VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

KONTEJNER PRO MIGRUJÍCÍ SOFTWAROVÉ KOMPONENTY BEŽÍCÍ NA OS ANDROID

A CONTAINER FOR MIGRATATING SOFTWARE COMPONENTS RUNNING ON ANDROID OS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. VLADIMÍR ŠČEŠŇÁK**
AUTHOR

VEDOUCÍ PRÁCE **RNDr. MAREK RYCHLÝ, Ph.D.**
SUPERVISOR

BRNO 2018

Abstrakt

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v českém (slovenském) jazyce.

Abstract

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v anglickém jazyce.

Kľúčové slová

Sem budou zapsána jednotlivá klíčová slova v českém (slovenském) jazyce, oddělená čárkami.

Keywords

Sem budou zapsána jednotlivá klíčová slova v anglickém jazyce, oddělená čárkami.

Citácia

ŠČEŠŇÁK, Vladimír. *Kontejner pro migrující softwarové komponenty bežící na OS Android*. Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce RNDr. Marek Rychlý, Ph.D.

Kontejner pro migrující softwarové komponenty bežící na OS Android

Prehlásenie

Prehasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením RNDr. Mareka Rychlého, Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Vladimír Ščešňák
26. apríla 2018

Poďakovanie

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant, apod.).

Obsah

Kapitola 1

Úvod

[TODO: Úvod, cieľ práce, teória, návrh...]

[TODO: kam by som mal dať reflexiu(?) - použitá technológia(?)]

Kapitola 2

Operačný systém Android

Táto kapitola sa venuje operačnému systému Android, ktorý v roku 2007 uverejnilo konzorcium Open Handset Alliance, pozostavajúce z technologických spoločností ako je Google. Android bol uvedený pod open-source licenciou `Apache/MIT` a spolu s ním uzrelo svetlo sveta aj prvá beta verzia Android Software Development Kit (v skratke SDK). Ako uvádza [5] v priebehu pár mesiacov si stiahlo prvú beta verziu zo stránky Googlu niekoľko miliónov ľudí a v roku 2008 bol v Spojených štátoch amerických predstavený prvý mobilný telefón bežiaci na tejto platforme.
Cieľom uvedenia tohto operačného systému bolo zjednodušenie vývoja a predávania softvérových aplikáci bežiacich na mobilných telefónoch a tiež zaviesť štandard podobný PC a Macintoshu, ktorý by bol primárne určený pre mobilné telefóny. V súčasnosti sa ale táto platforma rozšírla aj na mikropočítače, inteligentné televízory (známe ako Android TV), inteligentné hodinky (Android Wear), palubné systémy aut a ďalšie druhy elektroniky. V čase písaní tejto práce je operačný systém Android dostupný vo verzii 8.0 Oreo. Informácie uvedené v tejto kapitole boli čerpané zo zdroja [1]

2.1 Vrstvy operačného systému Android

Ako môžme vidieť na obrázku 2.1 samotný Android je založený na **linuxovom jadre**, čož je aj jeho prvá vrstva. To umožňuje využiť kľúčové funkcie zabezpečenia a tiež umožňuje výrobcom zariadení ľahko vyvinúť ovládače hardvéru.

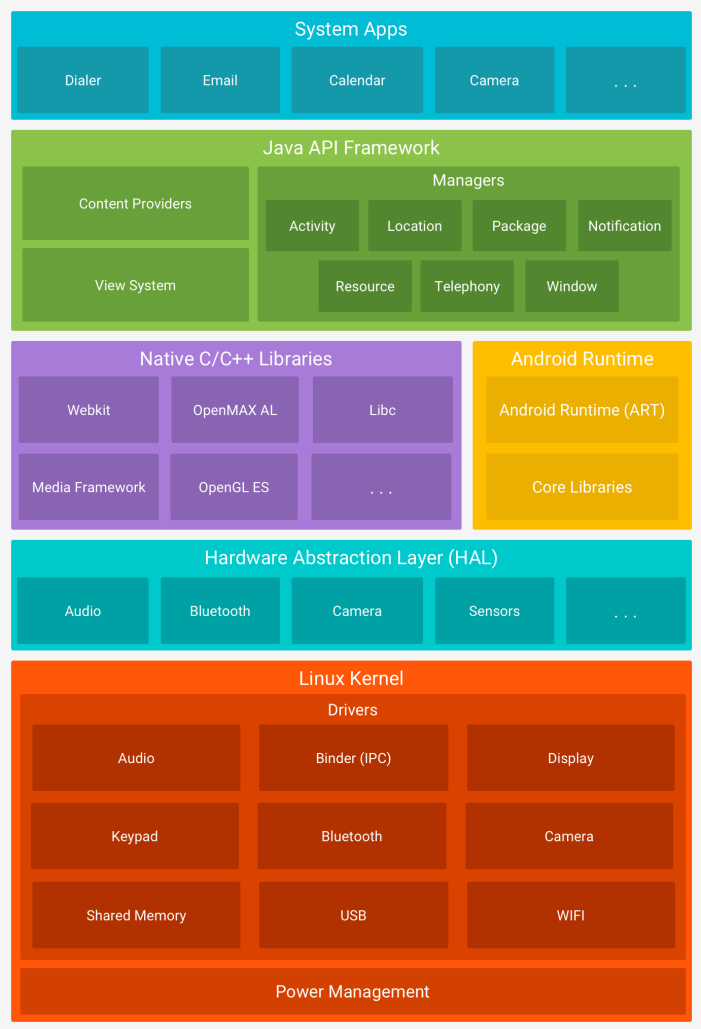
Ďalšou vrstvou je **hardvérová abstrakčná vrstva**, ktorá pozostáva z viacerých modulov a poskytuje štadardné rozhrania, ktoré vystavujú schopnosti hardvéru zariadenia do vyššej úrovni aplikačného rámca Java API. Úlohou vrstvy, ktorá je označená ako **Android Runtime** je tá, že každá aplikácia beží vo svojom vlastnom procese a má vlastnú inštanciu **Android Runtime**. Tá vykonáva beh viacerých virtuálnych strojov s nízkou pamäťou pomocou `DEX` súborov, čo je formát bytekódu špeciálne navrhnutého pre Android.

Mnoho základných komponentov a služieb systému Android, ako už aj vyššie spomenuté, vyžadujú natívne knižnice napísané v jazyku `C` a `C++` a samotný Android poskytuje aplikačné rozhranie, pre jednoduchší prístup z vyvýjaných aplikácii.

Java API Framework obsahuje základné stavebné bloky, písané v jazyku Java, ktoré potrebuje programátor pri vytváraní aplikácii. Existujú štyri rôzne typy komponentov aplikácie a to Aktivity, Služby, Príjmače a Poskytovatelia obsahu. Hlavne vďaka týmto blokom, môžme ľahko vytvárať prívetivé používateľské rozhranie, pristupovať k zdrojom ako

napríklad grafika, či iné súbory, alebo spravovať samotné aktivity. Túto vrstvu si popíšeme detailnejšie v ďalších častiach.

Poslednou vrstvou sú **Systémové aplikácie**, kde ide o množinu základných aplikácii, ktoré sú dodávane s každým Androidom. Ide hlavne o aplikácie ako telefonovanie, vytváranie textových správ, kamera, kalendár apodobne.



Obr. 2.1: Platforma Android

2.2 Aktivity a ich životný cyklus

Aktivity sú zakladným stavebným blokom aplikácii postavených na platforme Android. Sú akýmsi vstupným bodom medzi užívateľovom a aplikáciou. Samotná aktivita poskytuje okno, kde aplikácia vykresľuje svoje užívateľské rozhranie, ktoré môže vyplniť celú obrazovku, môže byť menšie ako celá obrazovka, alebo dokonca môže sa objaviť vpopredí iného okna.

Trieda `Activity` je na rozdiel od iných programovacích paradigiem, kde sú aplikácie spúštané volaním metódy `main()`, iniciované systémom Android v inštancii `Activity` vyvolaním špecifických metód, ktoré odpovedajú konkrétnym fázam svojho životného cyklu.

Pri interakcii užívateľa prechádzajú jednotlivé inštancie triedy `Activity` rôznymi stavmi. Týchto stavov je šesť a konkrétne sú to stavy *Created*, *Started*, *Resumed*, *Paused*, *Stopped*, *Destroyed*. Trieda `Activity` nám poskytuje niekoľko spätných volaní, ktoré dovoľujú aktivite vedieť, že sa nejako zmenil ich stav na základe ktorého, systém vytvára, zastavuje, obnovuje alebo ničí proces, v ktorom je daná aktivita umiestnená. Fázy životného cyklu a spätné volania, ktoré ich indikujú, si popíšeme nižšie a je ich možné tiež vyjadriť aj orientovaným grafom na obrázku 2.2.

Životný cyklus aktivity a jej spätné volania

- **`onCreate()`** - volanie, ktoré musí byť implementované. Toto volanie je spustené pri prvom vytvorení danej aktivity. V tomto bode vstupuje aktivita do stavu *Created*. V tejto metóde by mala byť implementovaná základná logika, ktorá sa vykoná len raz počase celej životnosti aktivity. Metóda obsahuje parameter `savedInstanceState` typu `Bundle`, obsahujúci predtým uložený stav aktivity. Ak je aktivita vytváraná prvý krát, hodnota tohto parametru je `null`. Aktivita ale hneď po spustení tejto metódy prejde do stavu *Started* a systém rýchlo za sebou zavolá metódy `onStart()` a `onResume()`.

- **`onStart()`** - ak sa aktivita už nachádza v stave *Started*, systém vyvoláva toto spätné volanie. Metóda má na starosti to, že robí aktivitu viditeľnú pre používateľa a pripravuje ju na možnú interakciu. Na tomto mieste by sa mal inicializovať kód, ktorý udržuje používateľské rozhranie, alebo registrovať príjmač `BroadcastReceiver`, ktorý sleduje zmeny, na základe ktorých sa potom mení používateľské rozhoranie. Po vykonaní tejto metódy aktivita prejde do ďalšieho stavu *Resumed* a volá metódu `onResume()`.

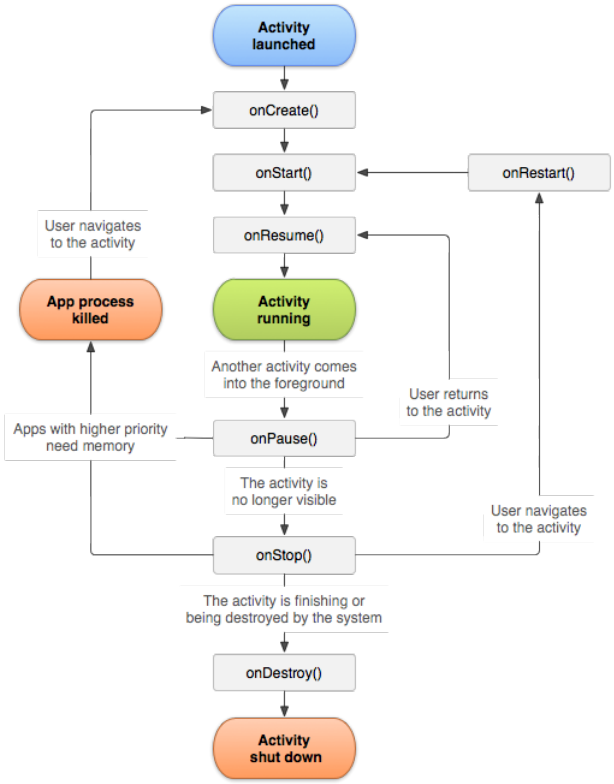
- **`onResume()`** - ak aktivita prejde do stavu *Resumed*, čo môže nastať buď prechodom zo stavu *Started*, alebo *Paused*, je vyvolaná táto metóda. V tomto stave aplikácia komunikuje s používateľom a reaguje na jeho interakciu, čo môže byť vybranie rôznych prvkov aplikácie, klepnutím, či iným povoleným gestom na dispeleji hardvéru, na ktorom tento operačný systém beží. V tejto metóde by sa mali inicializovať komponenty, ktoré neskôr v metóde `onPause()` uvoľníme. Aktivita a s ňou celá aplikácia ostáva v tomto stave, až kým nenastane udalosť podobná prejdeniu užívateľa do inej aktivity, prijatia hovoru, či vypnutie obrazovky zariadenia, kedy daná aktivita stráca pozornosť. Ak dôjde k tejto udalosti, aktivita prejde do stavu *Paused* a vyvolá sa metóda `onPause()`.

- **`onPause()`** - systém zavolá túto metódu ako prvú indikáciu toho, že používateľ opúšta danú aktivitu. Nie vždy to znamená, že užívateľ aplikáciu zruší a musí dôjsť k volaniu metódy `onDestroy()`. V tomto momente sa aktivita nachádza v stave *Paused* a v tejto metóde by mali byť implementované veci ako zastavenie rôznych animácii, prehrávanie médií, alebo uvoľnenie rôznych komponent. Vykonanie tejto metódy však tiež neznačí prechod do iného stavu. V tomto stave aktivita ostáva, dokiaľ nedôjde znovu k obnoveniu aktivite, v tomto prípade sa stav zmení na *Resumed* a systém vráti uloženú inštanciu danej aktivity. Ak je aktivita úplne neviditeľná, stav aktivity sa zmení na *Stopped* a systém zavolá metódu `onStop()`.

- **`onStop()`** - ak je už aktivita neviditeľná pre používateľa, vstupuje do stavu *Stopped*. To sa môže stať napríklad vtedy, ak je spustená nová aktivita, ktorá pokrýva celú obrazovku. Tu by sa mali začať uvoľnovať zdroje, ktoré používateľ ďalej nepotrebuje.

Tiež by malo dôjsť k uvoľneniu príjmača `BroadcastReceiver`, ak bol zaregistrovaný v metóde `onStart()`. Aby sa predišlo pretečeniu pamäti je dôležité v tomto mieste uvoľniť všetky prostriedky, ktoré programátor použil, nakoľko je možné, že systém zabije proces bez toho, aby zavolal poslednú metódu, životného cyklu aktivity, metódu `onDestroy()`. Z tohto stavu môže byť aktivita vrátená k interakcii s používateľom a systém zavolá metódu `onRestart()` a prejde do stavu *Started*, alebo dôjde k ukončeniu aktivity a systém zavolá metódu `onDestroy()` a prejde do stavu *Destroyed*.

- `onDestroy()` - metóda, ktorá je volaná pred zničením aktivity. Je to posledná metóda, ktorú aktivita dostáva a v tomto momente ju systém vyvolá, pretože bola vyvolaná metóda `finish()`, alebo systém ničí proces obsahujúci túto aktivitu kvôli šetreniu zdrojov. Táto metóda môže byť volaná aj v prípade, že dôjde k zmene orientácie obrazovky no hneď na to, je volaná metóda `onCreate()`, aby sa obnovil proces a s ňou aj komponenty, ktoré obsahuje a došlo tak k prekresleniu obrazovky. Táto metóda uvoľnuje všetky zdroje, ktoré neboli uvoľnené skoršími volaniami ako je napríklad metóda `onStop()`.



Obr. 2.2: Životný cyklus aktivity

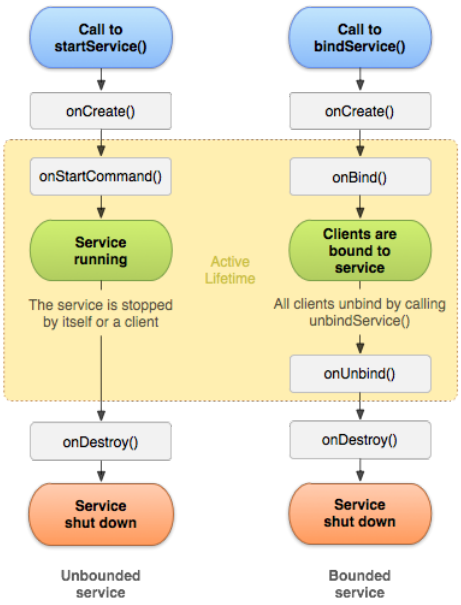
2.3 Služby (dlhotrvajúce operácie) v systéme Android.

Služba je komponenta aplikácie, ktorá má na starosti vykonávanie dlhodobých operácii na pozadí, bez užívateľského rozhrania. Službu môže spustiť iná komponenta, ako napríklad aplikácia, a táto služba beží na pozadí naďalej aj keď dôjde k prepnutiu na inú aplikáciu.

Služba môže z pozadia spracovávať sieťové transakcie, prehrávať hudbu, spracovávať vstupy/výstupy súborov, alebo komunikovať s poskytovateľom obsahu. Existujú 3 rôzne typy služieb:

- **Foreground** - ide o službu, ktorá sa vykonáva na popredí a je teda viditeľná pre používateľa a sú indikované zobrazením notifkácie v status bare. Ako príklad si môžme uviesť prehrávanie hudby. Služby tohto typu pokračujú vo vykonávaní, aj keď používateľ nijako neinteraguje s aplikáciou.
- **Background** - služba, ktorá vykonáva operáciu na pozadí a používateľ ju tak priamo nezaznamenáva. Ide napríklad o získavanie aktuálnej polohy užívateľa.
- **Bound** - službu nazveme bound práve vtedy, ak sa k nej viaže nejaká komponenta (napríklad Aktivita), ktorá využíva volanie metódy `bindService()`. Táto služba ponúka rozhranie klient-server, ktorý umožňuje komponentám komunikovať so službou, posielať žiadosti, príjmať ich výsledky a komunikovať medzi procesmi. Služba beží len do doby, kým je k nej naviazaná iná komponenta, pri odviazaní sa služba zničí.

Ako môžme vidieť na obrázku 2.3, životný cyklus služieb je jednoduchší ako vyššie životný cyklus aktivít.



Obr. 2.3: Životný cyklus služieb

Samotný životný cyklus služby, jeho vytvorenie až zničenie, môže vzniknúť z týchto dvoch možností

- **Spustenie služby** - pri tejto možnosti, je služba vytvorená, ak iná komponenta zavolá metódu `startService()`. Táto služba beží po dobu neurčitú a musí sa sama zastaviť vyvolaním metódy `stopSelf()`. Tiež ju môže môže zastaviť iná komponenta volaním metódy `stopService()`. V prípade, že je služba zastavená, systém ju zničí.
- **Viazaná služba** - druhá možnosť je tá, že služba je vytvorená keď iná komponenta (klient) volá metódu `bindService()`. Klient potom komunikuje so službou

prostredníctvom rozhoranie `IBinder`. Klient môže uzavrieť toto spojenie volaním metódy `unbindService()`. Na túto službu sa môžu viazať viacerí klienti, a ak sa všetci odpoja, je služba zničená systémom. Táto služba sa nemusí byť zastavovaná sama sebou.

Tieto dve možnosti nie sú úplne oddelené a môžme sa naviazať aj na službu, ktorá bola spustená vyvolaním metódy `startService()`. Implementovaním týchto metód môžme potom sledovať dve vnorené cykly životného cyklu služby. Prvý cyklus, ktorý môžeme pozorovať na obrázku 2.3 vľavo, sa sústreďuje na čas, medzi volaniami metód `onCreate()`, kde môže služba podobne ako aktivita inicializovať rôzne komponenty, alebo vytvárať nové vlákna, a metódou `onDestroy()`, kde dôjde k uvoľneniu všetkých použitých zdrojov. Druhý cyklus môžme vidieť na rovnakom obrázku vpravo, kde aktívna životnosť služby začína volaním metódy `onStartCommand()`, alebo `onBind()`. Každá metóda je potom spracovávaná v zmysle `Intentu`, ktorý bol predávaný ako parameter metódam `startService()`, alebo `bindService()`. Ak je služba spustená, jej aktívna životnosť sa končí súčasne s ukončením celej životnosti a ak je služba viazaná, tak aktívna životnosť končí návratom z metódy `onUnbind()`.

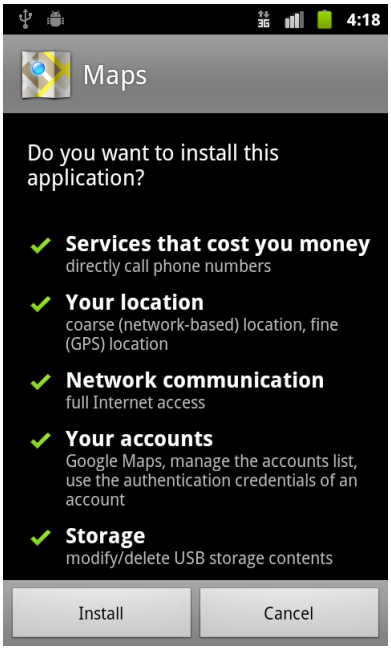
2.4 Zabezpečenie

Systém Android obsahuje prvotriedne bezpečnostné funkcie a spolupracuje s vývojarmi a implementátormi zariadení, aby udržali platformu Andorid a celý ekosystém bezpečný. Robustný a bezpečnostný model je nevyhnutný na to, aby bol zabezpečený silný, energický ekosystém pre aplikácie a zariadenia postavené na tejto platforme.Výsledkom toho je aj to, že celý životný cyklus vývoja Androidu podlieha prísnemu bezpečnostnému programu. Systém Android je navrhnutý tak, aby chránil dôvernosť, integritu a dostupnosť používateľov, dát, aplikácii, zariadenia a siete.

Android bol navrhnutý s viacvrstvovou bezpečnosťou, ktorá je dostatočne flexibilná na to, aby podporovala otvorenú platformu, akou je Android a zároveň chránila všetkých používateľov. Android ponúka programátorom aplikácii pomoc pri riešení bezpečnostných otázok a to vydávaním stabilnej platformy. Tiež existuje tím, ktorý sa o bezpečnosť stará a kontroluje aplikácie a ich potencionálne zraniteľné miesta a navrhuje programátorom spôsoby, ako tieto problémy riešiť. Systém Android nezabúda ani na používateľov a dovoľuje im chrániť svoje súkromie tým, že im zobrazuje oprávnenia, ktoré môže daná aplikácia vykonávať (obrázok 2.4), a ak sa používateľovi nepáčí, môže oprávnenie zakázať.

Android beží na širokej škále hardvérových konfigurácii a hoci je procesorovo-agnostický (beží na rôznych procesoroch rovnako), využíva výhody určitých hardverových bezpečnostných funkcií, ako je napríklad `ARM eXecute-Never`[1] . Samotný operačný systém Android je postavený na jadre linuxu. Všetky prostriedky zariadenia, ako sú napríklad funkcie fotoaparátu, údaje GPS, telefónne funkcie, sú prístupné cez operačný systém. Aplikácie určené pre platformu Android sú najčastejšie písané v programovacom jazyku `Java` a bežia v Android runtime (ART). Aplikácie bežia v rámci bezpečnostného prostredia, ktorý je obsiahnutý v rámci aplikačného sandboxu. Aplikácie tak získavajú osobitnú časť súborového systému, v ktorom môžu zapisovať súkromné údaje, vrátane databáz a nespracovaných súborov.

[1] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0360f/CACHFICI.html



Obr. 2.4: Príklad oprávnení, ktoré aplikácia požaduje

Zabezpečenie na úrovni jadra

Základom pre mobilné počítačové prostredie je jadro linuxu s operačným systémom Android s niekoľkými kľúčovými bezpečnostnými funkciami, ktorými sú užívateľsky založený model oprávnení, izolovanie procesov, rozšírený mechanizmus pre bezpečnú medziprocesorovú komunikáciu a schopnosť odstrániť nepotrebné a potenciálne neisté časti jadra. Základným bezpečnostným cieľom jadra linuxu je navzájom izolovať medzi sebou zdroje používateľov. Linux tak zabraňuje používateľovi A čítať súbory používateľa B, zabezpečuje, aby používateľ A nevyčerpal pamäť používateľa B, zabezpečuje, aby používateľ nevyčerpal zdroje používateľa B a zabezpečuje, aby používateľ A nevyčerpal zariadenia používateľa B (napríklad GPS, Bluetooth).

Aplikačný sandbox

Systém Android priraďuje každej aplikácii jedinečné používateľské ID (UID) a spúšta ho ako samostatný proces. Tento prístup sa líši od ostatných operačných systémov, kde sa používajú viaceré aplikácie s rovnakými oprávneniami používateľa. Jadro vynucuje bezpečnosť medzi aplikáciami a systémom na úrovni procesov prostredníctvom štandardných vlastností linuxu, ako sú identifikátory používateľov a skupín, ktoré sú priradené aplikáciam. Ak sa napríklad aplikácia A pokúsi urobiť niečo škodlivé, ako je čítanie údajov aplikácie B, operačný systém to chráni, pretože aplikácia A nemá príslušné používateľské privilégia. Sandbox je teda jednoduchý, kontrolovateľný a založený na oddeľovaní procesov a oprávnení v štýle `UNIX`-u.

Vďaka tomu že je aplikačný sandbox v jadre, sa tento bezpečnostný model rozširuje aj na natívny kód a aplikácie operačného systému. Všetok softvér, ktorý beží nad jadrom, ako sú knižnice operačného systému, alebo aplikácie, bežia v aplikačnom sandboxe. V systéme Android neexistujú žiadne obmedzenia, ako písať aplikáciu, tak aby bola docieľená potrebná bezpečnosť a v tomto ohľade je natívny kód rovnako bezpečný ako aj interpretovaný kód.

Aby sa predošlo tomu, že dôjde k poškodeniu pamäte v jednej aplikácii a neohrozilo to ostatné aplikácie a bezpečnosť zariadenia, bežia aplikácie v aplikačnom sandboxe na úrovni operačného systému. Takto chyba v pamäti umožní ľubovoľné spustenie kódu len v kontexte konkétnej aplikácie s povoleniami, ktoré vytvoril operačný systém. Treba však podotknúť, že tak ako všetky bezpečnostné funkcie, ani aplikačný sandbox nie je nezlomný.

Kryptografia

Pre aplikácie Android tiež poskytuje sadu kryptografických API. Patria k nim implementácie štandardných a bežne používaných kryptografických primitív, ako sú `AES`, `RSA`, `DSA`, `SHA`. Rozhrania API sa však používajú aj na protokoly vyššej úrovne ako sú protokoly `SSL` a `HTTPS`. Vo verzii Android 4.0 bola predstavená trieda `KeyChain`, ktorá umožňuje aplikáciam používať úložisko systémových poverení pre súkromné kľúče a reťazce certifikátov.

Konfigurácia zabezpečenia siete

Táto funkcia umožňuje aplikáciam prispôsobiť nastavenia zabezpečenia siete v bezpečnom deklaratívnom konfiguračnom súbore bez úpravy kódu aplikácie. Tieto nastavenia je možné nakonfigurovať pre konkrétne domény a pre konkrétnu aplikáciu. Kľúčové funkcie:

- **Vlastné nastavenie dôvernosti** - dochádza k modifikácii certifikačných autorít, ktoré aplikácia považuje za dôveryhodné. Napríklad dôvera konkrétnym certifikátom s vlastným podpisom, alebo obmedzenie súboru verejných certifikačných autorít, ktorým aplikácia verí.
- **Prepisy len pri ladení** - pre ladiace účely, kedy sa neohrozuje základňa používateľov, ktorí aplikáciu už používajú.
- **Obmedzenie návštevnosti** - chránenie aplikácie pred pripojeniami pomocou nešifrovaného `HTTP` protokolu namiesto protokolu `HTTPS`
- **Pripojenie certifikátu** - obmedzenie, aby aplikácia dôverovala len k pripojeným certifikátom.

2.5 Prístup k zdrojom

Okrem vyšie spomenutého nám operačný systém Android ponúka tiež možnosti pristupovať k zdrojom zariadenia. Týmito zdrojmi sú úložisko súborov, poloha zariadenia (známa ako GPS), kamera, senzory a rôzne ďalšie prvky konektivity, ktoré nám dovoľujú spájať sa s inými zariadeniami a patria medzi nich napríklad technológia Bluetooth, NFC, USB.

Úložište súborov

Android ponúka niekoľko možností ako je možné ukladať údaje aplikácii. Ich použitie sa líši od veľkosti, či viditeľnosti pre ostatné aplikácie. Poďme si bližšie popísať aké možnosti nám systém ponúka:

- **Interné úložisko súborov** - v predvolenom nastavení sú súbory uložené do interného ukladacieho priestoru a k týmto súborom nemôžu pristupovať ostatné aplikácie. Pri odinštalovaní aplikácie sú súbory nachádzajúce sa v tomto úložisku odstránené.

Android nám tiež ponúka možnosť uchovávať niektoré údaje len dočasne, k čomu slúži špeciálny adresár vyrovnávacej pamäte, ktorý má každá aplikácia, avšak pri zaplnení interného úložiska nám tieto súbory systém zmaže. Pri odstránení sa tieto súbory taktiež odstránia.

- **Externé úložisko súborov** - tento úložný priestor je nazvaný externým, pretože prístup k nemu nie je zaručený. Používatelia môžu tento priestor pripojiť k počítaču ako externé pamäťové zariadenie a môže byť aj fyzicky odstrániteľné (SD karta). Súbory uložené na externom úložisku sú čitateľné pre všetkých používateľov a môžu byť modifikované. Toto úložisko by malo slúžiť na ukladanie používateľských údajov, ktoré by mali byť prístupné iným aplikáciam a mali by ostať uložené aj v prípade, že užívateľ aplikáciu odinštaluje. Súbory je možné ukladať aj do adresáru špecifického pre danú aplikáciu, najmä ak ide o veľké dáta a tieto údaje sa pri odinštalovaní aplikácie odstránia.

- **Zdieľané preferencie** - toto riešenie je vhodné, ak nie je potrebné ukladať množstvo informácií a nevyžaduje sa ani žiadna štruktúra. Ide vlastne o ukladanie primitívnych dát vo formáte kľúč-hodnota. Pre zjednodušenie práce s týmito primitívami existuje rozhranie `SharedPreferences`, ktoré uľahčuje ukladanie týchto typov. Toto rozhranie ukladá páry kľúč-hodnota do súborov `XML`, ktoré pretrvávajú v reláciach používateľov, aj keď je aplikácia zabitá. Úložište je vhodné pre ukladanie jednoduchých dátovych typov a údajov, ako je napríklad uloženie najvyššieho skóre užívateľa.

- **Databázy** - Android poskytuje plnú podporu `SQLite` databázam. Pri vytváraní databázy, je databáza prístupna len aplikácii, ktorá ju vytvorila a je vhodná pre ukladanie štrukturovaných údajov.

Prístup k poloha zariadenia

Jednou z jedinečných funkcií mobilných aplikácii je informovanosť o polohe. Používatelia mobilných zariadení si so sebou berú svoje zariadenia a pridávanie informácii o lokalite do aplikácie ponúka používateľom viac kontextuálny zážitok. Rozhranie pre poskytovanie lokality umožňujú do aplikácie pridávať informáciu o polohe pomocou automatického sledovania polohy, geofencingu a rozpoznávaniu činností.

Snímače

Väčšina zariadení so systémom Android má vstavané snímače, ktoré merajú pohyb, orientáciu a rôzne podmienky prostredia. Tieto snímače sú schopné poskytovať hrubé dáta s vysokou presnosťou. Sú užitočné pri polohovaní trojrozmerného zariadenia, alebo pri sledovaní zmien okolitého prostredia. Niektoré snímače sú hardvérové, iné zas softvérovo založené. Android podporuje tri široké kategórie senzorov:

- **Snímače pohybu** - snímače, ktoré merajú zrýchlenie a silu otáčania pozdĺž troch osí. Táto kategória záhrňa senzory ako akcelometre, gravitačné senzory, gyroskopy a rotačné vektorové snímače.

- **Snímače prostredia** - tieto snímače merajú rôzne enviromentálne parametre, ako je teplota, tlak okolitého vzduchu, osvedlenie a vlhkosť. Do tejto kategórie spadajú barometre, fotometre a teplomery.

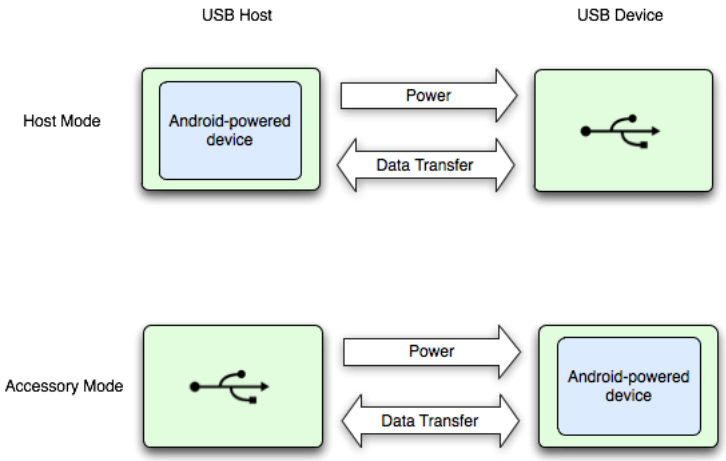
- **Snímače polohy** - merajú fyzickú polohu zariadenia. Tu patria senzory orientácie a magnometre.

Prvky konektivity

Podpora technológie **Bluetooth** nie je výnimkou ani u tohto operačného systému. Táto technológia umožňuje zariadeniam bezdrôtovo vymieňať dáta s inými Bluetooth zariadeniami. Android ponúka aplikačný rámec pre prácu s Bluetooth funkciami. Prosterdníctvom tohto rámcu môžu aplikácie vykonávať skenovanie ďalších Bluetooth zariadení, pripojenie sa k iným zariadeniam prosterdníctvom vyhľadávania služby, preniesť údaje do a z iných zariadení.

Ďalšiou podporovanou je služba **Near Field Communication** skôr známa pod skratkou NFC. Služba umožňuje zdielať malé dáta medzi štítkami NFC a zariadením, ktoré túto technológiu podporuje, alebo medzi dvoma takými to zariadeniami. Existujú jednoduché štítky, ktoré ponúkajú len sématické čítanie a písanie a niekedy umožňujú len čítanie karty. Na druhej strane existujú aj zložité štítky, ktoré ponúkajú rôzne matematické operácie a obsahujú kryptografický hardvér, na overovanie prístupu.

Android podporuje aj rôzne periférie **USB** a to pomocou dvoch režimov a to USB príslušenstvo (USB accessory) a USB hostiteľ (USB host), ktoré sú znázornené aj na obrázku 2.5. V prvom menovaniom funguje externý USB hardvér ako hostiteľ a ako príklad si môžeme uviesť dokovaciu stanicu. V hostiteľskom režime funguje zariadenie ako hostiteľ a medzi príklady patria napríklad digitálne kamery.



Obr. 2.5: USB režimy

Kapitola 3

Použité technológie

V tejto kapitole si predstavíme technológie, ktoré boli pri vypracovávaní tejto práce použité. Najskôr si povieme, čo je to serializácia a ako funguje a následne si predstavíme reflexiu, kde pôjdeme viac do hĺbky. Rozoberieme si, čo reflexia je, aké mechanizmy používa a predstavíme si aj niektoré metódy, ktoré budú v tomto projekte používané. Ďalej si povieme niečo o Wi-Fi Peer-to-Peer, čo je technológia, vďaka ktorej dokážu zariadenia medzi sebou komunikovať v prípade, že sú pripojené na rovnakú Wi-Fi sieť a táto technológia bude použitá pri posielaní si súborov medzi sebou.

3.1 Serializácia

Ako uvádza [3] serializácia objektov v jazyku Java dovoľuje previesť každú inštanciu triedy implementujúcu rozhranie `Serializable`, alebo `Externalizable` na sekvenciu bajtov (serializácia), ktorá môže byť neskôr použitá pre úplnú rekonštrukciu stavov pôvodného objektu (deserializácia). Objekty ktoré sú prevádzané do tejto sekvencie často tvoria vzťah s inými objektami, a tak ak je objekt uložený, uložia sa aj všetky objekty, ktoré su z tohto objektu prístupné, aby nedošlo k poškodeniu vzťahov medzi týmito objektami. Serializované objekty je možné prenášať cez počítačovú sieť, alebo medzi zariadeniami. *[TODO: Vysvetliť na príklade?]*

3.2 Reflexia

Ako hovorí [2] reflexia je schopnosť bežiaceho programu preskúmať sám seba jeho softvérové prostredie a zmeniť svoje správanie. Aby program mohol vykonať toto sebaskúmanie, musí poznať svoju reprezentáciu. Tieto informácie nazývamé `metadáta`. V objektovo orientovanom svete sú `metadáta` organizované do objektov nazývaných `metaobjekty` a samotná kontrola `metaobjektov` počas behu sa nazýva `introspekcia`. Vo všeobecnosti existujú tri techniky reflexie, pomocou ktorých môžme ľahko zmeniť správanie a to priama modifikácia metaobjektu, operácie na používanie `metadát` (napríklad dynamické vyvolanie metód) a intercession, kde ide o reflexnú schopnosť, ktorá mení správanie programu tým, že priamo ovládne toto správanie.

Podľa [4] sa jedná o pokročilú vlastnosť jazyka, ktorá umožňuje vykonávať operácie, ktoré by inak nebolo možné vykonávať, napríklad umožňuje obísť zapúzdrenie (sprístupňuje

privátnych členov). To však znamená riziko narušenia správneho behu programu a aj vďaka tomu, by mala byť reflexia používaná čo najmenej a ľuďmi s dostatočnými znalosťami. Veľkou výhodou, ktorú nám reflexia umožňuje, je tá, že aplikácie sa ľahšie prispôsobujú meniacim sa požiadavkam.

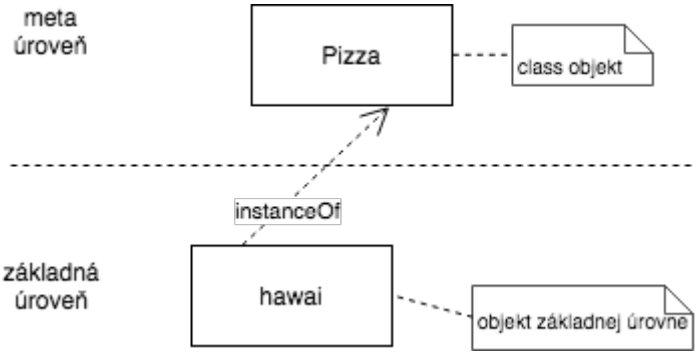
Samotnú reflexiu a introspeckiu si môžme predstaviť tak, že sa na seba pozeráme do zrkadla. Zrkadlo nám poskytuje reprezentáciu samého seba, náš odraz, ktorý môžme skúmať. Skúmanie samého seba v zrkadle nám poskytuje dôležité informácie o tom, čo máme oblečené, či nie sme špinavý, a tiež nám zrkadlo povie niečo o našom správaní. Napríklad či úsmev vyzerá úprimne, alebo či nejaké gesto nevyzerá príliš prehnane. To nám môže pomôcť pre pochopenie toho, ako prispôsobovať svoje správanie aby sme spravili čo najlepší dojem na iných ľudí. Podobne musí pri introspekcii program poznať vlastnú reprezentáciu, ktorá je najdôležitejším štrukturálnym prvkom reflexného systému. Skúmaním vlastnej reprezentácie dokáže program získať správne informácie o svojej štruktúre a správne sa rozhodovať pri vykonávaní dôležitých rozhodnutí.

```
public static void setObjectColor( Object object, Color color ) {
   Class cls = object.getClass();
   Method method = cls.getMethod( "setColor", new Class[] {Color.class} );
   method.invoke( obj, new Object[] {color} );
}
```

Výpis 3.1: Použitie reflexie - pseudokód

Vo výpise 3.1 je znázornené využitie reflexie. Predstavme si nasledujúci problem, ktorý nám vznikol pri implementácii používateľského prostredia, kedy sú použité rôzne štandartné vizuálne komponenty jazyku Java (produkty tretej strany) a tieto komponenty sú integrované do aplikácie. Každá komponenta poskytuje metódu `setColor`, ktorá ako parameter očakáva typ `java.awt.Color` . Problém je v tom, že hierarchia je nastavená tak, že jediná spoločná základná trieda je `java.lang.Object` a tieto komponenty nemôžu odkazovať pomocou typu, ktorý podporuje túto metódu. Metóda `setObjectColor` pomocou reflexie vyžaduje triedu objektu, ktorý bol poslaný v parametroch metódy a následne v triede hľadá metódu `setColor`, ktorá ako parameter vyžaduje typ `Color`. Obe tieto volania sú istou formou introspekcie, ktoré umožňujú aby sa program preskúmal. Posledný riadok volá výslednú metódu objektu a predáva jej parameter typu `Color` - toto volanie možno označiť ako **dynamické vyvolanie**, čo je vlastnosť reflexie, ktorá umožňuje programu vyvolať metódu objektu v dobe behu, bez určenia metódy v čase kompilácie. Toto riešenie zistí, ktorá metóda `setColor` je k dispozícii počas behu programu pomocou introspekcie a vyvolá metódu, ktorá sa práve vyžaduje. V tomto výpise sa tiež používajú inštancie `Class` a `Method` na vyhľadanie odpovedajúcej metódy, a tieto objekty sú súčasťou vlastnej reprezentácie Javy. Tieto objekty nazývame `metaobjekty` a uchovávajú informácie o programe.

Pre lepšie pochopenie `metaobjektov` uvažujme príklad 3.1. Na dosiahnutie hlavného účelu programu hovoríme o objektoch na **základnej úrovni**, čo je na obrázku objekt `hawai`, ktorý je inštanciou objektu `Pizza` a s týmito objektami pracujeme pri vývoji. Objekt `Pizza` je zasa objekt(`metaobjekt`) na **meta úrovni**. `Metaobjekty` sú pri reflexii výhodou a poskytujú všetky informácie, ktoré su potrebné, a často poskytujú aj spôsoby na zmenu štruktúry programu, jeho správania alebo údajov.



Obr. 3.1: Základna a meta úroveň v reflexii

Vyhľadávanie metódy počas behu programu

Mohli sme si všimnúť, že v metóde 3.1, ktorá je uvedená ako príklad, má v parametroch `object`, ktorý je typu `java.lang.Object` a túto triedu dedí každá trieda v Jave. Táto trieda obsahuje metódu `getClass`, ktorá je často používaná pri začatí reflexného programovania, nakoľko mnohé reflexívne úlohy vyžadujú objekty, ktoré reprezentujú triedy. Metóda vracia inštanciu `java.lang.Class` a tieto inštancie sú `metaobjektami`, ktoré Java používa na reprezentáciu tried, ktoré tvoria program. Tieto objekty sú najdôležetejším druhom `metaobjektov`, pretože všetky Java programy sa skladajú z týchto tried.

`Class` objekty nám poskytujú programovacie `metadáta` o `fieldoch` triedy, konštruktoroch, metódach a vnorených triedach. Poskytujú aj informácie o hierarchii dedičnosti a poskytujú prístup k reflexívnym vlastnostniam.

Trieda `Class` nám poskytuje tieto metódy[1] na skúmanie metód:

- `Method getDeclaredMethod(String name, Class[] parameterTypes)` - vracia objekt `Method`, ktorý reflektuje zadanú deklarovanú metódu triedy, alebo rozhrania reprezentovaného týmto `Class` objektom.
- `Method[] getDeclaredMethods()` - vracia pole objektov `Method`, ktoré reflektuje všetky metódy deklarované triedou, alebo rozhranie reprezentované týmto `Class` objektom.
- `Method getMethod(String name, Class[] parameterTypes)` - vracia objekt `Method`, ktorý reflektuje zadanú metódu, ktorá je verejná, alebo rozhoranie reprezentované týmto objektom `Class`.
- `Method[] getMethods()` - vracia pole objektov `Method`, ktoré reflektuje všetky verejné metódy, alebo rozhranie reprezentované týmto `Class` objektom, vrátané tých, ktoré deklaruje trieda, alebo rozhranie a tiež tie, ktoré boli zdedené z rodičovských tried, alebo rodičovských rozhraní.

Trieda `Method` a dynamické vyvolanie

Táto trieda sa nachádza v balíčku `java.lang.reflect` a je to trieda `metaobjektov`, ktorá reprezentuje metódy. Každý objekt tejto triedy poskytuje informácie o metóde, jej návra-

[1] https://docs.oracle.com/javase/7/docs/api/java/lang/Class.html
[2] Zoznam všetkých metód: https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Method.html

tovom type, názvu, typy parametrov, typ výnimky prípadne anotáciu. Objekt `Method` nam taktiež umožňuje zavolať metódu, ktorú predstavuje. Metódy, ktoré okrem iných táto trieda obsahuje[2]:

- `Annotation[] getDeclaredAnnotations()` - vracia všetky anotácie, ktoré sú priamo prítomné v tomto prvku.
- `Class getDeclaringClass()` - vracia objekt `Classs` reprezentujúci triedu, alebo rozhranie, ktoré deklaruje metódu reprezentovanú týmto `Method` objetkom.
- `Class[] getExceptionTypes()` - vracia pole objektov `Class`, ktoré reprezentujú typy výnimiek, ktoré boli deklarované na zahodenie.
- `int getModifiers()` - vracia modifikátory jazyku Java pre metódu reprezentovanú objektom `Method` v podobe celého čísla.
- `String getName()` - vracia názov metódy ako reťazec.
- `Class[] getParameterTypes()` - vráti pole objektov `Class`, ktoré reprezentujú formálne typy parametrov, v poradí ako su deklarované.
- `Class getReturnType()` - vráti objekt `Class`, ktorý reprezentuje formálny návratový typ metódy.
- `Object invoke(Object obj, Object[] args)` - vyvolá metódu reprezentovanú objektom `Method` na zadanom objekte s určenými parametrami.

Dynamické vyvolanie umožňuje programu zavolať metódu na objekt počas behu programu, bez určenia metódy v čase kompilácie. Prvý parameter metódy `invoke` je cieľ volania metódy, alebo objekt, ktorý túto metódu vyvolá. Druhým parametrom na vyvolanie je pole typu `Object`, ktorý metóda `invoke` predá dynamicky vyvolanej metóde ako jej aktúalne parametre.

3.3 Wi-Fi Peer-to-Peer

Wi-Fi peer-to-peer (P2P) je technológia, pomocou ktorej sa zariadenia s príslušným hardvérom dokážu vzájomne prepojiť cez Wi-Fi sieť bez prostredného pristúpového bodu. Táto technológia nám umožňuje skúmať zariadenia, pripojené na rovnakú Wi-Fi sieť, pripojiť sa k nim a vzájomne komunikovať. Wi-Fi peer-to-peer bude použitá vo vzorovej aplikácii, na posielanie spracovavaného projektu medzi dvoma až viacerými zariadeniami.

Kapitola 4

Migrujúce softwarevé komponenty

[TODO: čo to je, existujúce riešenia]

Kapitola 5

Návrh kontajneru pre komponenty bežiace na systéme Android

[TODO: nejaké základne myšlienky, čo a prečo]

5.1 Návrhový diagram

5.2 Architektúra kontajneru

[TODO: uml...popísať, čo je čo]

Kapitola 6

Implementácia

[TODO: popísať len kontajner, alebo spojit to aj s android aplikaciou?]

Kapitola 7

Tvorba vzorovej Android aplikácie

[TODO: všetko k tomu]

Kapitola 8

Záver

[TODO: čo sa podarilo, čo nie, aký je výsledok....]

Literatúra

[1] Developers, A.: *Android developer guide*. [Online; navštíveno 15.03.2018].
URL https://developer.android.com/guide/index.html

[2] Forman, I. R.; Forman, N.: *Java Reflection in Action (In Action Series)*. Greenwich, CT, USA: Manning Publications Co., 2004, ISBN 1932394184.

[3] Genčúr, M.: *Rámec pro dynamickou aktualizaci aplikací v jazyce Java*. [Online; navštíveno 15.03.2018].
URL http://www.fit.vutbr.cz/study/DP/DP.php?id=6955&file=t

[4] Kozák, D.: *Srovnání výkonu a vlastností objektově orientovaných databází*. [Online; navštíveno 23.04.2018].
URL http://www.fit.vutbr.cz/study/DP/BP.php?id=14162&file=t

[5] Rogers, R.; Lombardo, J.; Mednieks, Z.; aj.: *Android Application Development: Programming with the Google SDK*. O'Reilly Media, Inc., první vydání, 2009, ISBN 0596521472, 9780596521479.