kompletní řešení nasazení operačního systému (specifikace driverů aj.) pro mobilní operátory a výrobce zařízení a v neposlední řadě pro vývojáře aplikací poskytuje efektivní nástroje pro jejich vývoj – Software Development Kit (SDK).

### 1.1 Historie a vývoj

Společnost Android Inc. byla založena v Kalifornii v říjnu 2003 Andym Rubinem, Richem Minerem, Nickem Searsem a Chrisem Whitem. Google Inc. ji v srpnu roku 2005 odkoupil a udělal z ní svoji dceřinou společností. [2]

Po odkupu společností tým Googlu pod vedením Andyho Rubina vyvinul platformu založenou na Linuxovém jádře a v září roku 2007 Google získal několik patentů v oblasti mobilních technologií. [3]

5. listopadu v roce 2007 bylo vytvořeno uskupení Open Handset Alliance. Konsorcium, které zahrnovalo společnosti zabývající se výrobou mobilních telefonů, čipů nebo mobilních aplikací, např. Google, HTC, Intel, LG, Motorola, NVIDIA, Qualcomm, Samsung, Texas Instruments a dalších 25 společností. Cílem tohoto konsorcia bylo vyvinout otevřený standard pro mobilní zařízení. V ten samý den Open Handset Alliance ohlásil svůj první produkt, Android, otevřenou mobilní platformu postavenou na jádře Linux verze 2.6. [1]

V říjnu roku 2008 byl ve Spojených státech amerických uveden první komerční telefon vyrobený firmou HTC s operačním systémem Android (v České republice byl uveden v lednu 2009) a zároveň s tím bylo uvolněno SDK verze 1.0. V roce 2009 roste počet

zařízení používající Android na více jak dvacet. Na konci roku 2010 se Android stal vedoucí smartphone platformou a na počátku roku 2012 už jednoznačně dominoval trhu se smartphony.

## 1.2 Architektura

Architektura operačního systému Android je rozdělena do 5 vrstev. Každá vrstva má svůj účel a nemusí být přímo oddělena od ostatních vrstev.

Nejnižší vrstva architektury je jádro operačního systému, které tvoří abstraktní vrstvu mezi používaným hardwarem a zbytkem softwaru ve vyšších vrstvách. Jádro systému Android je postaveno na Linuxu ve verzi 2.6. Je využito jeho mnoha vlastností, jako podpora správy paměti, správa sítí, zabudované ovladače nebo správy procesů, například souběžného běhu aplikací, které běží jako samostatné procesy s oprávněním stanoveným systémem. Tato vlastnost přispívá ke stabilitě a také ochraně systému. Naopak systém nepodporuje grafické uživatelské rozhraní X Window System a ani úplnou sadu GNU knihoven. Důvodem použití jádra Linux byla také vlastnost poměrně snadného sestavení na různých zařízeních a tím zaručená přenositelnost. [4]

Další vrstvou jsou knihovny, které jsou napsány v C/C++ kódu a využívají je různé komponenty systému. Tyto funkce jsou vývojářům poskytnuty prostřednictvím Android Application Framework. Zde jsou uvedeny pouze některé příklady knihoven:

**Media Framework** – knihovna podporuje přehrávání video a audio formátů, stejně jako obrazové soubory.

**WebKit** – knihovna webového prohlížeče, který podporuje i vložené náhledy webových stránek.

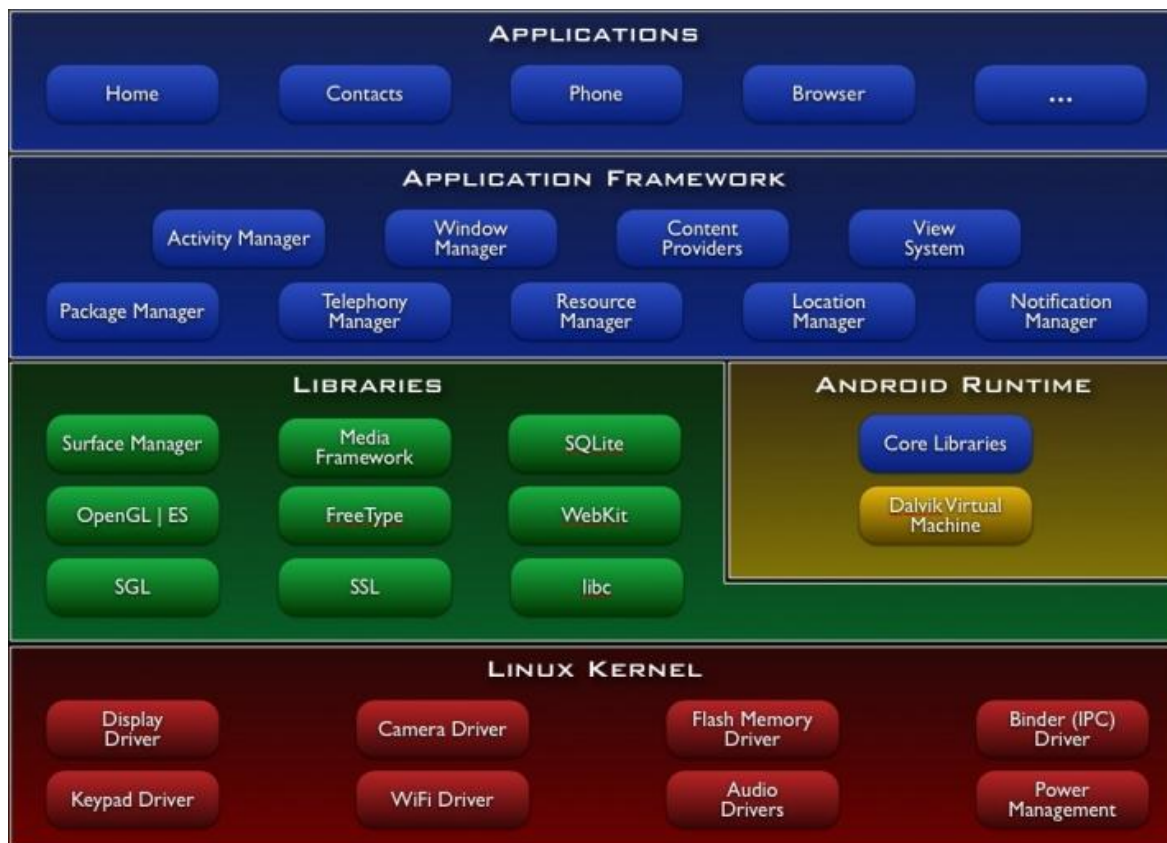
**libc** – odvozená BSD standardní knihovna systému C vyladěná pro embedded zařízení.

**SQLite** – odlehčená relační databázová knihovna.

**OpenSSL** – knihovna pro využití protokolu SSL v síťové komunikaci.

**FreeType** – knihovna pro rendering bitmapových a vektorových fontů.

**OpenGL** – knihovna na vykreslování 3D grafiky. [5]



Obr. 1. Architektura OS Android [6]

Android Runtime vrstva obsahuje aplikační virtuální stroj zvaný Dalvik VM, který byl vyvíjen od roku 2005 speciálně pro Android. Dalvik Virtual Machine (DVM) je registrově orientovaná architektura, využívá základních vlastností Linuxového jádra, jako je správa paměti nebo práce s vlákny. Vznik nového virtuálního stroje byl iniciován ze dvou důvodů. Prvním důvodem byla licenční práva, kdy jazyk Java a jeho knihovny jsou volně šiřitelné, zatímco Java VM není. Dalším důvodem byla optimalizace virtuálního stroje pro mobilní zařízení a to především v oblasti poměru úspory energie a výkonu. V této vrstvě jsou také obsaženy základní knihovny programovacího jazyka Java. Knihovny se svým obsahem blíží platformě Java Standard Edition. Hlavní rozdíl je v nepřítomnosti knihoven pro uživatelské rozhraní (AWT a Swing), které byly nahrazeny knihovnami uživatelského rozhraní pro Android nebo přidání knihovny Apache pro práci se sítí. Překlad aplikace napsané pro Android probíhá zkompilením zdrojového kódu do Java byte kódu pomocí stejného kompilátoru, jako je používán v případě překladu Java aplikací. Poté se překompiluje Java byte kód pomocí Dalvik kompilátoru a výsledný Dalvik byte kód je

spuštěn na DVM. Každá spuštěná Android aplikace běží ve svém vlastním procesu, s vlastní instancí DVM. [4]

Vrstva **Application framework** je pro vývojáře nejdůležitější. Poskytuje přístup k velkému počtu služeb, které mohou být použity přímo v aplikacích. Tyto služby mohou zpřístupňovat data z jiných aplikací, prvky uživatelského rozhraní, upozornovací stavový řádek, aplikace běžící na pozadí, hardware používaného zařízení a mnoho dalších funkcí. Základní sada služeb zahrnuje především:

Sada prvků **View** – Tyto prvky jsou použity pro sestavení uživatelského rozhraní jako seznamy, textové pole, tlačítka, checkboxy a jiné.

**Content providers** – Umožňuje přístup k obsahu (např. kontakty) jiných aplikací.

**Resource manager** – Poskytuje přístup „nekódovým“ zdrojům, jako jsou řetězce, grafika, přidané soubory.

**Notification manager** – Umožňuje všem aplikacím zobrazit vlastní upozornění ve stavovém řádku.

**Activity manager** – Řídí životní cyklus aktivit a poskytuje orientaci v zásobníku s aktivitami. [6]

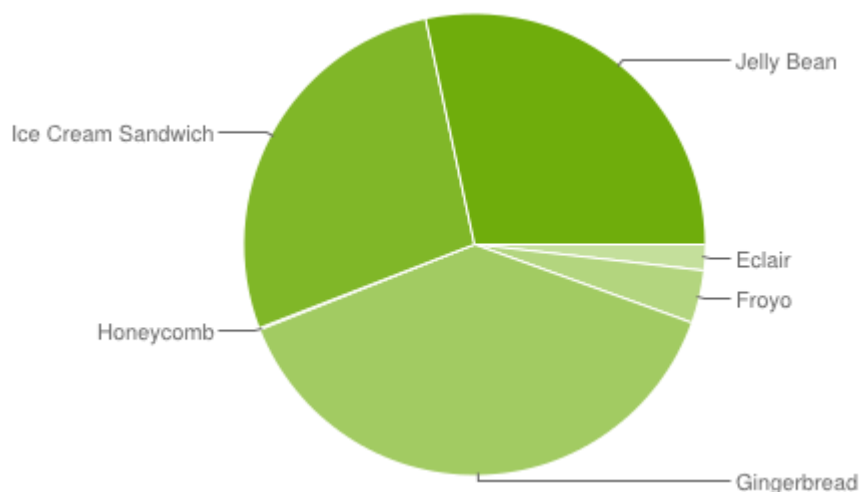
Nejvyšší vrstvu systému tvoří základní aplikace, které využívají běžní uživatelé. Může jít o aplikace předinstalované nebo dodatečně stažené z Android Marketu. Například emailový klient, SMS program, kalendář, mapy, prohlížeč, kontakty a další aplikace i od třetích stran. [7]

Základní stavební kameny v aplikacích jsou komponenty *Activity* reprezentující obrazovku, *Service* umožňující provádět akce na pozadí, *Content providers* poskytující přístup k uživatelským datům jiných aplikací a *Broadcast receiver* reagující na příchozí oznámení. Všechny tyto komponenty musí být definovány v souboru *AndroidManifest.xml*, uloženém v kořenovém adresáři projektu. Kromě *Content provideru* mohou komponenty mezi sebou kooperovat pomocí zpráv, tzv. *Intentů*. [6], [8]

### 1.3 Aktuální verze

Poslední vydanou verzí je Jelly Bean, s označením 4.1-4.2. Byla představena v červenci 2012, jako další vylepšení čerstvě nasazené verze Ice Cream Sandwich (4.0), která byla představena v listopadu 2011.

Obě tyto verze se začínají pozvolna prosazovat na trhu, avšak stále je nejpoužívanější verzí Android Gingerbread (2.3). Ten běží na 38,4% všech Android zařízení (1. 5. 2013). Důvodem je hlavně neochota výrobců vydat aktualizaci pro starší mobilní zařízení. [7]



Obr. 2. Nejpoužívanější verze OS Android (Zdroj: Oficiální stránky Android)

### 1.4 Nedostatky platformy

Každý výrobce chytrého telefonu si může systém Android upravit, jak uzná za vhodné. Nejen z tohoto důvodu jsou výrobci také zodpovědní za aktualizace systému na uživatelských zařízeních. A zde je kámen úrazu, protože úpravy a přizpůsobení systému výrobcům trvá poměrně dlouho. Navíc záleží pouze na jejich rozhodnutí, které zařízení nový systém dostane a které ne.

Jednotliví výrobci také systém vybavují svými aplikacemi – jedná se o takzvaný bloatware neboli nevyžádané předinstalované aplikace. I když je uživatel nepoužívá, není možno je odstranit běžnou cestou a zbytečně tak zabírají paměť zařízení.

Google při vývoji platformy stanovil požadavky na hardware velmi volně. Výhodou sice je přenositelnost na různé typy zařízení (telefony, fotoaparáty, televize, tablety i počítače), ale s tím přichází značná složitost při vývoji aplikací, na kterou si vývojáři stěžují. I přes

všechny mechanismy, které platforma poskytuje pro přizpůsobení různým displejům, není prakticky možné dosáhnout toho, aby aplikace vypadala na různých displejích stejně. [9]

Dalším poměrně častým problémem je velká chybovost nejen systému samotného, ale také aplikací. K chybovosti systému částečně přispívá i to, že každý výrobce si systém přizpůsobuje po svém. Toto přizpůsobení mají na starosti programátoři a ti nejsou neomylní. Obzvláště u levných telefonů je tento problém velmi závažný, protože se zde projevuje navíc nevytlačení systému. Uživatelské zkušenosti s takovým zařízením jsou velmi tristní a uživatelé bývají spíše od platformy odrazeni.

K chybovosti aplikací přispívá poměrně velkorysý přístup k nahrávaným aplikacím na Google Play. Tím se zde dostává malware, nestabilní aplikace a programy, které parazitují na práci někoho jiného (například tím, že porušují autorská práva). S tím také souvisí slabá ochrana aplikací proti krádeži a tak vývojáři přicházejí o zisk. [9] [10] [11]

## 1.5 Jazyk Java

### 1.5.1 Historie

Základy Javy lze nalézt v projektu Oak, který vznikl ve firmě Sun na počátku devadesátých let pro řízení elektronických výrobků. V roce 1994 byl přenesen jako programovací jazyk do prostředí počítačů pod názvem Java (horká káva).

Velice významným faktorem pro rozvoj používání Javy se stalo v roce 1995 zařazení její podpory do tehdy velice populárního prohlížeče Netscape Navigator 2.0. Tato podpora umožňovala rozšíření funkčnosti webových stránek, pomocí Java appletů, programů v Javě, stahovaných současně s webovou stránkou a spouštěných přímo v prohlížeči na straně klienta. Později tato podpora byla zavedena i do dalších prohlížečů a applety se staly nedílnou součástí internetových stránek.

Tento mechanismus dokonce vedl ke vzniku myšlenky NC (net computers), které měly existovat bez pevných disků, operačního systému a lokálně instalovaných programů, měly obsahovat pouze integrovaný internetový prohlížeč a veškeré programy se měly spouštět skrz něj, ze síťových serverů ve formě Java appletů. Jako výhody se kromě úspor na hardwarových komponentách uváděla zjednodušená administrace počítačových sítí, zvýšená bezpečnost, rychlé instalace a aktualizace programů a další důvody. Tato vize se

neujala, ceny těchto NC, bez diskových mechanik nebyly o mnoho nižší než standardní PC, a co bylo možná podstatnější, ze strany počítačových firem nedošlo k masovému přepsání aplikačního softwaru do formy appletů, i když pokusy zde byly.

I přes neúspěch tohoto pokusu (který snad měl šanci nahradit stávající platformu PC) si Java své místo na slunci udržela, na webových stránkách našla svojí "parketu" v plně internetových aplikacích, které zprostředkovávají komunikaci mezi klientem na internetovém prohlížeči a službách přístupných přes internetový server. Zde se využily výhody Javy, její robustnost, stabilita, rozsah funkcí a hlavně bezpečnost. I proto podporu Java appletů najdeme v největší míře na webových stránkách internetových bankovníctví a dalších aplikací, vyžadujících vysokou míru stability a zabezpečení.

### 1.5.2 Rozšíření Javy

Java se však neomezuje pouze na Java applety, právě naopak. Její výhodu, multiplatformitu, deklarovalo populární heslo: „Write once, run everywhere“. Zdrojový kód je při vývoji přeložen do spustitelného mezikódu (byte code), který lze pak spouštět pomocí nainstalovaného runtime prostředí (Java Virtual Machine), přímo na různých typech počítačů či technických zařízeních bez nutnosti nového překladu.

Protože však tato přenositelnost není a nemůže být (hlavně z technických příčin rozhraní) stoprocentní, vyvinulo se několik edicí Javy, lišících se drobnými rozdíly a rozšířeními a spojených společným jazykem a velkou skupinou knihoven.

Java je rozšířená na rozsáhlé skupině počítačů různých specifikací, na Linuxu, Unixu, Windows, a dalších. Většina distribucí Linuxu již v sobě obsahuje vývojové prostředky i runtime prostředí Javy.

Na rozdíl od jazyka C, který je rozšířen stejně nebo i více, by přenositelnost programu měla být dána ne pouze jen na úrovni zdrojového kódu, ale spustitelného programu. [12]

### 1.5.3 Vlastnosti jazyka

Java je vyspělý programovací jazyk, obsahující všechny vlastnosti, které jsou vyžadovány v moderním programování, od modularity programu, řídicích konstrukcí, přes silnou typovou kontrolu, práci s vlákny, ošetření výjimek, správu paměti, i silnou podporu pro databáze, XML a síťové operace.

K jejím výhodám, kromě již zmíněné multiplatformity, patří robustnost, škálovatelnost a vysoká bezpečnost.

Nižší rychlost, způsobená zpracováním v runtime prostředí může být urychlena s pomocí specializovaných překladačů na cílovém prostředí (Java just-in-time, JIT).

I když základní vývojové prostředí obsahuje pouze řádkový překladač, existuje mnoho vývojových nástrojů a rozšíření dalších firem autorů včetně IDE, i s podporou vývoje GUI aplikací.

Obsah Javy však nelze omezit jen na výčet jejích příkazů. Java je především silně objektová, což umožňuje v ní modelovat, vytvářet, používat a rozšiřovat rozsáhlé knihovny a systémy.

Právě objektově je třeba myslet ne jen při psaní programu, ale již při návrhu a analýze. Pro tyto účely byl vytvořen UML, Unified Modeling Language, modelovací jazyk slouží k objektovému modelování a popisu konstrukcí reálného světa, převáděných do světa počítačů a informačních systémů. [13]

## 1.6 Návrh softwaru

V ideálním světě by bylo možné každý systém spustit okamžitě. Systém by nepotřeboval žádný prostor, nikdy by neobsahoval chyby a jeho výstavba by nestála ani pětník. Ve skutečném světě je však jednou z klíčových úloh návrháře vyvážit protichůdné vlastnosti a nalézt rovnováhu mezi nimi. Je-li rychlá odezva důležitější než minimalizace doby vývoje, může se návrhář rozhodnout pro určitý návrh. Je-li však důležitější opak, vytvoří návrhář návrh jiný.

Jelikož návrh není procesem deterministickým, jsou návrhové techniky obvykle heuristické (spíše se používají empirická pravidla nebo se zkoušejí věci, jež by měly fungovat), místo aby se využívalo opakovatelných procesů se zaručenými a předvídatelnými výsledky. Součástí návrhu je rovněž dobře známý proces pokusů a omylů. Návrhový nástroj nebo návrhová technika, jež fungovaly v jednom případě, nemusí fungovat v jiném projektu.

Návrhy vznikají a zdokonalují se díky postupným revizím, neformálním diskusím, přepisováním a korekturám v kódu.

*„Naplánujte si, že jeden návrh zahodíte. Zahodíte jej stejně.“*

*– Fred Brooks*

*„Pokud naplánujete, že zahodíte jednu variantu, nakonec zahodíte dvě.“*

*– Craig Zerouni*

Vysoce kvalitní návrh má několik obecných znaků. Určité cíle si však navzájem odporují. Úkolem dobrého návrhu je vytvořit dobrou sadu kompromisů z protichůdných cílů. Nyní obecné znaky budou popsány blíže:

### **Minimální složitost**

Primárním cílem návrhu by měla být minimalizace složitosti. Je nutné vyvarovat se tvorby „důmyslných“ návrhů, raději vytvořit „jednoduchý“ a snadno srozumitelný návrh. Pokud návrh neumožňuje po vnoření se do určité části bezpečně ignorovat většinu ostatních součástí programu, nejedná se o příliš dobrý návrh.

### **Snadná údržba**

Jde o návrh, jenž zohledňuje nutnost údržby programátorem. Systém musí být navržen tak, aby sám sebe dostatečně popisoval.

### **Volné vazby**

Programátor by se měl snažit mezi různými částmi programu navrhovat minimální počet propojení. Používat zásadu volby vhodných pojmů (abstrakce) nejen při návrhu rozhraní tříd, ale rovněž při návrhu zapouzdření, ukryvání informací. Minimální souvislost minimalizuje rozsah prací nezbytných při integraci, testování a údržbě. [14]

### **Rozšiřitelnost**

Znamená, že programátor může rozšířit systém bez dopadu na nadřazenou strukturu. Může pak změnit jakoukoliv součást systému, aniž se to negativně odrazilo na ostatních součástech. Nejpravděpodobnější změny obvykle způsobí systému nejmenší trauma.

### **Znovupoužitelnost**

Navrhovat systém tak, aby se daly jeho jednotlivé součásti použít i v ostatních systémech.

### **Vysoká míra užití**

Danou třídu používá spousta ostatních tříd. To je znak návrhu systému, který dobře využil pomocné třídy na nižší úrovni systému.

### **Užití nepříliš velkého množství tříd**

Programátor by měl navrhovat třídu tak, aby používala středně velký počet jiných tříd. Užití poměrně velkého množství tříd (více než 7) znamená, že třída používá velký počet ostatních tříd a že se může stát zdrojem složitosti.

### **Přenositelnost**

Systém je navržen tak, aby jej bylo možné přenést do jiného prostředí.

### **Štíhlost**

Navrhovat systémy tak, aby neobsahovaly žádné zbytečné součásti navíc. Při úpravách kódu se totiž musí velmi často vytvářet dodatečný kód, který je třeba brát v úvahu, kontrolovat a testovat jej. Budoucí verze softwaru musí zůstat kompatibilní rovněž s tímto dodatečným kódem. [15]

### **Rozvrstvení**

Návrh je dobré rozložit do několika vrstev, aby se programátor mohl na systém dívat rovněž pod jiným úhlem. Systém přitom musí ze všech úhlů vypadat stejně. [14]

### **Standartní techniky**

Programátor se musí pokusit dát celému systému důvěrně známou formu užitím standardizovaných běžných postupů. [15]

## **1.7 Návrhové principy**

Návrh je nejdůležitějším prvkem celé tvorby softwaru. Návrh předchází implementaci a určuje vlastnosti budovaného softwaru. Dojde-li k chybám během návrhu, bývají tyto chyby nejnákladnějšími chybami projektu, pokud se neodstraní co nejdříve.

Návrhové vzory ukazují, jak řešit typické problémy při návrhu software v objektově orientovaných jazycích. Principy objektově orientovaného návrhu pomáhají vytvořit kvalitní návrh softwaru, který bude předcházet vzniku symptomů špatného designu (ztuhlost, křehkost, špatná znovupoužitelnost).

Hlavním smyslem těchto principů je omezování závislostí mezi jednotlivými částmi software a omezování potřeb sekundárních úprav po každé změně. Závislosti jsou hlavní příčinou neudržitelnosti softwaru a vyžadují zásadní změny návrhu, nebo rovnou vytvoření nové verze od začátku.

Uvedené principy poslouží jako vodítka při zvažování různých variant návrhu. V mnoha případech neexistuje jedno správné řešení, ale je nutné volit mezi několika neideálními řešeními. Naopak dogmatické lpění na dodržování určitého principu může být spíše na škodu. [16]

### 1.7.1 Principy SOLID

Autorem těchto 5 principů je Robert C. Martin:

- Princip jedné odpovědnosti (Single Responsibility Principle – SRP)  
„Třídy by měly mít jedinou zodpovědnost / jediný důvod ke změně.“
- Princip otevřenosti a uzavřenosti (Open-Closed Principle – OCP)  
„Třídy by měly být otevřené pro rozšiřování, ale uzavřené pro změny.“
- Liskovův princip zaměnitelnosti (Liskov Substitution Principle – LSP)  
„Podtřídy by měly být zaměnitelné s jejich bazovými třídami.“
- Princip oddělení rozhraní (Interface Segregation Principle – ISP)  
„Více specifických rozhraní je lepší než jedno univerzální rozhraní.“
- Princip obrácení závislostí (Dependency Inversion Principle – DIP)  
„Závislost by vždy měla být na abstraktním ne na konkrétním. Konkrétnější musí záviset na abstraktnějším ne naopak.“ [17]

### 1.7.2 Deméteriův zákon (Law of Demeter – LoD)

Taktéž nazýváno pravidlem nejmenší znalosti (Principle of least knowledge) neboli ukrývání informací. Princip definuje omezení v tom, s jakými objekty by mělo být přímo komunikováno a s jakými ne - výsledný kód je méně vzájemně provázaný a jeho udržování je jednodušší. Jde tedy o doporučení důkladného návrhu veřejných rozhraní tříd.

Aplikace Deméteřina zákona vede k přehlednějšímu a čistšímu kódu. Udržování software je pak mnohem jednodušší a je sníženo riziko vzniku následných chyb. Tento princip nahlíží na třídy jako na černé skříňky, není nutno znát vnitřní implementaci třídy, ale pouze její veřejné rozhraní. [18]

### 1.7.3 Principy DRY

Zkratka „Don't Repeat Yourself“ neboli „Neopakuj se“. Pomáhá vytvářet kvalitnější kód a ušetří práci při jeho údržbě. Autoři principu jsou Andrew Hunt a David Thomas (v díle: Programátor pragmatik). Tvrdí, že každá část programu by měla být unikátní, takže při případně změně programu je daná úprava prováděna pouze jednou a na jednom místě. Dojde-li k situaci, při které programátor kopíruje kód, měl by daný kód být přepracován, například za použití abstrakce. [19]

### 1.7.4 Principy GRASP (General Responsibility Assignment Software Patterns)

Přidělování zodpovědností (kdo bude co dělat) a návrh jejich kooperace je důležitým a netriviálním krokem při návrhu software. Návrhové principy GRASP (General Responsibility Assignment Software Patterns), které sestavil Craig Larman. Při návrhu software je důležité přidělování zodpovědností (kdo bude co dělat a co kdo ví) a návrhy jejich kooperace. Už princip SOLID definuje, že každá entita by měla mít jednu zodpovědnost. Zde je zodpovědnost dále rozváděna. Slouží jako pomůcka při rozhodování o přidělování zodpovědností (společný slovník pojmů). Jednotlivých 9 principů GRASP má různou úroveň abstraktnosti:

#### 1. Protected variations – chráněné změny

Identifikujte místa pravděpodobných změn a nestability a zodpovědnosti přiřaďte stabilním rozhraním, která kolem nich vytvoříte.

#### 2. High cohesion – vysoká soudržnost

Přiřazujte zodpovědnosti tak, aby zůstala zachována vysoká soudržnost (aby třída nedělala více nesouvisejících věcí).

#### 3. Low coupling – slabá provázanost

Zodpovědnosti přiřazujte tak, aby provázanost zůstala co nejmenší. (Pokud jsou prvky provázané, změny jednoho z nich ovlivňují i druhý.)

#### **4. Pure fabrication – pouhá konstrukce**

Pokud by přiřazení zodpovědnosti některé třídě představující objekt z problémové domény narušilo ostatní principy, je vhodné vytvořit novou třídu, která nepředstavuje nic z oblasti domény – třídu vytvořenou pouze jako konstrukční prvek pro zlepšení kvality návrhu.

#### **5. Polymorphism**

Pokud chování závisí na typu objektu (třídě), přiřaďte zodpovědnost za toto chování pomocí polymorfických metod třídě, na které toto chování závisí.

#### **6. Indirection – nepřímé vazby**

Pokud se potřebujete vyhnout přímé vazbě, přiřaďte zodpovědnost prostředníkovi, takže prvky nebudou muset být propojeny přímo.

#### **7. Information expert**

Zodpovědnost přiřaďte prvku, který má informace potřebné pro splnění této zodpovědnosti.

#### **8. Creator – tvůrce**

Přiřaďte třídě B zodpovědnost za vytváření instancí třídy A, pokud platí jedno nebo více z následujících:

- B je agregátor objektů A.
- B obsahuje objekty A.
- B uchovává záznamy o objektech A.
- B úzce spolupracuje s objekty A.
- B má inicializační data pro A (B je informační expert pro inicializaci A).

#### **9. Controller**

Zodpovědnost za zpracování systémových událostí přiřaďte speciální třídě nebo třídám, které nebudou součástí ani uživatelského rozhraní ani doménové vrstvy, ale budou sloužit jako rozhraní mezi nimi. [20]

## 2 APLIKACE DOSTUPNÉ NA TRHU

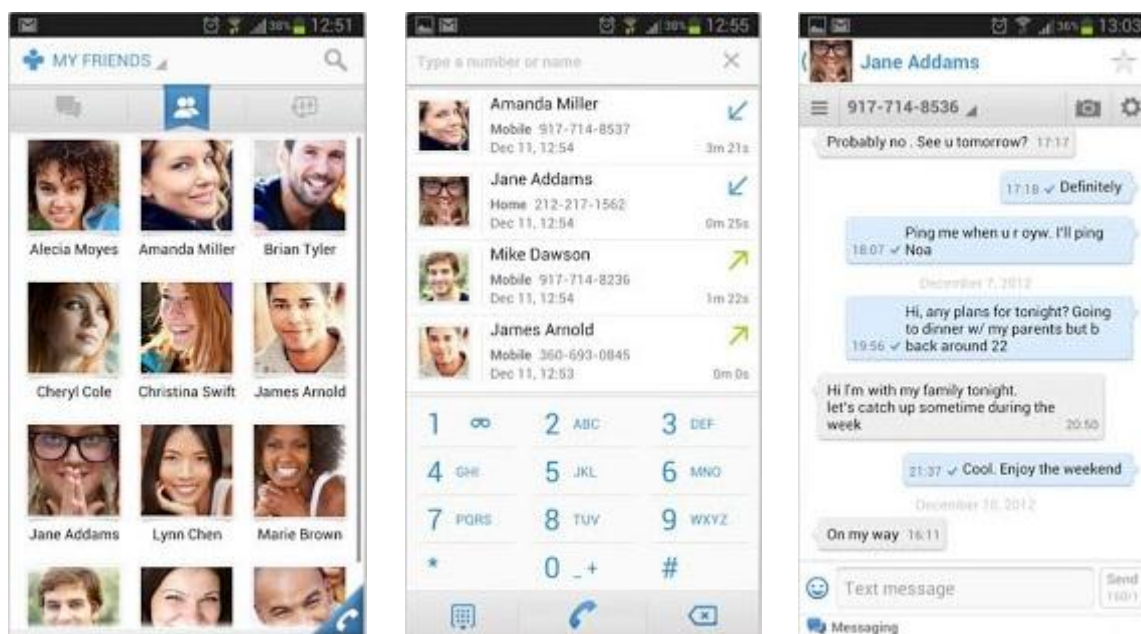
Přímá konkurence na trhu s aplikacemi pro OS Android nebyla nalezena. Kontaktní aplikace v chytrých telefonech jsou provedeny formou telefonního seznamu a kontakty jsou pak ukládány přímo v paměti telefonu. Tyto kontakty je pak možné synchronizovat například pomocí synchronizace účtu Google.

Jsou zde velmi omezené možnosti jak tyto kontakty sdílet s jinými uživateli a čím větší počet uživatelů, tím větší úskalí. Je obtížné reprezentovat vazby mezi společnostmi a jednateli.

Kontaktní aplikace nevytváří vlastní úložiště a jednotlivé kontakty jsou tak přítomny v systému telefonu. Pro některé uživatele to může znamenat problém – promíchání osobních a pracovních kontaktů.

Výhodou je plná integrace do systému, takže při příjmu hovoru je vidět jméno volajícího – společnosti nebo jednatele.

Jednou z těchto aplikací je **Contacts+**, která plně nahrazuje výchozí telefonní seznam v systému. Je graficky velmi zdařilá a přehledná, výchozí obrazovkou jsou kontakty, lze je zobrazit jako seznam nebo mřížku a také nastavit jejich řazení – abecední, dle četnosti nebo dle aktuálnosti.

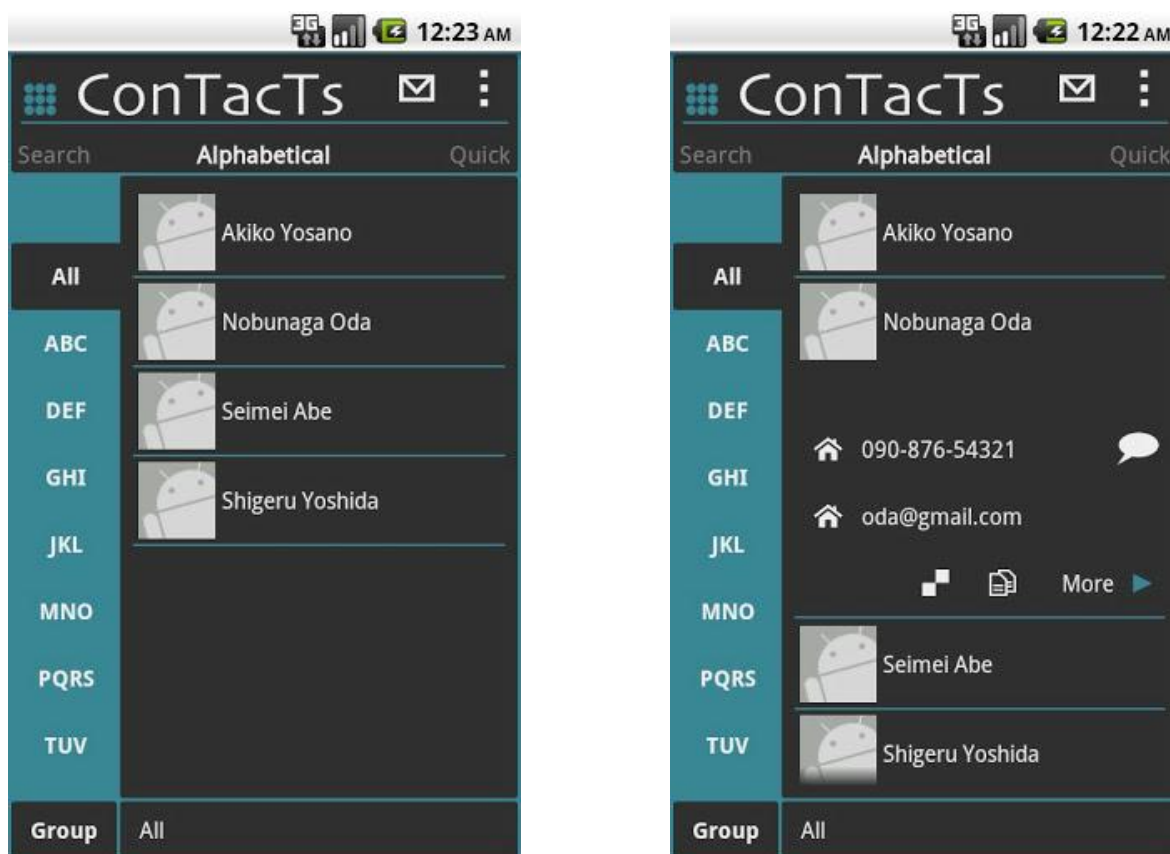


Obr. 3. Obrazovky Contacts+ (Zdroj: Google Play)

Další obrazovkou (na Obr. 3. uprostřed) je historie hovorů, kde lze vybráním položky v seznamu opětovně vytočit kontakt a jsou zde informace o hovorech jako čas a datum volání, délka trvání hovoru nebo jednalo-li se o příchozí, odchozí anebo nepřijatý hovor.

Poslední důležitou obrazovkou jsou SMS zprávy (na Obr. 3. vpravo). Zprávy se zobrazují přehledně v bublinách v rámci konverzace, u každé je čas jejího přijetí nebo odeslání. Mezi jednotlivými obrazovkami je možné přecházet tažením prstu, což usnadňuje navigaci v aplikaci.

Další podobnou aplikací je **ConTAcTs**, která je již graficky střídmější. Zobrazuje opět několik záložek, mezi kterými se dá přecházet tažením prstu. Výchozí záložkou je abecední seznam kontaktů, u kterého je možnost jednotlivé položky rozklepnout a zobrazit detailní kontaktní informace přímo na místě.



Obr. 4. Seznam kontaktů v aplikaci ConTAcTs (Zdroj: Google Play)

Druhou záložkou je seznam vybraných kontaktů – buď oblíbených anebo naposledy použitých. Zbylé záložky jsou vyhledávání a historie volání, která zobrazuje příchozí,

odchozí nebo nepřijaté hovory, délku jejich trvání a čas a datum uskutečnění. Aplikace poskytuje dobrou podporu skupin kontaktů a práce s nimi je snadná. Neumí však měnit data kontaktů a využívá k tomu výchozí systémové aplikace, tudíž není plnohodnotnou náhradou správy kontaktů.

Ani jedna ze zmíněných aplikací však nespĺňuje požadavky na synchronizaci kontaktů a bylo by nutné se spolehnout například na synchronizaci Google.

### 3 NÁVRH MOBILNÍ APLIKACE A WEBOVÉ SLUŽBY

Mobilní aplikace má pracovat jako firemní telefonní seznam s obchodními kontakty. Předmětem aplikace tedy jsou kontaktní informace. U společnosti jsou zásadní informace její název, adresa, telefonní a faxová čísla, emailové adresy a webové stránky. Každou společnost může zastupovat jeden nebo více jednatelů. U nich je nutné poskytovat jméno a příjmení, titul, postavení, telefonní a faxová čísla, emailové adresy. Uživatelé budou pomocí této aplikace kontaktovat dané společnosti nebo osoby, aplikace musí umožnit snadné použití předmětných informací, jako jsou telefonní čísla a emailové adresy.

Důležitým prvkem je automatické sdílení těchto kontaktů neboli synchronizace. Ta vyžaduje aplikaci dostupnou pro všechny používaná zařízení. Touto aplikací bude webová služba dostupná v síti Internet. Mobilní aplikace bude svá lokální data synchronizovat s touto službou, dojde-li ke změně dat, odesílá tyto služby ke zpracování a přijímá od ní data změněná na jiném zařízení. Data budou přenášena ve formátu JSON, který je jednodušší než formát XML a přesto stále poskytuje velmi dobrou podporu pro strukturovaná data.

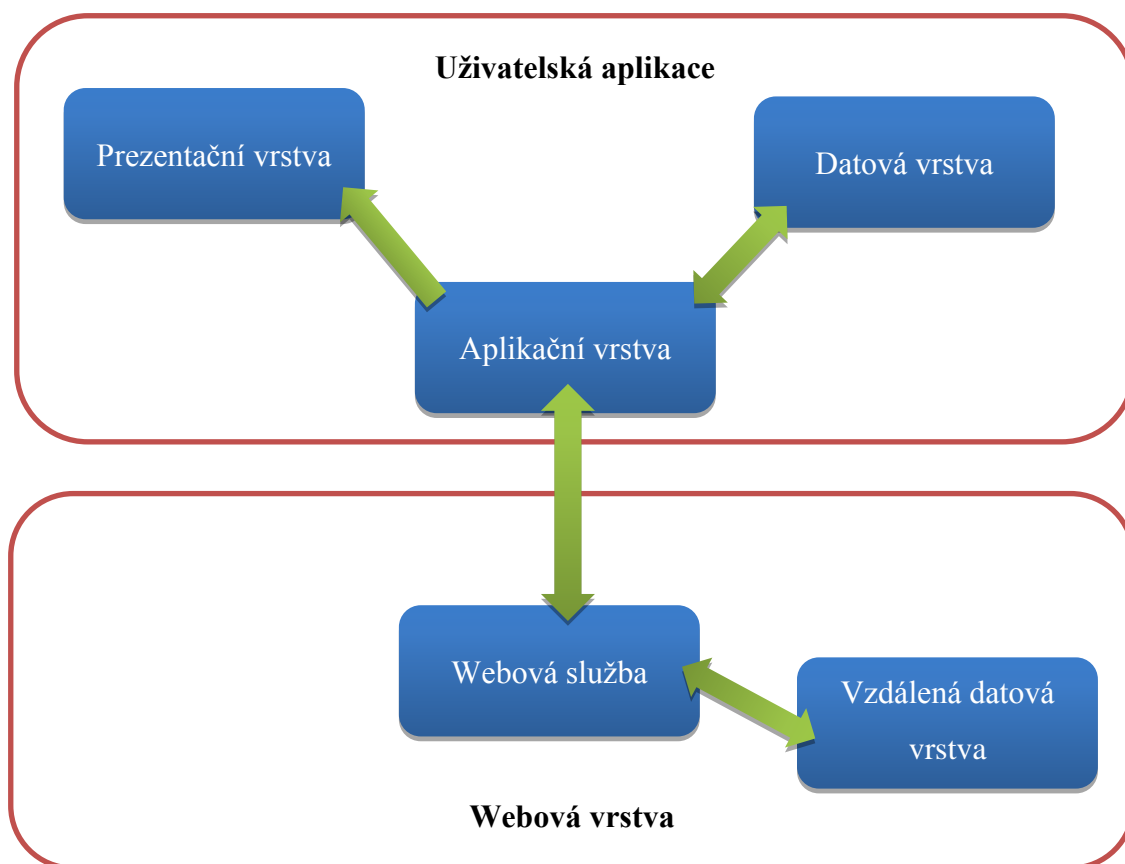
Obecnými požadavky jsou kompatibilita, jednoduchost, správnost a rychlost. Aplikace by bez problémů měla běžet na nejpoužívanějších verzích systému Android, to znamená od verze 2.3 po verzi 4.2. Jednoduché ovládání aplikace umožní snadnější přijetí uživateli, kteří se nebudou muset učit s aplikací pracovat.

Správnost činnosti aplikace znamená, že se aplikace chová, jak uživatel očekává a patří mezi doporučení společnosti Google k vývoji mobilních aplikací. Využívání ovládacích prvků k činnostem, pro které byly určeny jiné prvky, je pro uživatele velmi matoucí a může vést k chybám obsluhy. V takových případech nelze vinu svalovat jednoduše na uživatele, protože nemohl předpokládat, že například tlačítko *Zpět* se v aplikaci chová jinak než v jiných aplikacích.

Rychlost aplikace je pro uživatele taktéž důležitá, ovšem záleží také na četnosti použití dané funkce. Bude-li nejčastěji používaná funkce pomalá, bude to pro uživatele citelně horší, než pokud bude pomalá funkce, která je spouštěna jen sporadicky.

### 3.1 Návrh struktury

Mobilní aplikace využije vícevrstvé architektury, která je doporučena tvůrci operačního systému. Je to prezentační, aplikační a datová vrstva. Prezentační vrstva je viditelná uživatelům, prezentuje data v uživatelsky přívětivém formátu a je tvořena tzv. *Layouty* obsahující jednotlivé prvky *View*. Aplikační vrstva se stará o zabezpečení funkčnosti aplikace, všechny uživatelské vstupy jsou předány sem a zpracovány, výstupy do prezentační vrstvy vytváří právě vrstva aplikační. Tvoří ji třídy odvozené od třídy *Activity*. Datová vrstva je reprezentována datovými entitami (třídy) a dalšími třídami, které zprostředkují operace s daty nad úložištěm dat – databází. V případě lokálního úložiště na OS Android se jedná o databázi SQLite. Důležitou činností aplikace však má být synchronizace kontaktů, pak tedy bude důležité jak lokální datové úložiště, tak vzdálené úložiště – tj. databáze umístěná v síti Internet, která bude pro aplikaci dostupná při aktivním datovém připojení. Komunikační rozhraní bude tvořit webová služba a komunikace bude probíhat skrze HTTP požadavky pomocí formátu JSON.



Obr. 5. Diagram vrstev systému

## 3.2 Návrh mobilní aplikace

### 3.2.1 Návrh datové vrstvy

Na začátek je podstatné vhodně navrhnout datové struktury, se kterými bude aplikace pracovat. Až po jejich navržení je možné do detailu promyslet, jak se s daty bude zacházet a jak tedy bude potřeba aplikační logiku přizpůsobit. Datové entity budou reprezentovány jako jednotlivé třídy, což je ověřená programátorská technika. Těchto technik a doporučených pravidel se práce bude držet při celém návrhu aplikace, avšak při návrhu datových entit je jejich vliv nejvíce viditelný. Třídy budou navrženy jako tzv. „černé skříňky“, to znamená, že všechny jejich vlastnosti budou soukromé (neveřejné). Přístup k nim je umožněn přes přístupové metody, tzv. „getter a setter“, čímž je docíleno důkladné zapouzdření třídy. Návrh těchto „přístupových rozhraní“ je tedy důležitým aspektem.

Hlavním předmětem aplikace je firma a její kontaktní informace. Název třídy pro společnost zvolíme *Company*. Tento název je dostatečně vypovídající o obsahu třídy a jejím použití. Dalšími vhodnými názvy by mohly být například *CompanyContact*, *CompanyInfo* nebo *CompanyClass*.

Kontakt na společnost musí obsahovat její název, telefonní kontakt a email, jak bylo zjištěno při analýze zadání. Jak společnost, tak jednatel mají kontaktní telefonní a faxová čísla a emailové adresy. Zde by se vyplatilo nezahrnovat tyto informace do tříd společnosti a jednatele, ale využít určitou abstrakci. Jednou možností je vytvořit rozhraní *Contact*, které bude obsahovat dané vlastnosti a deklaráce metod. Nevýhodou však je, že každá z implementujících tříd by musela definovat každou metodu, tudíž metodu pro získání telefonního čísla by bylo nutné definovat vícekrát.

Druhou možností je vytvořit třídu *Contact* obdobně jako rozhraní, s tím rozdílem, že by již definovala přístupové metody. Třídy jednatele a společnosti by pak obsahovaly třídu *Contact* a umožnili k ní buď přímý přístup, nebo by implementovaly přístupové metody k metodám této třídy, čímž třídu *Contact* zakryjí a nebude navenek známo její použití.

V případě, že by se jednalo o jednoduché informace bez další nutnosti zpracování, byla by vhodnější možnost první. Avšak zde jednotlivé prvky budou obsahovat více hodnot oddělených středníkem. To je pro člověka méně čitelná informace, takže je vyžadováno

další zpracování. Proto je vhodnější použít druhý přístup. Výhody jsou zjevné, veškerá práce s těmito kontaktními údaji bude zapouzdřená ve třídě *Contact*, na jednom místě (vyhovuje doporučení principů DRY). To usnadní případné úpravy v budoucnosti, jako je například změna oddělovacího znaku. Oddělením logiky zpracování kontaktních údajů od logiky tříd *Executive* a *Company* bude redukována složitost – znamená to méně věcí, na které musí programátor při úpravách myslet.

Obdobně bude řešena adresa - třída *Address*. Bude obsahovat název ulice, město, stát a poštovní směrovací číslo. Adresu bude mít společnost i jednatel.

Třída *Executive* bude velmi podobná třídě *Company*, pouze s několika málo rozdíly. Nebude obsahovat popis a webovou stránku společnosti, ale jméno, titul a pozici jednatele. Ostatní prvky budou shodné. V prvotních verzích návrhu byly třídy podstatně více rozdílné a lišilo se i jejich zpracování. Postupným vývojem návrhu se však třídy přiblížili k sobě natolik, že by bylo velmi příhodné využít abstrakce a odvodit třídy od společného předka. Veškerá práce s nimi by se pak sjednotila využitím polymorfismu. Proti tomuto však stál již částečně implementovaný kód a jeho přepsání by bylo příliš časově náročné. V celém systému tak zůstávají dvojí programové části, plnicí velmi podobný účel. Je to jasným porušením pravidla DRY a poučením do budoucna.

Poslední datovou entitou bude třída *CERelation*, která představuje vazbu mezi společnostmi (*Company*) a jednatelem (*Executive*). Název třídy vznikl zkrácením názvu, který je příliš dlouhý – *CompanyExecutiveRelation*. Zde by bylo vhodné, aby jeden jednatel mohl zastupovat více společností nebo naopak jedna společnost mohla být zastupována více jednatelem. Z tohoto důvodu byla navržena tato třída, aby bylo možné vazby definovat libovolně. Obsahem této třídy budou unikátní identifikátory – jeden pro společnost a jeden pro jednatele. Tímto je vazba jasně definována.

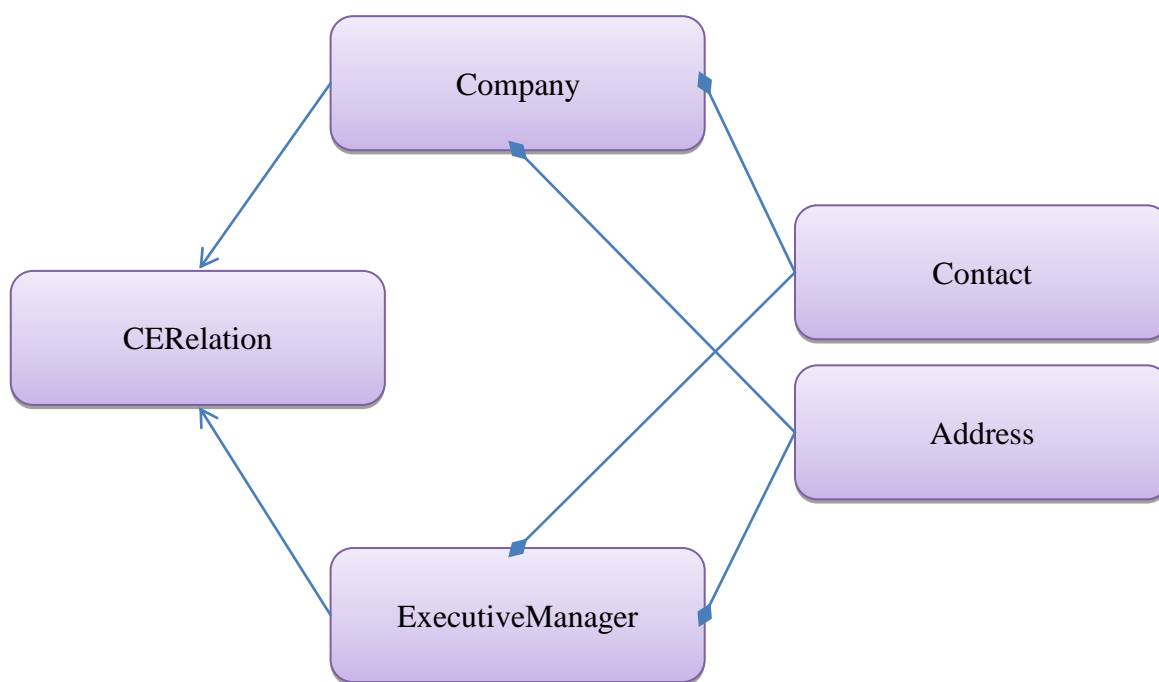
Aby se s entitami dalo lehce pracovat při uživatelských operacích, budou všechny zmíněné třídy obsahovat lokální identifikátory. Tato velká celá čísla (typ SQL BIGINT) jsou vytvářena v databázi jako hodnota automaticky iterujícího sloupce.

Vzhledem k tomu, že data budou synchronizována mezi více zařízeními, tj. mezi více samostatnými databázemi nedají se lokální identifikátory použít při synchronizaci a nebudou vůbec přenášeny. Každá entita bude obsahovat unikátní identifikátor napříč

systemem, který bude sloužit hlavně pro synchronizaci. Lokální identifikátor bude často v jednotlivých databázích rozdílný, avšak globální identifikátor je neměnný.

Alternativou tohoto přístupu je použití jediného globálního identifikátoru, pak je ovšem veškerá práce s datovou entitou v lokální databázi vázána na tento identifikátor, který je posloupností bytů, případně řetězcem a tedy ve vyhodnocování pomalejší než běžné číslo.

Datové entity budou dále obsahovat také datum vytvoření a datum poslední editace. Tyto data jsou důležitá, neboť rozhodnutí o tom, zda se má daná entita synchronizovat nebo ne se řídí právě jimi. Není zapotřebí neustále dokola posílat data, která nedoznala žádných změn. Další výhodou je právě rozpoznání dat, která byla modifikována. Není nutné porovnávat data navzájem mezi službou a mobilní aplikací, stačí se řídit podle data poslední modifikace.



Obr. 6. Diagram datových tříd

Po návrhu datových entit je možné přejít k návrhu jejich stálého úložiště. K tomu bude využita interní databáze, kterou poskytuje OS Android svým aplikacím. Jedná se o kompaktní relační databázi SQLite, která je implementována v jediné knihovně, takže nevyžaduje svůj vlastní proces pro práci s databází. SQLite splňuje téměř kompletně standard SQL92, avšak přesto jsou zde některé nedokonalosti – tou hlavní je absence

datových typů reprezentujících časové údaje. Z tohoto důvodu bude nutné je vždy převádět na řetězec při zápisu do databáze a při čtení převádět zpět na datum a čas.

Pro uložení dat aplikace vzniknou jednoduché čtyři tabulky:

- *companies* pro společnosti,
- *executives* pro jednatele,
- *addresses* pro adresy,
- *cerelations* pro vazby.

Správu jednotlivých datových položek budou mít na starosti správčovské třídy (například *CompanyManager*). Každá tato třída zastřeší správu jedné datové tabulky v databázi. Bude se starat o vytvoření tabulky, vkládání, úpravu, mazání a získávání záznamů z tabulky. Tyto základní CRUD operace budou doplněny o další jim podobné, které usnadní práci s daty použitím specializovaných dotazů do databáze – například získání všech jednatelů pro danou společnost.

Bude vytvořena báze abstraktní třída *BaseManager*, od které se odvodí všechny ostatní správčovské třídy. Pro navržené entity tedy vzniknou:

- *CompanyManager*
- *ExecutiveManager*
- *AddressManager*
- *CERelationManager*

Třída *Contact* nebude mít vlastní správčovskou třídu, protože v databázi nebude reprezentována samostatnou tabulkou. Kontaktní data budou ukládána přímo do tabulek společností a jednatelů a to ve formátu hodnot oddělených středníky, jež je variantou formátu CSV (používá ji například MS Excel v české verzi).

### 3.2.2 Návrh uživatelského rozhraní (UI) a aplikační vrstvy

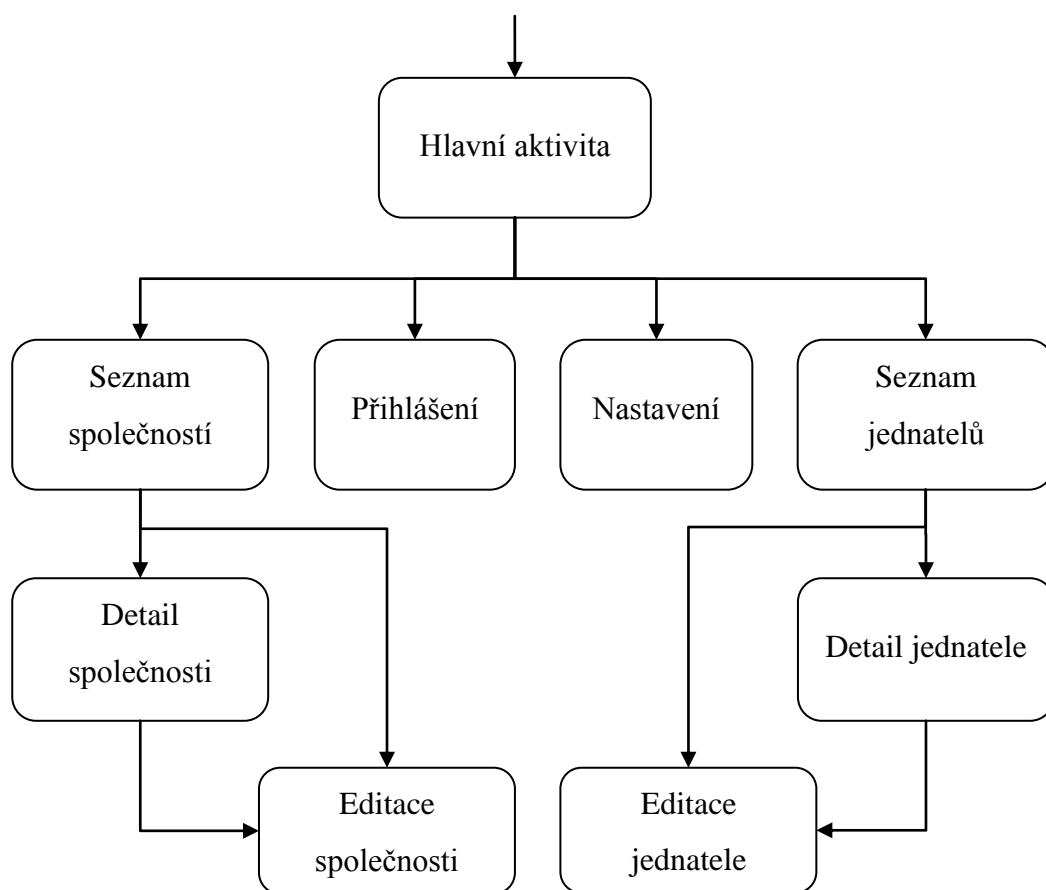
Při návrhu uživatelského rozhraní je kladen důraz na přehledné, jednoduché a intuitivní ovládání. Uživatelská pozornost nesmí být odváděna nedůležitými informacemi, naopak požadované informace mají být uživateli prezentovány v jasně srozumitelné formě.

Aplikace využije systémové komponenty, pro zachování systémového vzhledu, na který jsou uživatelé zvyklí. Uživatelé budou tak při práci s aplikací jistější a sníží se pravděpodobnost chyb obsluhy. Bude-li zapotřebí vytvořit vlastní komponentu, pak je na místě snaha o dosažení systémového vzhledu a funkce.

Pro ovládání aplikace bude využito pokročilých možností dotykových displejů a tím zjednodušeny některé operace, například zobrazení detailu tažením po položce v seznamu. Použití takového způsobu ovládání však nesmí být jedinou možností a uživateli musí být dána jistá volnost. Nebude po něm vyžadováno ovládání složitých gest nebo „střefování prstem“ na miniaturní tlačítka. Návrh se řídí doporučením:

„Udělej aplikaci tak, aby se dala ovládat nosem.“

- Matěj Konečný (Zdrojak.cz)



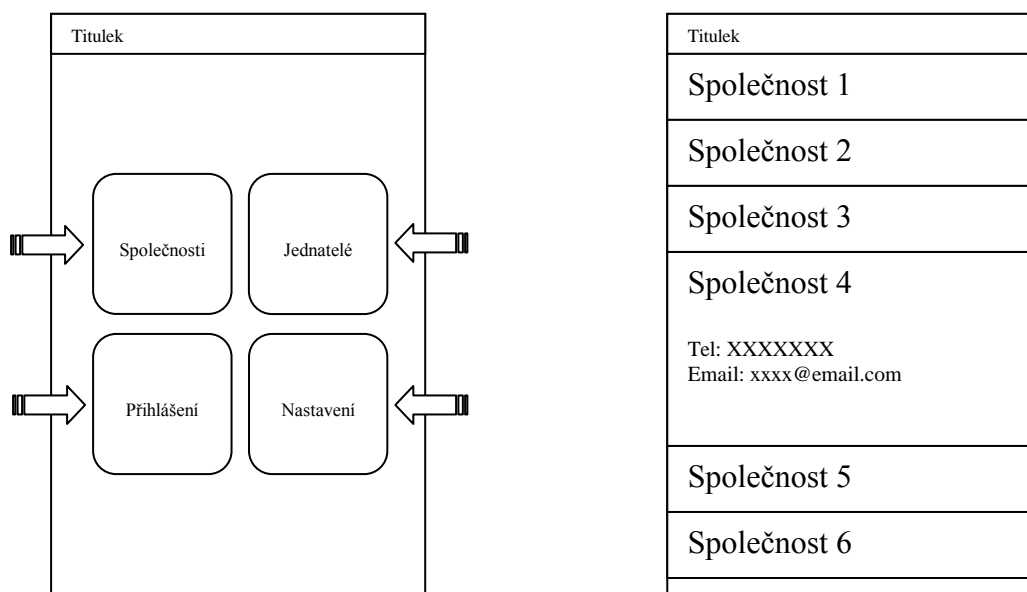
Obr. 7. Diagram aktivit

Základním stavebním prvkem mobilní aplikace na platformě Android je aktivita neboli třída odvozená od třídy *Activity*. Každá tato aktivita je součástí aplikační vrstvy, protože

zpracovávají všechny události, které v aplikaci mohou nastat. Do prezentační vrstvy patří obrazovky definované v souborech XML, které se nazývají *Layout*. Aktivita při svém spuštění definuje, jaký *Layout* ji bude reprezentovat, z tohoto důvodu probíhá návrh aplikační a prezentační vrstvy dohromady.

### Hlavní aktivita

Po spuštění aplikace naběhne hlavní obrazovka, jedná se o rozcestník se čtyřmi velkými tlačítky, která budou animována přiletem ze stran. Animace by měla navodit pocit dynamičnosti aplikace, být jednoduchá a rychlá, aby neobtěžovala uživatele. Měla by stáhnout pozornost uživatele na tato tlačítka. Prvním tlačítkem aplikace přejde na seznam společností, druhým na seznam jednatelů. Třetím tlačítkem se spustí akce přihlášení uživatele. Pokud není přihlášení vyžadováno, nemusí se jím uživatel nijak zabývat. Poslední tlačítko otevře nastavení aplikace.



Obr. 8. Wireframe hlavní obrazovky (vlevo) a seznamu společností (vpravo)

### Aktivita seznamu společností

Zde bude zobrazen seznam společností přes celou obrazovku, v každé položce bude název společnosti. Při klepnutí na položku se tato rozevře a zobrazí další informace o společnosti (viz Obr. 8). Tažením po položce seznamu zleva doprava se přejde na aktivitu detailu společnosti, tažením zprava doleva bude spuštěna aplikace volání s předvyplněným

telefonním číslem společnosti. Dlouhým podržením na položce se vyvolá kontextové menu, ve kterém budou následující možnosti – zobrazení detailu společnosti, spuštění editace společnosti a smazání společnosti.

Hlavní menu na této obrazovce, vyvolávané tlačítkem *Menu* na zařízení, obsahuje pouze položku přidání nové společnosti.

### **Aktivita detailu společnosti**

Přehledně zobrazí detail společnosti rozdělenou na sekce. V horní části bude zobrazen název a popis společnosti, níže bude adresa. Pod ní budou následovat telefonní a faxová čísla, emailové adresy a webová adresa stránek společnosti. Na to dole naváže seznam jednatelů společnosti.

Klepnutím na kontaktní informace se vyvolá příslušná akce, například již zmíněné spuštění aplikace volání nebo aplikace pro elektronickou poštu.

V hlavním menu budou položky pro spuštění editace společnosti, smazání společnosti, přidání dalšího jednatele a tři položky pro nastavení výchozího telefonního a faxového čísla a emailové adresy. Tyto výchozí kontaktní informace budou zobrazeny v rozšířeném popisu položky v seznamu společností a použity pro rychlé akce, například ono volání při tažení prstem.

Při zvolení položky v seznamu jednatelů dané společnosti v dolní části obrazovky, dojde k přechodu na aktivitu detailu vybraného jednatele. Podržením položky v seznamu bude vyvoláno kontextové menu, ve kterém budou možnosti úpravy jednatele a jeho odebrání – čímž bude zrušena vazba mezi daným jednatelem a společností.

### **Aktivita úpravy společnosti**

Zobrazí informace o společnosti v podobném rozložení jako detail, ale s možností úpravy. Je možno měnit informace, přidávat a odebírat telefonní a faxová čísla a emailové adresy. Změny budou rovnou ukládány do databáze.

Detail	Editace
Název společnosti	Název: <u>Společnost</u>
<b>Popis</b> Popis společnosti	Popis: <u>Popis společnosti</u>
<b>Adresa</b> Ulice: Náhodná 5984 Město: Zlín Stát: Česká republika PSČ: 76001	<b>Adresa</b> Ulice: <u>Náhodná 5984</u> Město: <u>Zlín</u> Stát: <u>Česká republika</u> PSČ: <u>76001</u>
<b>Kontaktní informace</b> Tel: Fax: Email:	<b>Kontaktní informace</b> Tel: _____ Fax: _____ Email: _____
<b>Jednatelé</b> Jednatel 1 Jednatel 2 Jednatel 3	<input type="button" value="Uložit"/> <input type="button" value="Zrušit"/>

Obr. 9. Wireframe obrazovek detailu a editace

### Aktivity jednatelů

Aktivity jednatelů budou téměř totožné s aktivitami společností, liší se pouze zobrazovanými daty. Činnosti v jednotlivých obrazovkách budou shodné.

### Aktivita přihlášení

Zobrazí jednoduchý formulář pro zadání uživatelského jména a hesla. Při úspěšné autentifikaci bude aplikace používat pro komunikaci se službou pouze těchto přihlašovacích údajů. Po zadání údajů se aplikace vrátí na hlavní obrazovku.

### Aktivita nastavení

Zobrazí obrazovku s různými nastaveními aplikace. Bude zde například možnost automatického přihlášení nebo zakázání synchronizace.

### 3.2.3 Synchronizační služba

Pro synchronizaci dat bude sloužit synchronizační služba, běžící v pozadí. OS Android nabízí více typů služeb. Pro potřeby aplikace nejvíce vyhovuje služba typu *IntentService*, tvoří ji třída odvozená právě od třídy *IntentService*. Tato služba je spouštěna s nějakým záměrem (anglicky intent) a po jeho splnění se automaticky ukončí.

Ve své pracovní metodě bude komunikovat s webovým rozhraním a synchronizuje s ním lokální data – přijme aktualizovaná data a odešle data lokálně změněná.

Podstatným rozhodnutím je, jak danou službu spouštět. Nejsnadnějším řešením je službu spouštět přímo při startu aplikace. Pak nastává problém v tom, že uživatel nemusí hned vidět aktuální data, bude-li synchronizace trvat nějaký čas.

Ideálním signálem ke spuštění služby je připojení zařízení k Internetu. V takovém případě je velká šance, že jsou data aktualizována ještě dříve, než dojde ke spuštění aplikace. Ani toto řešení však není bez problémů a to když bude zařízení neustále připojeno k Internetu. Událost změny připojení k síti se pak téměř nebude vyskytovat (pouze v případě výpadku signálu). Je tedy žádoucí oba způsoby zkombinovat a při startu aplikace zkontrolovat, zda-li došlo v nedávné době k synchronizaci dat. Další možností je provádět synchronizaci pomocí systémového časovače, například každou hodinu, nebo každý den.

Synchronizace dat bude prováděna dle data poslední úpravy. Platná data jsou ta, která byla editována poslední a nahrazují tak data neplatná.

Pro spuštění služby v reakci na změnu stavu připojení je zapotřebí posluchač systémových událostí. Je to opět třída odvozená od třídy *BroadcastReceiver*. Posluchač událostí zkontroluje, je-li k dispozici internetové připojení, a pokud ano, vydá signál ke spuštění služby. Posluchače událostí i službu bude nutné zavést do manifestu aplikace a specifikovat na jaké události bude posluchač reagovat.

### 3.3 Návrh webového rozhraní

Webové rozhraní musí být dostupné pokud možno odkudkoli, to docela dobře splňuje webová služba, která je zpřístupněna po síti Internet. Zařízení s mobilní aplikací tak budou mít možnost synchronizovat data všude tam, kde budou mít přístup na Internet.

Specifickým požadavkem je využití formátu JSON k přenosu dat mezi aplikací a webovou službou. Toho lze snadno dosáhnout použitím RESTful služby, která není orientována pouze na formát XML jako služba SOAP, ale umožňuje používat různé formáty, mezi nimi také JSON. RESTful služba je stavěna na práci se zdroji, všechny zdroje mají svůj identifikátor URI. Ke zdroji se přistupuje HTTP požadavky typu GET, POST, PUT nebo DELETE. Tyto požadavky vypovídají o činnosti se zdrojem, požadavkem GET se zdroj

získá, požadavkem POST se zdroj vytváří, DELETE slouží k odstranění zdroje a PUT funguje jako nahrazení (případně vložení) zdroje.

### 3.3.1 Návrh datové vrstvy

V databázi budou vytvořeny podobné čtyři tabulky jako v lokální databázi mobilní aplikace, ale zde bude již možné využít plnohodnotné datové typy. Nebude tedy nutno časové údaje převádět na řetězce. Datové entity budou zde opět třídy velmi podobné třídám z mobilní aplikace, nebude ale implementována třída *Contact*, neboť zde není potřeba. Kontaktní informace ve službě nebudou nijak zpracovávány, ale pouze přijaty a zapsány do databáze nebo naopak přečteny z databáze a odeslány mobilní aplikaci.

Každou databázovou tabulku bude spravovat k tomu určená třída. Tyto správcovské třídy budou obdobou tříd z mobilní aplikace a budou přizpůsobené pro práci s MySQL databází. Budou mít na starosti všechny potřebné operace s daty nad databází.

Nebude docházet k nevratnému smazání dat, data budou pouze deaktivována (označena příznakem v tabulce databáze) a neaktivní data již dále nebudou viditelná. K jejich synchronizaci ale přesto může dojít, pokud si je mobilní aplikace vyžádá – tím pak budou data smazána z lokální databáze aplikace.

### 3.3.2 Návrh komunikačního rozhraní služby

Jednotlivé datové entity budou představovat zdroje RESTful služby. Zapotřebí jsou pouze dvě činnosti – získání sady upravených dat jiným uživatelem a odeslání sady lokálně upravených dat, takže stačí požadavky typu GET (pro získání) a POST (pro odeslání dat). Pro každou datovou entitu je tedy potřeba dvou veřejných metod, které obstarají dané funkce. Komunikace je bezstavová, to znamená, že každý požadavek na službu musí obsahovat všechny potřebné informace k jeho obsloužení. Na začátku komunikace si musí klient vyžádat komunikační token, který použije k dalšímu požadavku. Bez daného tokenu mu bude komunikace odepřena. Je-li vyžadováno přihlášení uživatelů, musí klient při žádosti o token přidat i uživatelské přihlašovací údaje. Token klientovi nebude přidělen, pokud údaje nesouhlasí. Více o zabezpečení komunikace lze nalézt v kapitole o bezpečnosti aplikace. Při komunikaci budou řídicí informace přenášeny v hlavičce HTTP požadavku nebo odpovědi. Data budou převedena do formátu JSON při jeho tvorbě.

## **II. PRAKTICKÁ ČÁST**

## 4 IMPLEMENTACE

Pro tvorbu mobilní aplikace je použito vývojové prostředí Eclipse IDE, které bylo společností Google vybráno jako výchozí prostředí pro vývoj aplikací na platformě Android a je dostupné zcela zdarma.

Taktéž vývoj webového rozhraní probíhal v prostředí Eclipse, konkrétně ve verzi Enterprise (pro Javu EE). Webové rozhraní vyžaduje pro svůj běh aplikační server, kterých pro tento účel existuje několik (JBoss, Tomcat, GlassFish, a další). Zvolen byl Apache Tomcat, jež poskytuje kompletní řešení, zároveň je jednoduchý a snadno použitelný v prostředí Eclipse. Aplikační server Apache Tomcat je taktéž dostupný zdarma.

### 4.1 Implementace mobilní aplikace

#### 4.1.1 Datová vrstva

Pro reprezentaci dat jsou vytvořeny třídy *Company*, *Executive*, *Address*, *Contact* a *CERelation* podle návrhu. Jako lokální identifikátor bylo zvoleno dlouhé celé číslo – typ *Long*. Globální identifikátory budou reprezentovány instancemi třídy *UUID*, která je součástí balíčku *java.util*. Počet kombinací, které může tento objekt nabývat je  $2^{61}$ , takže lze předpokládat, že ke kolizi těchto identifikátorů za celou dobu provozu aplikace nedojde.

#### Třída *Address*

Reprezentuje adresu společnosti nebo jednatele. Prvky této třídy slouží pro uložení dat adresy, její jedinečné identifikaci v lokální databázi i globálně v celém systému. Jednotlivé prvky odpovídají sloupcům tabulky v databázi a jsou všechny soukromé. Z důvodu kvalitního zapouzdření třídy tvoří její vnější rozhraní metody pro získání a nastavení konkrétního prvku – tzv. *getter* a *setter*. Třída obsahuje jeden bezparametrický konstruktor, který inicializuje všechny prvky třídy a předchází tím chybám.

Třída překrývá několik metod, tou první je metoda *equals(Object)* zděděná od *Object*, její implementace zde porovnává instance *Address* člen po členu a jsou-li shodné, vrací *true*. Touto metodou lze snadno zjistit, obsahují-li instance stejná data.

Další překrytou metodou je *toString()*, která nyní vrací data adresy zformátovaná pro přímé zobrazení uživateli. Poslední překrytou metodou je metoda *clone()*, pro využití této metody

je však potřeba implementovat rozhraní *Cloneable*. Tato metoda vytvoří hlubokou kopii aktuální instance.

Třída implementuje jednu vlastní metodu a tou je *isEmpty()*, která kontroluje vnitřní datové prvky třídy a vrací hodnotu *true*, jsou-li prázdné, tedy reprezentují-li prázdnou adresu. V opačném případě vrací *false*. Význam je hlavně ve zjednodušení kontroly při práci se třídou. Je-li potřeba uložit do databáze adresu, lze jednoduchým voláním této metody zjistit, jestli instance opravdu obsahuje nějaká data.

### **Třída *Contact***

Tato třída obsahuje telefonní čísla, faxová čísla a emailové adresy. Jejím hlavním úkolem je poskytovat tato kontaktní data v přívětivém tvaru. Jejími prvky jsou privátní seznamy řetězců (*ArrayList<String>*), jeden pro každý typ dat. Důležité jsou zde metody, které třída poskytuje. Pomocí trojice metod *parsePhones(String)*, *parseFaxes(String)* a *parseEmails(String)* přečte z řetězce hodnot oddělených středníky jednotlivé hodnoty a ty si uloží do svých interních seznamů. Přesně opačnou funkci zastávají metody *getPhonesToString()*, *getFaxesToString()*, a *getEmailsToString()*, pomocí kterých jsou interní data převedena a vrácena jako řetězec hodnot oddělených středníky.

*Contact* taktéž překrývá metodu *equals(Object)* pro porovnání instancí podle obsahu a metodu *clone()* pro vytvoření hluboké kopie a tedy implementuje rozhraní *Cloneable*.

Interní data třídy jsou k dispozici přes přístupové metody, je možno získat celý seznam, jedinou položku anebo výchozí položku, kterou je první položka v seznamu. Vzhledem k této funkcionalitě jsou potřeba i metody pro nastavení výchozí položky – *setDefaultPhone(String)*, *setDefaultFax(String)* a *setDefaultEmail(String)*.

### **Třída *Company***

Reprezentuje společnost, tedy její kontaktní informace. Obsahuje instanci tříd *Address* a *Contact*. Výchozí telefonní, faxová čísla a email zpřístupňuje přímo pomocí metod *getDefaultPhone()*, *getDefaultFax()*, a *getDefaultEmail()*, jinak jsou data těchto instancí zpřístupněna přes referenci instance – využitím metod *getContact()* a *getAddress()*.

Třída překrývá metodu *equals(Object)*, ve které využívá metod *equals(Object)* instancí tříd *Address* i *Contact*. Implementuje rozhraní *Cloneable* a metodu *clone()* pro vytvoření hluboké kopie.

### **Třída *Executive***

Jak již bylo zmíněno v návrhu, je tato třída velmi podobná *Company*, co se jejího obsahu týče. Postupnou revizí návrhu bylo dosaženo rozhodnutí, že třída *Executive* bude obsahovat taktéž adresu a využívat kompletně třídu *Contact*. Rozdílem tak zůstaly pouze prvky jména jednatele, jeho titul a pozice ve společnosti. Stejně jako *Company* i tato třída implementuje rozhraní *Cloneable* a překrývá metodu *clone()* pro vytvoření hluboké kopie. Překrývá také metodu *equals(Object)*, díky které lze snadno porovnat dvě instance třídy *Executive* a zjistit obsahují-li shodná data.

Navíc oproti *Company* je zde statická metoda *formatName(Executive)*, která vrací jméno zadaného jednatele v lidsky čitelné podobě, tedy naformátováno jako *Příjmení Jméno, Titul*.

### **Třída *CERelation***

Další datovou třídou je *CERelation*, která reprezentuje vazbu mezi jednatelem a společností. Podle návrhu má tato třída obsahovat identifikátory společnosti a jednatele. Nemůže se však jednat o lokální identifikátory, neboť by při synchronizaci byly nepoužitelné. Naskytuje se několik možných řešení:

- a) Při odesílání dat službě převést lokální identifikátory na globální a opačně při příjmu.
- b) Používat v této třídě jen globální identifikátory.
- c) Používat lokální i globální identifikátory, při synchronizaci - lokální vypustit při odesílání a dohledat při příjmu.

Zvolena byla poslední možnost, která umožňuje jednoduché použití lokálních identifikátorů a zároveň znamená nejmenší změny v návrhu. Třída tedy obsahuje identifikátory lokální pro práci v mobilní aplikaci a při synchronizaci jsou využity identifikátory globální.

### **Třída *DbHelper***

Na začátek je vytvořena třída *DbHelper* odvozením od třídy *SQLiteOpenHelper*, aby zastřešila přístup k databázi. Definuje konstanty jména databáze a její verze, které ve svém konstruktoru předává konstrukturu rodičovské třídy:

```
public DBHelper(Context context) {  
    super(context, DB_NAME, null, DB_VERSION);  
    this.context = context;  
}
```

Parametrem tohoto konstrukturu je instance třídy *Context*, která definuje aktuální kontext, ze kterého je konstruktor volán. Třída *DbHelper* překrývá dvě metody rodičovské třídy, jsou to *onCreate(SQLiteDatabase)* a *onUpgrade(SQLiteDatabase, int, int)*. První ze jmenovaných se stará o vytvoření databáze – v tomto případě zavolá metody *createTable(SQLiteDatabase)* od všech správcovských tříd.

### **Třída *BaseManager***

Po implementaci datových tříd jsou vytvořeny správcovské třídy pro práci s databází. Jako jejich základ byla navržena třída *BaseManager*. Je koncipována jako abstraktní třída, aby mohla definovat několik vlastní metod, které zdědí dceřiné třídy a zároveň deklarovat metody, které dceřiné třídy musejí implementovat. Povinné metody jsou *getTableName()* a *getColNames()*, které vrací jméno tabulky databáze a pole jmen sloupců tabulky.

Dále tato třída definuje konstanty, které jsou společné pro všechny správcovské třídy, tyto konstanty představují názvy sloupců v databázi a jsou použity pro jakýkoli SQL dotaz. Tím je zaručena jednoznačnost a omezena možnost překlepu, neboť názvy sloupců v řetězci dotazu je možné snadno splést.

### **Třída *AddressManager***

Definuje konstantami název tabulky v databázi a názvy jednotlivých sloupců. Pomocí těchto konstant vytváří SQL dotazy, kterými pracuje s databází. Obsahuje třídu *DbHelper*, aby mohla získat komunikační rozhraní s databází – instanci třídy *SQLiteDatabase*.

Jsou zde definovány dvě statické metody *createTable(SQLiteDatabase)* pro vytvoření tabulky a *dropTable(SQLiteDatabase)* pro smazání tabulky. Ty jsou využity při vytváření nebo upgradu databáze.

Povinně implementovaná metoda *getColNames()* vrací pole vytvořené z konstant názvů sloupců. To využívají metody třídy *SQLiteDatabase*, při generování SQL dotazů, ale je to výhodné i při ruční tvorbě dotazu. Výsledky dotazů typu *SELECT* jsou vráceny v instanci třídy *Cursor*. Proto je zde přidána soukromá metoda *cursorToAddress(Cursor)*, která projde instanci *Cursor* a vyčte z ní data právě pomocí konstant názvů sloupců. Při

získávání adres z databáze tedy implementované metody vracejí vždy objekty *Address*, případně jejich seznam.

Protože je vhodné, aby se adresa z databáze získávala zároveň s jednatelem nebo společností, jsou v této třídě mechanismy, které umožní připojení tabulky v SQL dotazech klauzulí JOIN v jiných třídách. Jsou zde definovány konstantami aliasy sloupců, aby nedocházelo ke konfliktům při spojení tabulek, tyto přejmenované sloupce vrací metoda *getAliasedColNames()*. To vše podpoří metoda *cursorToAddressAliased(Cursor)*, která z instance čte data pomocí aliasů sloupců.

Díky tomuto přístupu se nemusí ostatní správcovské třídy, které třídu *AddressManager* používají, zabývat tím jaké sloupce definuje a jak se jmenují. Při případné změně názvu sloupce anebo přidání dalšího sloupce není nutné měnit nic v jiných třídách.

### **Třída *CompanyManager***

Opět jsou zde definovány konstanty pro názvy sloupců, zde však již není potřeba definovat aliasy, protože nejsou využity.

Důležitá je metoda *cursorToCompany(Cursor)*, která čte data společnosti z objektu *Cursor*. Je volána ve všech metodách využívající dotazů typu *SELECT*.

Tato třída má na starosti kompletní práci s objekty *Company* nad databází. Pro získání společnosti používá možností třídy *AddressManager*. V metodách *getCompaniesForExecutive(long)* a *getNotAssignedCompanies(long)* je navíc využívána i třída *CERelationManager* pro připojení tabulky vazeb. Je tak dosaženo vyčtení všech společností zastupovaných daným jednatelem v jediném dotazu.

K synchronizaci slouží metody *getCountUpdatedSince(Date)* a *getCompaniesUpdatedSince(Date)*, které získávají společnosti editovány od určeného data pro následné odeslání webové službě.

### **Třída *ExecutiveManager***

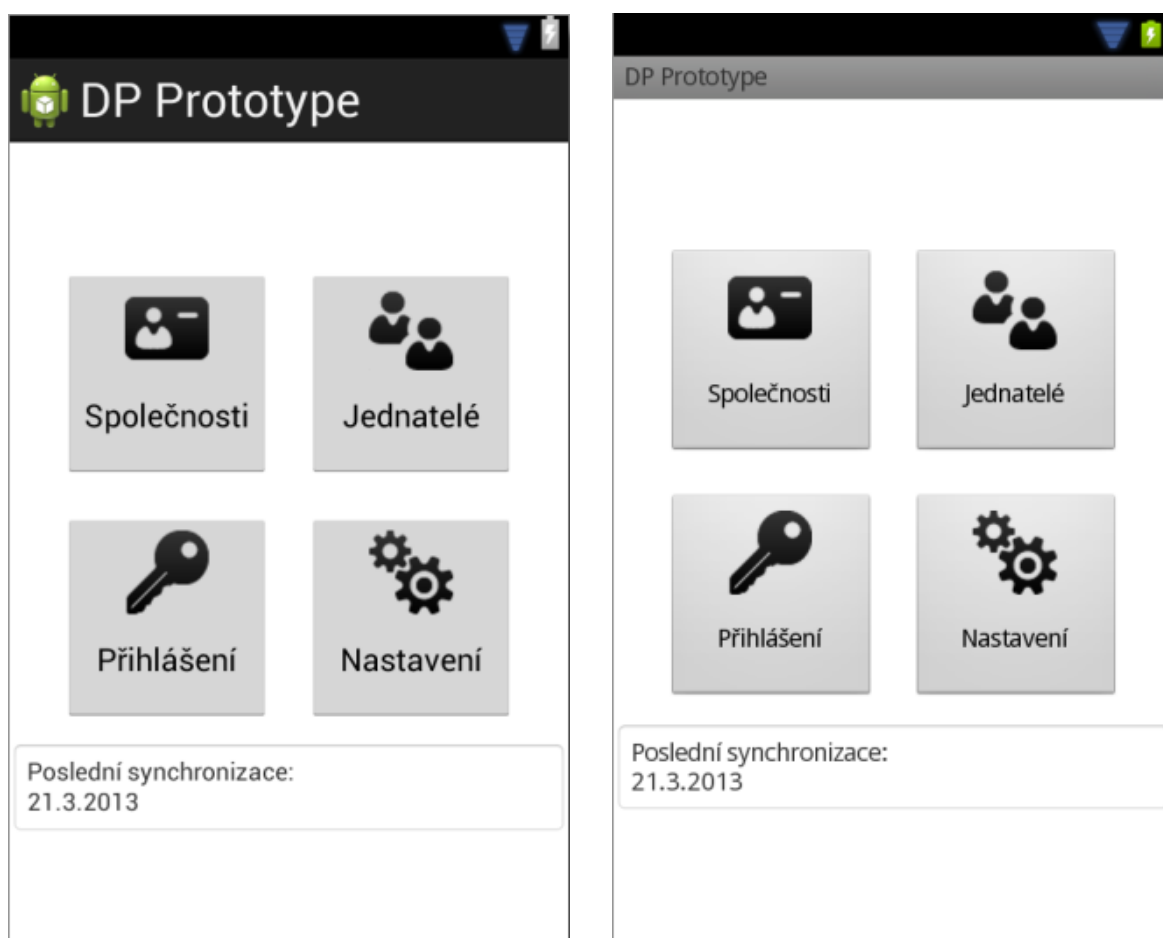
Zastřešuje správu jednatelů, odvozuje se od *BaseManager*. Její implementace je velmi podobná *CompanyManager*. Znovu je použit *AddressManager* pro získání jednatele i s adresou dohromady jedním SQL dotazem a *CERelationManager* pro využití tabulky vazeb. Podstatné jsou zde metody pro synchronizaci *getCountUpdatedSince(Date)* a *getExecutivesUpdatedSince(Date)*.

### Třída *CERelationManager*

Jsou zde definovány názvy sloupců a názvu tabulky v databázi. Převod z objektu *Cursor* na objekt *CERelation* zajišťuje metoda *cursorToRelation(Cursor)*. Není zde potřeba převod pomocí aliasů sloupců, neboť tabulka je využívána jenom jako vazební a samotná data z ní nejsou čtena při použití jinými manažery. Při získávání vazby přímo jsou použity běžné názvy sloupců.

#### 4.1.2 Uživatelské rozhraní – prezentační vrstva

Implementace uživatelského rozhraní se zabývá tvorbou jednotlivých obrazovek, tzv. layouts, a byly vytvořeny v editoru layoutů v prostředí Eclipse. To umožňuje přímé zobrazení layoutu v prostředí Android zvolené verze. Lze tedy sledovat změny, ke kterým v zobrazení dojde při použití různých verzích. Obrázek níže ukazuje hlavní obrazovku aplikace zobrazenou v Androidu 4.2 (vlevo) a starším Androidu 2.3 (vpravo).



Obr. 10. Hlavní obrazovka aplikace

## Hlavní obrazovka

Layout vychází z lineárního rozložení (*LinearLayout*) s vertikálním uspořádáním a centrováním na střed. Jsou vloženy další lineární rozložení, které již obsahují tlačítka a mají horizontální uspořádání s centrováním na střed. Tímto je dosaženo vycentrování všech tlačítek ke středu obrazovky libovolné velikosti. Zároveň to umožňuje použít animaci příletu tlačítek ze stran, protože rodičovské prvky jsou roztaženy až ke krajům obrazovky. Animovaný prvek se totiž nezobrazuje korektně při animaci mimo svůj rodičovský prvek.

## Detail společnosti, detail jednatele

Layouty jsou oba založeny na lineárním rozvržení s vertikálním uspořádáním. V horní části je vložen šedý pruh s názvem společnosti (případně jménem jednatele). Pod tímto pruhem je rolovatelný obsah použitím prvku *ScrollView*. Jsou zde zobrazeny textové informace v prvcích *TextView*, každý oddíl informací je uvozen nadpisem. Tento nadpis byl navržen jako text s vodorovnou linkou pod ním. Prvkek *TextView* však takové možnosti nenabízí a tak byla vytvořena vlastní komponenta *LabelView*, která obsahuje prvek *TextView* pro zobrazení textu a pod ním umístěný prvek *View* tvořící onu vodorovnou linku.

Telefonní a faxová čísla a emailové adresy jsou zobrazeny použitím další vlastní komponenty. Při použití například deseti telefonních čísel by se obrazovka roztahovala na délku a tak by uživatel musel příliš rolovat. Proto je využito komponenty *ContactItemDetailView*, která zobrazí pouze první položku a zbytek schová. Na pravé straně zobrazí tlačítko, kterým lze rozklepnout zobrazení a vidět tak zbylé položky.

## Editace společnosti a jednatele

Opět se jedná o lineární rozložení s vertikálním uspořádáním. Zobrazuje data společnosti nebo jednatele předvyplněné do editovacích polí *EditText*. I zde je použit prvek *LabelView* pro uvození sekcí dat. Na spodní straně obrazovky je umístěn plovoucí panel se dvěma tlačítky, který zůstává stále viditelný. Tlačítka nabízí možnost uložit nebo zrušit změny.

### 4.1.3 Aplikační vrstva

Implementace aplikační vrstvy probíhala shora dolů po částech. Bylo zde nalezeno několik problémů, které vyřešila tvorba vlastních komponent. Vytvoření vlastní komponenty není na platformě Android nic složitějšího, většinou se jedná o prosté odvození od existující komponenty.

### **Třída *MainActivity***

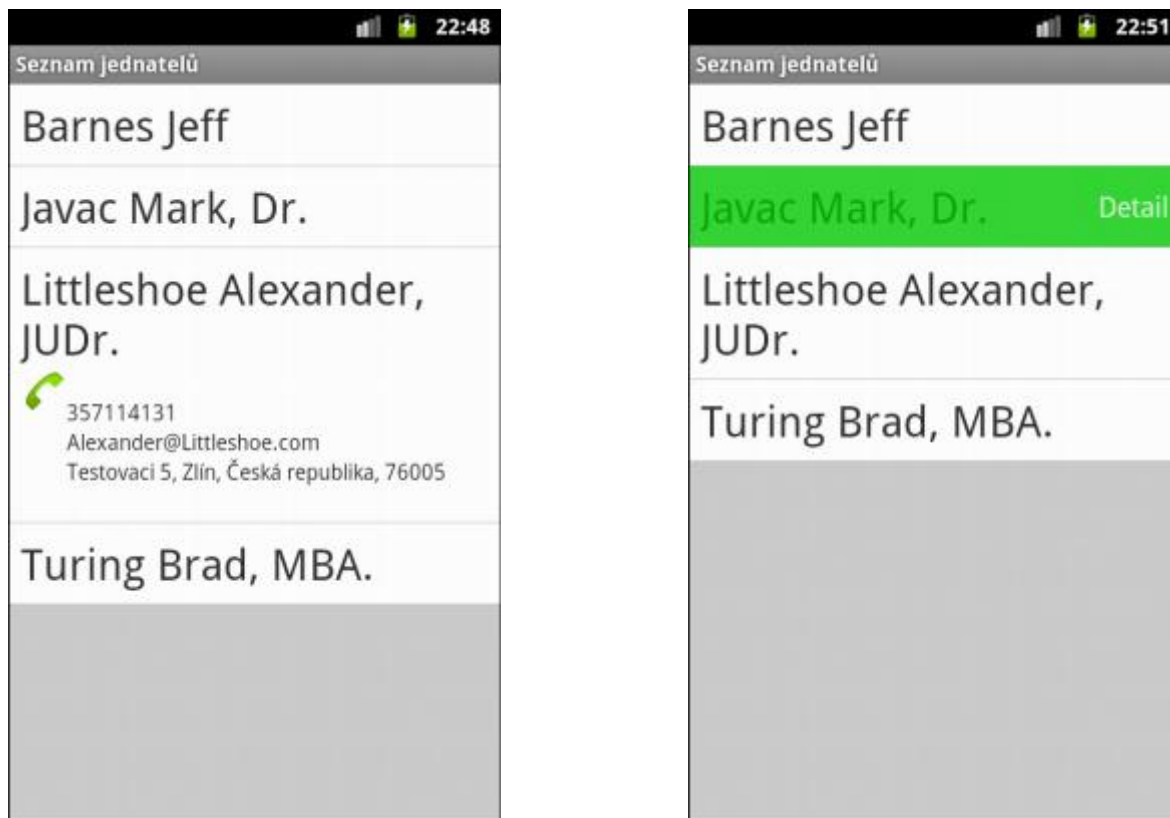
Je odvozena od třídy *Activity* a překrývá metodu *onCreate(Bundle)*, která je spouštěna při vytvoření aktivity *ActivityManagerem* operačního systému. V této metodě definuje layout, který bude zobrazen na obrazovku a v něm vyhledá tlačítka, aby jim mohla být přiřazena animace. Nakonec tato metoda spouští synchronizaci, tím je zajištěno spuštění pouze při startu aplikace. Pomocí metody *onResume()* je upraven text v poli data poslední synchronizace. Další spuštění synchronizace má na svědomí metoda *onDestroy()*, která je volána při odstraňování aktivity z paměti, to je nejčastěji při ukončení aplikace, ovšem k jejímu zavolání nemusí dojít.

### **Třída *CompanyListActivity* a *ExecutiveListActivity***

Tyto třídy budou popsány společně, protože se velmi podobají. Odvozují se od třídy *ListActivity*, tudíž představují obrazovku se seznamem položek. Není potřeba definovat layout, protože layout seznamu je předem daný. Dle návrhu se mají položky při klepnutí rozevřít a zobrazit dodatečné informace a navíc by měly reagovat na tažení prstem do stran. K tomu je potřeba využít vlastní implementaci položek seznamu a s tím spojenou implementaci vlastního adaptéru, který položky v seznamu spravuje.

Funkce „tažení prstu“ (tzv. *swipe*) však přináší problémy s použitím běžného seznamu *ListView*, jehož způsob zpracování dotekových událostí nedovoloval odchylovat a rozlišovat jednoduché kliknutí od pohybu prstu. Nezbyvá tedy nic jiného než vytvořit vlastní komponentu seznamu – třídu *SwipeListView*. Implementace zmíněných tříd bude popsána později.

Účelem těchto dvou aktivit je hlavně vytvoření instance seznamu *SwipeListView*, získání položek z databáze pomocí datové vrstvy, vytvoření adaptéru s těmito položkami a jeho předání seznamu. Dalšími činnostmi jsou tvorba hlavního a kontextové menu a obsluha událostí z těchto menu a také událostí ze seznamu *SwipeListView*.



Obr. 11. Ukázka seznamu jednatelů

### Třída *SwipeListView*

Odvozena od klasického seznamu *ListView*, aby překryla jeho zpracování dotykových událostí. Překrytím metody *onInterceptTouchEvent(MotionEvent)* je odchyťována událost doteku prstu, tato událost však není zkonsumována, ale je pouze zaznamenána poloha, kde k dotyku došlo a událost je dále propagována hierarchií prvků. Pokud by došlo ke zkonsumování události, budou všechny následující události spojené s touto událostí posílány rovnou této třídě. To má za následek, že událost o klepnutí není doručena aktivitě, která má na starost zavolat rozklepnutí položky.

Seznam *SwipeListView* je totiž koncipován tak, aby mohl pracovat s libovolnými položkami, takže o jejich možnosti „rozkliknutí“ nemůže vědět. Metodou *onInterceptTouchEvent(MotionEvent)* je odchyťován pouze první dotek – tedy přiložení prstu na dotykovou plochu. Další činnost, pohyb nebo uvolnění prstu, je odchyťován metodou *onTouchEvent(MotionEvent)*, která událost zpracovává a rozhoduje, jestli došlo k pohybu nebo se jednalo pouze o klepnutí. Tímto je dosaženo toho, že při pohybu prstu po položce je událost obsluhována seznamem a pokud se jedná pouze o dotek bez pohybu je

událost propagována hierarchií dále, kde jej odchyť aktivita a zavolá rozklepnutí na daném prvku.

Seznam při zpracování událostí volá metody rozhraní *SwipeListItem*, pokud jej položky implementují, mohou na události tažení prstem reagovat. Také obsahuje rozhraní *OnListItemSwipeListener*, jehož metody seznam volá při uvolnění prstu po dostatečné tažené vzdálenosti. Implementováním tohoto rozhraní v aktivitách je možno reagovat a například spustit aktivitu detailu společnosti nebo jednatele. Rozhraní obsahuje dvě metody, jsou to *onItemSwipeLeft(View, int)* a *onItemSwipeRight(View, int)* pro tažení doleva a doprava.

*SwipeListView* je tedy absolutně nezávislý a znovupoužitelný, stačí implementovat rozhraní *OnListItemSwipeListener*.

### **Třída *CollapsingListItemView***

Jedná se o položku seznamu, která se odvozuje od lineárního rozložení. Obsahuje jeden řádek s textem, se stylem jako položka seznamu, pod kterým je další textová oblast. Tato textová oblast je implicitně schována a při akci rozklepnutí je zobrazena.

Implementuje rozhraní *SwipeListItem*, aby mohla reagovat na události tažení prstem. Využívá vícevrstvého rozložení *expandable\_sub\_item*, popsaného dříve. V implementaci metody *setOffset(float)* přijímá parametrem taženou vzdálenost a tuto vzdálenost převádí na hodnotu průhlednosti, kterou nastavuje požadované vrstvě v rozložení. Zde dochází k problému u starších verzí OS Android. V novějších verzích je k dispozici metoda *setAlpha(float)*, která nastaví průhlednost celé vrstvy na požadovanou hodnotu. Ve starších verzích, než je Honeycomb (3.0), tato metoda není a je tedy nutné nastavovat průhlednost pomocí nastavení barvy každému prvku ve vrstvě. Díky tomuto rozdílnému přístupu je potřeba verzi OS kontrolovat.

### **Třída *CompanyListAdapter* a *ExecutiveListAdapter***

Tyto třídy představují datový adaptér pro seznamy jednatelů nebo společností. Odvozeny jsou od třídy *BaseAdapter* a jejich činností je poskytovat seznamu data a položky na vyžádání. Vnitřně jsou data reprezentována seznamem *ArrayList<T>*, který je naplněn daty v konstruktoru nebo pomocí metody *setExecutives(ArrayList<Executive>)* případně *setCompanies(ArrayList<Company>)*. Pro seznam poskytuje položku na dané pozici

metoda *getView(int, View, ViewGroup)*, která vrací položku jako instanci třídy *CollapsingListItemView*.

Počet položek adaptéru vrací překrytá metoda *getCount()*, položku na dané pozici metoda *getItem(int)*.

### **Třídy *CompanyDetailActivity* a *ExecutiveDetailActivity***

Odvozeny od třídy *Activity*, starají se o zobrazení detailu společnosti nebo jednatele. Jako rozložení definují *activity\_company\_detail* nebo *activity\_executive\_detail*. Aktivity se starají o vytvoření menu, o získání potřebných dat a obsluhu událostí. Na spodní straně obrazovky je zobrazen seznam přiřazených jednatelů společnosti nebo společností zastupovaných jednatelem. Protože data jsou zobrazena v rolovatelném obsahu – prvku *ScrollView* – není zde možné použít obyčejný seznam *ListView*. Android nedovoluje použití rolovatelného prvku uvnitř jiného. Z tohoto důvodu byl vytvořen jednoduchý seznam *LinearLayout*, který nepodporuje rolování a pouze skládá prvky za sebe. Aktivity obsluhují i události tohoto seznamu, takže implementují rozhraní *OnLinearLayoutClickListener*. Jeho metoda *onLinearLayoutClick(LinearLayout, int)* je volána při kliknutí na položku seznamu *LinearLayout*. Dále je zde také tvorba kontextového menu pro tento seznam a jeho obsluha pomocí metod *onCreateContextMenu(ContextMenu, View, ContextMenuInfo)* a *onContextItemSelected(MenuItem)*.

### **Třídy *CompanyEditActivity* a *ExecutiveEditActivity***

Tyto třídy jsou odvozeny od *Activity* a zobrazují editační formulář pro úpravu dat jednatele nebo společnosti. Mají na starosti načtení dat a jejich následné uložení v případě změny. Při stisku tlačítka *Uložit*, je volána metoda *saveChanges(Company)*, která vytvoří hlubokou kopii aktuálního objektu, do kopie předá upravená data z formuláře a následně kopii i originál porovná. Pokud se rovnají, nedošlo k žádným změnám a není potřeba ukládat do databáze. Naopak pokud se nerovná, je aktualizován v objektu čas poslední úpravy a je provedeno uložení do databáze zavoláním potřebné metody správcovské třídy – například metody *updateCompany(Company)* třídy *CompanyManager*. Po uložení nebo zrušení změn dojde k ukončení aktivity voláním metody *finish()* zděděné od třídy *Activity*.

### Třída *SettingsActivity*

Třída je odvozena od *PreferenceActivity* a při svém vytvoření načte možnosti z XML souboru *settings.xml*.

### Třída *Utils*

Knihovni třída, která poskytuje konstanty platné pro celou aplikaci a metody, které jsou užitečné a často používané, například *parseDate(String)* a *formatDate(Date)* pro převod data na řetězec a zpět nebo *isStringNullOrEmpty(String)* pro zjištění zda je řetězec prázdný nebo roven *null*. Této metody je například využito v metodě *isEmpty()* třídy *Address*.

## 4.1.4 Synchronizační služba

### Třída *SyncService*

Je odvozena od třídy *IntentService*, což umožňuje, aby byla spouštěna jako služba na pozadí. Službu je nutné uvést v manifestu aplikace:

```
<service android:name="cz.sild.droid.test.dpProto.sync.SyncService"
         android:enabled="true"></service>
```

Třída *IntentService* vyžaduje implementaci metody *onHandleIntent(Intent)*. Tato metoda je volána systémem a může být volána vícekrát, je nutné ji proto tomu přizpůsobit. V této metodě je nejdříve zjištěno datum poslední synchronizace. Následně je započata komunikace se službou metodou *doCheckin()*, která odešle službě autorizační údaje uživatele a unikátní identifikátor komunikace. Obdrží-li úspěšně odpověď, vyčte z HTTP hlaviček přidělený *token*. Pomocí tohoto *tokenu* a identifikátoru komunikace je vytvořen identifikační hash řetězec, který je použit k dalšímu požadavku na službu. Zjistí se, jestli existují nějaká data k synchronizaci a pokud ano, je synchronizace provedena. Probíhá tak, že jsou změněná data vyžádána od služby, porovnají se s daty změněnými lokálně pro odhalení konfliktů. Konflikty jsou vyřešeny podle data poslední úpravy. Nakonec jsou změněná data očištěná od konfliktů odeslána službě k uložení.

### Třída *ConnectionReceiver*

Třída dědí od *BroadcastReceiver* a tím získává možnost naslouchat systémovým událostem, které jsou definovány v manifestu aplikace:

```
<receiver android:name="cz.sild.droid.test.dpProto.sync.ConnectionReciever">
    <intent-filter>
        <action android:name="android.net.conn.CONNECTIVITY_CHANGE">
        </action>
    </intent-filter>
</receiver>
```

Do manifestu je přidána položka *receiver*, které je nastavena akce *android.net.conn.CONNECTIVITY\_CHANGE* jako položka prvku *intent-filter*. Tímto se dává systému najevo, že pokud se vyskytne výše zmíněna akce, chce o ní vědět také třída *ConnectionReciever*.

Třída je povinná implementovat metodu *onReceive(Context, Intent)*, která je volána při výskytu události v systému. V metodě je získán z parametru *Intent* aktuální stav připojení. Pokud je zařízení připojeno k Internetu dojde ke spuštění služby *SyncService*.

### Použití Google GSON

Data jsou pro přenos převáděna do formátu JSON. K tomu je využita knihovna Google GSON, protože nabízí velmi jednoduché použití a lze také jednoduše definovat vlastní převod do JSON a zpět. Nakonec se ukázalo, že vlastní převod bude nutný, neboť datové entity se u služby trochu liší. Je využita třída *JsonDefinitions*, která pouze definuje konstanty pro názvy prvků ve formátu JSON.

Pro každou datovou entitu je vytvořena třída *Serializer* implementující rozhraní *JsonSerializer<T>* a třída *Deserializer* implementující *JsonDeserializer<T>*. Implementovaná metoda *serialize(T, Type, JsonSerializationContext)* převádí daný typ do formátu JSON pomocí třídy *JsonObject*. Využívá jí i metoda *deserialize(T, Type, JsonDeserializationContext)* pro přečtení dat z formátu JSON.

## 4.2 Implementace webové služby

Pro vytvoření REST webové služby v jazyce Java je použita implementace Jersey. Služba poběží na zvoleném aplikačním serveru Apache Tomcat 6.0 a bude využívat databázi MySQL.

Zdrojové kódy webové služby jsou rozděleny do několika balíčků (*package*) kvůli přehlednosti. Do balíčku *cz.sild.dpWebService.data* jsou umístěny datové entity a jejich správcovské třídy.

### 4.2.1 Datová vrstva

Pro uložení dat jsou vytvořeny následující tabulky v MySQL databázi (definice sloupců tabulek se nachází v příloze P I):

- *companies* pro společnosti
- *executives* pro jednatele
- *addresses* pro adresy
- *ce\_relations* pro vazby
- *users* pro uživatele

Přístup k těmto tabulkám z kódu služby je umožněn přes tzv. *connector*, výrobce databáze jej poskytuje pro několik různých programovacích jazyků. *MySQLConnector/J* je tedy určen pro jazyk Java a je potřeba ho načíst pro přístup k databázi:

```
Class.forName("com.mysql.jdbc.Driver");
```

Datové entity jsou velmi podobné entitám v mobilní aplikaci. Rozdílem je vynechání třídy *Contact*. Kontaktní informace jsou uvedeny jako proměnné typu *String* přímo v třídách *Company* a *Executive*. Třída *Address* je shodná jako v mobilní aplikaci.

#### Třída *CERelation*

Tato třída má zde obsahuje pouze globální identifikátory. Vzhledem k tomu, že se zde s daty nepracuje, nejsou zde lokální identifikátory vůbec potřeba.

#### Třídy pro práci s databází

Pro práci s datovými entitami nad databází jsou vytvořeny jednotlivé správcovské třídy, které jsou odvozeny od třídy *BaseManager*. Tato taktika zvýší přehlednost kódu a také ušetří čas potřebný pro implementaci těchto tříd.

#### Třída *BaseManager*

Abstraktní třída, definuje společné konstanty pro všechny správcovské třídy, deklaruje abstraktní metody, které jsou vyžadovány od všech tříd, a definuje několik metod společných pro všechny třídy.

### **Třída *AddressManager***

Tato třída je odvozena od *BaseManager* a má na starosti práci s tabulkou adres v databázi. Název tabulky a názvy sloupců definuje jako konstanty a tím rozšiřuje sadu již definovaných sloupců v rodičovské třídě.

Pro vytvoření tabulky slouží implementace metody *createTable()*. Při čtení dat z databáze je výsledek dotazu umístěn v instanci třídy *ResultSet*, pro získání dat adresy z tohoto objektu slouží metoda *parseAddress(ResultSet)*, která jako parametr přijímá objekt třídy *ResultSet*, z něj postupně vyčte data adresy, pomocí kterých vytvoří novou instanci třídy *Address* a tuto vrací jako výsledek.

### **Třída *CompanyManager***

Odvozena od *BaseManager*, slouží pro práci s tabulkou společností. Data k synchronizaci je možné získat voláním metody *getCompaniesUpdatedSince(Date)*. Uložení přijatých dat mají na starosti metody *insertCompany(Company)* a *updateCompany(Company)*.

### **Třída *ExecutiveManager***

Je odvozena od *BaseManager* a má na starosti tabulku jednatelů. Získání dat k synchronizaci lze použitím metody *getExecutivesUpdatedSince(Date)* a uložení dat do databáze lze provést metodami *insertExecutive(Executive)* a *updateExecutive(Executive)*.

### **Třída *CERelationManager***

Používá se pro vkládání a získávání vazeb mezi společnostmi a jednatelem. Odvozuje se od *BaseManager*. Základní operace jsou dostupné přes metody *insertRelation(CERelation)* a *updateRelation(CERelation)*.

### **Třída *UserManager***

Tato třída zastřešuje správu uživatelů v databázi a synchronizace se jí netýká. Slouží pro získání dat uživatele při jeho autentifikaci, případně pro přidání, změnu nebo odebrání uživatele. I tato třída je odvozena od *BaseManager*.

### **Výčet (Singleton) *UserDao***

Výčet je v jazyce Java doporučovaná implementace vzoru Singleton [21]. Ve výčtu se definuje pouze jedna hodnota, například *Instance* a dále se definují metody, které jsou přes tuto instanci zpřístupněny, a také soukromý bezparametrický konstruktor. Takto je docíleno

funkce *Sigletonu*. Výčet *UserDao* obsahuje hashovou tabulku (*HashMap*) uživatelů, ke které poskytuje přístup přes metody *addUser(String, User)*, *contains(String)*, *containsUser(User)*, *remove(String)* a *get(String)*. *UserDao* slouží jako seznam uživatelů komunikujících se službou a je využíván pro ověření přicházejících požadavků.

#### 4.2.2 Komunikační rozhraní služby

Hlavní balíček *cz.sild.dpWebService* obsahuje třídy zodpovědné za veřejné rozhraní služby. Implementace *Jersey* využívá pro mapování tříd a metod jako veřejných rozhraní anotace. Služba je dostupná na adrese *http://adresa\_serveru/dpWebService/rest* a jednotlivé metody jsou mapovány dodatkem k této adrese.

##### Třída *CheckinProvider*

Třída je zodpovědná za autentifikaci uživatele je dostupná na adrese „*/checkin*“. Implementuje dvě metody. První je *checkIn(long, String, String)*, která je výchozí metodou pro požadavky typu *GET* a tedy dostupná na stejné adrese. Stará se právě o ověření uživatele. Přijímá identifikátor uživatele, unikátní identifikátor komunikace a *hash* uživatelova hesla. Pokud jsou přijaté údaje správné, metoda odpoví *HTTP* odpovědí se stavem *200 OK* a do hlavičky odpovědi přidá vygenerovaný *token*. Nejsou-li údaje správné, odpoví se stavem *401 Unauthorized* nebo stavem *400 Bad Request* jsou-li údaje ve špatném tvaru, případně existuje-li už jiný uživatel komunikující pod daným identifikátorem.

Druhou metodou je *getSyncItemCount(String, String, String)*, která přijímá unikátní identifikátor komunikace, identifikační řetězec a datum poslední synchronizace. Provede ověření uživatele, pokud údaje nesouhlasí, pak vrací odpověď se stavem *401 Unauthorized*. Při špatných parametrech vrací *400 Bad Request*. Pokud ověření proběhlo v pořádku, metoda zjistí, zda jsou nějaké data změněna od zadané doby a jejich počet vrací v hlavičce odpovědi, spolu s nově vygenerovaným *tokenem*. Je dostupná pod adresou „*/checkin/sync*“ metodou *GET*.

##### Třída *CompanyProvider*

Je zpřístupněna pod adresou „*/company*“. Definuje metody *getUpdateSince(String, String, String)*, přístupnou pod adresou „*/company/sync*“ metodou *GET*. Metoda provede ověření uživatele, a pokud je úspěšně ověřen získá z databáze požadovaná data, převede je do

formátu JSON a vloží do odpovědi se stavem *200 OK*. Do hlavičky odpovědi je opět vložen nový *token*.

Druhou metodou je *postUpdatedSince(String, HttpServletRequest)*, která je dostupná pod stejnou adresou, ale pro požadavky typu *POST*. Tato metoda opět ověřuje uživatele a v případě úspěšného ověření přijatá data převede na objekty *Company* a uloží do databáze.

### **Třída *ExecutiveProvider***

Pracuje naprosto stejně jako *CompanyProvider*, jen je dostupná pod adresou „*/executive*“ a metody pod adresou „*/executive/sync*“. Má na starosti synchronizaci jednatelů.

### **Třída *CERelationProvider***

Taktéž je shodná s *CompanyProvider*. Stará se o synchronizaci vazeb. Její adresa je „*/cerelation*“ a metody jsou dostupné pod „*/cerelation/sync*“.

### **Třída *Utils***

Slouží jako knihovná třída, jsou zde definovány globální konstanty a některé důležité funkce. Například *generateToken(String)* pro vygenerování tokenu pro komunikaci nebo *validateUser(String, String)* pro ověření uživatele.

### **Využití JSON**

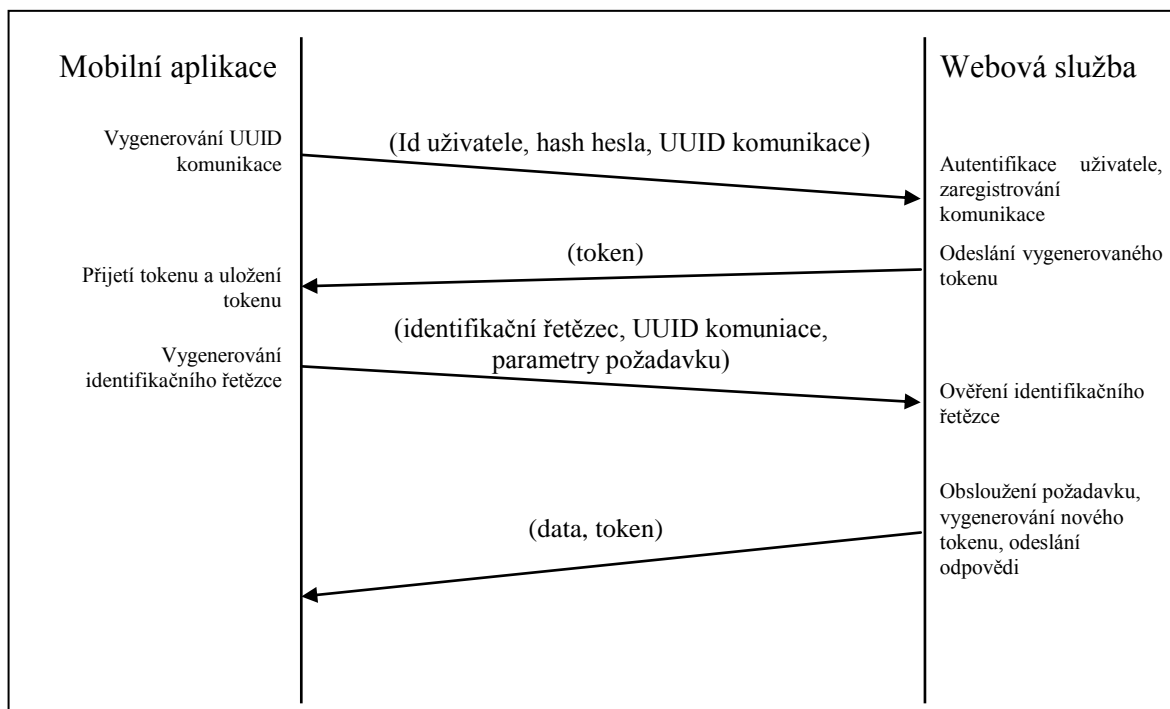
Pro převod do formátu JSON a naopak získání dat z něj je použita knihovna Google GSON a proces převodu je převzat z mobilní aplikace. Je pouze potřeba upravit třídy *CompanySerializer*, *CompanyDeserializer*, *ExecutiveSerializer* a *ExecutiveDeserializer* neboť zde není implementována třída *Contact* a tedy kontaktní data přečtená z objektu JSON jsou předána přímo objektům *Company* nebo *Executive*.

## 5 ZABEZPEČENÍ APLIKACE

Kontaktní informace jsou velmi často dostupné veřejnosti, například na webových stránkách společnosti. Aplikace tudíž nepracuje se soukromými daty, které by vyžadovaly důkladné zabezpečení. I přesto však byla zabezpečení věnována pozornost a to především zabezpečení komunikace.

Největším bezpečnostním rizikem je odcizení nebo ztráta mobilního zařízení. Lokální databáze mobilní aplikace je v rámci zařízení přístupná, takže pro útočníka nepředstavuje velký problém data získat. Bylo by možné použít šifrování v databázi *SQLite*, ovšem tato funkce není zdarma. Dále by bylo možno data šifrovat a ukládat je v obyčejné databázi šifrované. Pokud by to bylo požadavkem, druhá možnost se jeví jako nejlepší řešení ve spojení se zabezpečením celé aplikace uživatelským heslem.

Zabezpečení komunikace je realizováno s využitím generovaného *tokenu*. Komunikace začíná žádostí aplikace o přidělení *tokenu*, k tomu je potřeba unikátní identifikátor komunikace, který si klientská strana vygeneruje náhodně (pomocí *UUID.randomUUID()*), identifikátor uživatele a *hash* uživatelova hesla. Následně dojde k ověření uživatele, a pokud je úspěšné, je vygenerován *token* a odeslán v hlavičkách odpovědi. Klient tento *token* použije pro vytvoření identifikačního *hashe*, který přiloží k další žádosti. Tento identifikační řetězec je porovnáván na straně služby a v případě shody je požadavek obslužen. Nakonec je vytvořen nový *token* a přidán k odpovědi (viz Obr. 12).



Obr. 12. Diagram komunikace

Komunikace probíhá přes HTTP protokol, takže ji útočník může odposlouchávat. Je však navržena tak, aby i při odposlechu nemohl útočník získat přístup k libovolným metodám služby. Pro vytvoření identifikačního řetězce i pro jeho ověření je použit tzv. *salt* řetězec, který zná služba i mobilní aplikace, avšak není nikdy přenášen a nelze tedy jeho podobu zjistit odposlechem. Zachytí-li útočník *token* směřující ke klientovi, nebude mu nic platný, neboť nezná způsob jakým je vytvořen identifikační řetězec z tohoto *tokenu*. Taktéž uživatelské heslo není nikdy přenášeno nezabezpečené, ale pouze jeho *hash*, aby útočník odposlechem nemohl získat přístup do mobilní aplikace.

V případě, že by bylo vyžadováno lepší zabezpečení, tedy aby byla chráněna i přenášená data, je možné využít šifrování komunikace pomocí protokolu HTTPS.

## 6 TESTOVÁNÍ A VYHODNOCENÍ EFEKTIVNOSTI VYTVOŘENÉ APLIKACE

Při vývoji tříd datové vrstvy bylo použito jednotkové testování, které usnadnilo nalezení chyb. Pro snadnou implementaci jednotkových testů byl použit framework *jUnit*. Třídy uživatelského rozhraní vyžadují použití testovacích případů *ActivityUnitTestCase*, díky kterým jsou pak aktivity spuštěny v izolovaném kontextu (případné události nejsou delegovány systému).

Aplikace byla v rámci testu používána několika náhodně vybranými uživateli. Byla sledována jejich práce s aplikací, zda jim některé ovládací prvky dělají problém, jsou-li informace přehledné a není-li ovládání nebo uspořádání matoucí. Hodnocení uživatelů bylo převážně pozitivní, největší výtky směřovaly na černobílé provedení aplikace bez barev. Velmi kladně hodnotili použití tažení prstu po položce seznamu a možnost zadání více kontaktních údajů.

Cílovou skupinou uživatelů jsou střední a velké společnosti, jejichž obchodní oddělení komunikuje s velkou skupinou obchodních partnerů nebo zákazníků. Sdružení a synchronizace pracovních kontaktů je v takovém případě velmi žádoucí.

Přehledná aplikace, která nemá konkurenci, je vždy výhodným prostředkem pro zdokonalení mobilní komunikace, jak uvnitř společnosti, tak i navenek. Jednání s dodavateli, zahraniční pracovní cesty zaměstnanců i uložení kontaktů v databázi společnosti jsou mnohem více pohodlné, rychlé a bezpečné.

Společnosti osloví efektivním způsobem více potenciálních zákazníků či odběratelů a aplikace zaznamená veškeré kontaktní údaje (adresa, telefon, fax atd.).

Finanční zisk zde přináší hlavně poskytování serverové části, což zahrnuje správu a pronájem serveru s běžící synchronizační službou.

## ZÁVĚR

V rámci práce byla vytvořena mobilní aplikace určená pro systém Android a synchronizační webová služba. Aplikace plní požadavky zadání a umožňuje správu a synchronizaci kontaktů. Podstatnou výhodou aplikace je absence konkurence na trhu, což zvyšuje potenciál jejího případného uplatnění.

Teoretická část práce popisuje operační systém Android, jeho jednotlivé části a principy. Pojednává dále o návrhových principech, díky kterým mohl vzniknout kvalitní návrh aplikace i služby.

V praktické části je návrh podrobně rozebrán a popsán. Probíhal iterativně a několikrát byl revidován a změněn. Častokrát musely být nejprve možnosti ověřeny zkouškou v kódu a teprve pak s nimi mohlo být počítáno v návrhu. U návrhu seznamů společností a jednatelů bylo takto zjištěno, že běžný seznam v Androidu není dostatečný a bude třeba vytvořit vlastní.

Při návrhu struktur se projevilo opoždění návrhu a předčasná implementace, která je ale zdůvodnitelná ověřením reálných možností, což vedlo k tomu, že původně rozdílné třídy v prvotním návrhu mohly být ve výsledku odvozeny od společného předka a tím by došlo ke zjednodušení v celé aplikaci.

Snaha o kompatibilitu s verzí Androidu 2.3 i 4.0 přinesla několik úskalí, se kterými v návrhu nebylo počítáno. Novější Android přinesl například využití navigační lišty v horní části každé aplikace, což by usnadnilo navigaci v aplikaci. Naopak důležitým poznatkem je, že na platformě Android je tvorba služby v pozadí a její komunikace v síti Internet velmi jednoduchou záležitostí.

Kód aplikace byl psán s ohledem na znovupoužitelnost a snadnou udržovatelnost. Bylo dbáno na dobrou čitelnost kódu, aby programátor, který by aplikaci dále rozšiřoval nebo upravoval, nemusel kód zdlouhavě luštit.

Do budoucna je možné do aplikace zanechat více funkcí. Webová služba postrádá administrativní rozhraní, které by při reálném nasazení bylo velmi vhodnou pomůckou. Aplikace by mohla být lépe přizpůsobena pro největší displeje sjednocením několika obrazovek. Dalšími vhodnou funkcí je například integrace s kalendářem - zaznamenávání významných výročí, jmenný seznam svátků, termíny konferencí a porad. Také se může

hodit možnost importu a exportu kontaktů, například do systémového uložení nebo pro odeslání ve formě vizitky, případně možnost importu a exportu pomocí QR kódů.

## ZÁVĚR V ANGLIČTINĚ

As part of the work mobile application for Android platform and synchronization web service was designed and created. The application meets the requirements and allows user to manage and synchronize contacts. A major benefit of the application is the lack of competition in the market, which increases the potential for its possible application.

The theoretical part describes the Android operating system, its various parts and principles. Further it discusses the design principles, due to which a good design of the application and service could be made.

In the practical part the design is described and analyzed in detail. It was conducted iteratively and repeatedly revised and changed. Possibilities often had to be verified in code, before they can be used in the draft. While designing lists of companies and executives have been found, that implicit Android list is of no use and therefore a custom list is to be made.

Early implementation, justified by verifying the real possibilities, and delays in the design, led to the fact that originally different classes in the initial design have become very similar, so they could be derived from a common ancestor and thus simplify the entire application code.

Striving for compatibility with Android versions 2.3 and 4.0 has brought some difficulties with which was not calculated in the draft. Newer Android brought the navigation bar at the top of each application, which could simplify navigation. On the contrary, an important observation is that the creation of service in the background and its communication on the Internet is a very simple matter on the Android platform.

The application code was written with regard to reusability and easy maintainability. Minded the readability of code, the programmer, which would further extend or modify the code, did not have to tediously decipher it.

In the future it is possible to add multiple functions. Web Service lacks the administrative interface, which is very suitable tool in the case of real deployment. Application can be better adapted for the largest displays by unification of several screens. Another suitable feature can be an integration with calendar - recording of significant anniversaries, dates of meetings and conferences. It also can come in handy to have ability to import and export

contacts, such as into system contact storage or to send them in the form of business cards, or the ability to import and export contacts using QR codes.

**SEZNAM POUŽITÉ LITERATURY**

- [1] ANDROID DEVELOPERS.COM *What is Android?* [online], ©2011 [cit. 2013-05-02]. Dostupné z WWW: <http://developer.android.com/guide/basics/what-is-android.html>
- [2] OPEN HANDSET ALLIANCE.COM *Industry Leaders Announce Open Platform for Mobile Devices* [online]. ©2007 [cit. 2013-05-02]. Dostupné z WWW: [www.openhandsetalliance.com/press\\_110507.html](http://www.openhandsetalliance.com/press_110507.html)
- [3] CLABURN, Thomas. *Google's Secret Patent Portfolio Predicts gPhone* [online]. InformationWeek, 19. září 2007 [cit. 2013-05-02]. Dostupné z WWW: [http://www.informationweek.com/googles-secret-patent-portfolio-predicts/201807587?cid=nl\\_IWK\\_daily](http://www.informationweek.com/googles-secret-patent-portfolio-predicts/201807587?cid=nl_IWK_daily)
- [4] ANDROID DEVELOPERS.COM *Application Fundamentals* [online]. ©2011, [cit. 2013-05-05]. Dostupné z WWW: <http://developer.android.com/guide/components/fundamentals.html>
- [5] LEE, Wei-Meng. *Beginning Android Application Development*. Indianapolis: Wiley Publishing, Inc., 2011. ISBN 978-1-118-08780-0.
- [6] FELKER, Donn. *Android Application Development For Dummies*, New York, Hoboken: Wiley Publishing, Inc., 2011. ISBN 978-0-470-77018-4.
- [7] ANDROID DEVELOPERS. *What is Android?* [online]. ©2011, [cit. 2013-05-02]. Dostupné z WWW: <http://developer.android.com/about/index.html>
- [8] MEDNIEKS, Zigurd, DORNIN, Laird, MEIKE, G. Blake, NAKAMURA, Masumi. *Programming Android*. Sebastopol: O'Reilly Media, 2012. 2. vydání ISBN 978-1-449-31664-8.
- [9] TRHOŇ, Ondřej. *8 problémů, které mohou Androidu zlomit vaz.* [online]. ©2013 [cit. 2013-05-15]. Dostupné z WWW: <http://www.androidmarket.cz/ruzne/8-problemu-ktere-mohou-androidu-zlomit-vaz/>
- [10] ELECTRONISTA.COM. *Studies: Android's malware, piracy problem growing* [online], ©2013 [2013-05-15]. Dostupné z WWW: <http://www.electronista.com/articles/13/05/15/drives.developers.to.freemium.model.hurts.platform/>

- [11] KINGSLEY-HUGHES, Adrian. *Could Android fail?* [online], ©2013 ZDNet. [2013-05-01]. Dostupné z WWW: <http://www.zdnet.com/could-android-fail-7000014761/>
- [12] HATINA, Petr. *Programování v jazyku Java (1) – Úvod* [online]. ©2004 Linuxsoft.cz [2013-05-01]. Dostupné z WWW: [http://www.linuxsoft.cz/article.php?id\\_article=244](http://www.linuxsoft.cz/article.php?id_article=244)
- [13] HEROUT, Pavel. *Učebnice jazyka Java*. České Budějovice: Kopp, 2010. 2. vydání. ISBN 978-80-7232-398-2.
- [14] PARNAS, David L. *On the Criteria To Be Used in Decomposing Systems into Modules* [online], ©1972 Communications of the ACM, č. 12, Carnegie-Mellon University, Pittsburgh, PA 15213, Prosinec 1972. Dostupné z WWW: <http://sunner.cn/courses/C/reference/Parnas72.pdf>
- [15] MCCONNELL, Steve. *Dokonalý kód - Umění programování a techniky tvorby software*. Brno: Computer Press, 2006. ISBN 80-251-0849-X.
- [16] JONÁŠ, Martin. *Návrhové principy SOLID* [online]. ©2012 Zdrojak.cz [2013-05-01]. Dostupné z WWW: <http://www.zdrojak.cz/clanky/navrhove-principy-solid/>
- [17] MARTIN, Robert C. *Design Principles and Design Patterns* [online], ©2009 Objectmentor.com [cit. 2013-05-14]. Dostupné z WWW: [http://www.objectmentor.com/resources/articles/Principles\\_and\\_Patterns.pdf](http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf)
- [18] LIEBERHERR, Karl. *Law of Demeter* [online], ©2004. College of Computer and Information Science, Northeastern University. [cit. 2013-05-14]. Dostupné z WWW: <http://www.ccs.neu.edu/home/lieber/LoD.html>
- [19] JONÁŠ, Martin. *Návrhové principy DRY* [online]. ©2012 Zdrojak.cz [2013-05-01]. Dostupné z WWW: <http://www.zdrojak.cz/clanky/navrhove-principy-dry/>
- [20] JONÁŠ, Martin. *GRASP – 6 – Controller a jak GRASP používat* [online]. ©2012 Zdrojak.cz [2013-05-01]. Dostupné z WWW: <http://www.zdrojak.cz/clanky/grasp-6-controller-a-jak-grasp-pouzivat/>
- [21] BLOCH, Joshua. *Effective Java*. Boston: Addison-Wesley Professional, 2008. ISBN 978-0-321-35668-0.

**SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK**

např.	například
tj.	to je
tzv.	takzvaný
BSD	Berkeley Software Distribution – operační systém odvozený od Unixu
GNU	projekt zaměřený na svobodný software
HTTP	Hypertext Transfer Protocol - internetový protokol určený pro výměnu hypertextových dokumentů
JSON	JavaScript Object Notation - datový formát nezávislý na počítačové platformě
OS	Operační systém
REST	Representational State Transfer - architektura rozhraní, navržená pro distribuované prostředí
SDK	Software Development Kit – balík vývojových nástrojů
SOAP	Simple Object Access Protocol - je protokolem pro výměnu zpráv založených na XML
SQL	Structured Query Language – dotazovací jazyk používaný především v databázích
SSL	Secure Sockets Layer – protokol, který poskytuje zabezpečení komunikace šifrováním
UI	User Interface – uživatelské rozhraní
VM	Virtual Machine – virtuální stroj
XML	Extensible Markup Language – strukturovaný formát textově uložených dat
AWT	Abstract Window Toolkit – sada komponent pro tvorbu uživatelského rozhraní v Javě
CSV	Comma-separated values - hodnoty oddělené čárkami, datový formát

**SEZNAM OBRÁZKŮ**

Obr. 1. Architektura OS Android .....	13
Obr. 2. Nejpoužívanější verze OS Android .....	15
Obr. 3. Obrazovky Contacts+ .....	24
Obr. 4. Seznam kontaktů v aplikaci ConTacTs .....	25
Obr. 5. Diagram vrstev systému.....	28
Obr. 6. Diagram datových tříd .....	31
Obr. 7. Diagram aktivit .....	33
Obr. 8. Wireframe hlavní obrazovky (vlevo) a seznamu společností (vpravo) .....	34
Obr. 9. Wireframe obrazovek detailu a editace .....	36
Obr. 10. Hlavní obrazovka aplikace .....	45
Obr. 11. Ukázka seznamu jednatelů .....	48
Obr. 12. Diagram komunikace.....	58

## SEZNAM PŘÍLOH

P I Tabulky v databázi MySQL

## PŘÍLOHA P I: TABULKY V DATABÁZI MYSQL

Tab. 1. Definice tabulky *companies*

Název sloupce	Datový typ	Podrobnosti
id	BIGINT	NOT NULL, PK, AI
name	VARCHAR(256)	NOT NULL
desc	VARCHAR(1024)	
phones	VARCHAR(512)	
faxes	VARCHAR(512)	
emails	VARCHAR(512)	
www	VARCHAR(1024)	
addressid	BIGINT	FK
uuid	CHAR(36)	
active	Bit	NOT NULL, DEFAULT 1
created	TIMESTAMP	NOT NULL, DEFAULT CURRENT_TIMESTAMP
updated	TIMESTAMP	NOT NULL

Tab. 2. Definice tabulky *executives*

Název sloupce	Datový typ	Podrobnosti
id	BIGINT	NOT NULL, PK, AI
name	VARCHAR(512)	
surname	VARCHAR(1024)	
title	VARCHAR(64)	
position	VARCHAR(256)	
phones	VARCHAR(512)	
faxes	VARCHAR(512)	
emails	VARCHAR(512)	
addressid	BIGINT	FK
uuid	CHAR(36)	
active	Bit	NOT NULL, DEFAULT 1
created	TIMESTAMP	NOT NULL, DEFAULT CURRENT_TIMESTAMP
updated	TIMESTAMP	NOT NULL

Tab. 3. Definice tabulky *addresses*

Název sloupce	Datový typ	Podrobnosti
addressid	BIGINT	NOT NULL, PK, AI
street	VARCHAR(1024)	
city	VARCHAR(1024)	
country	VARCHAR(512)	
postal	VARCHAR(64)	
uuid	CHAR(36)	
active	Bit	NOT NULL, DEFAULT 1
created	TIMESTAMP	NOT NULL, DEFAULT CURRENT_TIMESTAMP
updated	TIMESTAMP	NOT NULL

Tab. 4. Definice tabulky *ce\_relations*

Název sloupce	Datový typ	Podrobnosti
id	BIGINT	NOT NULL, PK, AI
companyuuid	CHAR(36)	
executiveuuid	CHAR(36)	
uuid	CHAR(36)	
active	Bit	NOT NULL, DEFAULT 1
created	TIMESTAMP	NOT NULL, DEFAULT CURRENT_TIMESTAMP
updated	TIMESTAMP	NOT NULL