Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Modulární aplikace na platformě Android

Bakalářská práce

Autor: Daniel Pichl
Studijní obor: Aplikovaná informatika

Vedoucí práce: doc. Ing. Filip Malý, Ph.D.

Hradec Králové duben 2015

Prohlášení

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne Daniel Pichl

Poděkování

Děkuji doc. Ing. Filipu Malému, Ph.D. za odborné vedení a pomoc v průběhu tvorby bakalářské práce.

Anotace

Obsahem této práce je popis tvorby aplikace podporující modularitu v reálném čase pro platformu Android. Cílem aplikace je umožnit okamžité zavádění a spouštění uživatelem vybraných modulů. Tyto moduly budou k aplikaci dodávány ve formě samostatných souborů. Jejich funkcionalita může zahrnovat širokou škálu možností a nahrazovat jiné jednotlivé aplikace, jejichž schopnosti tím spojí a zjednoduší využití zařízení s platformou Android.

Kvůli některým specifickým vlastnostem systému Android nemusí být možné dosáhnout ideální funkčnosti této aplikace. V práci je tedy zahrnuta i definice a postup řešení případných omezení při snaze aplikaci vytvořit.

Annotation

Title: Modular Application for the Android Platform

The contents of this thesis describe the process of creation of an application which supports real-time modularity on the Android platform. The goal of the application is to enable the user to immediately load and run modules of his choice. These modules will be supported in the form of stand-alone files. The modules' functionality can include a wide range of options and replace other, separate applications, bundling their capabilities and making the use of an Android device easier.

The ideal functionality of this application might not be reached on account of some specific characteristics of the Android system. The thesis therefore includes definitions and methods of resolution of limitations encountered in the process of development of the application.

Obsah

1 Úvod **1**

2 Cíl a požadavky **2**

3 Uvedení do problematiky **3**

3.1 Modularita 3

3.1.1 Modularita obecně *3*

3.1.2 Modularita v Javě *3*

3.2 Aplikace na platformě Android 5

3.3 Předpokládané problémy 6

4 Analýza **7**

4.1 Modulární aplikace pro osobní počítače 7

4.1.1 Zjednodušení skrze Groovy *8*

4.2 Problematika kompilace třídy na platformě Android 10

4.3 Dynamické načítání externích aktivit 11

5 Existující řešení **12**

5.1 Spouštění dynamických Groovy skriptů 12

5.1.1 Knihovny pro podporu Groovy *12*

5.1.2 Grooidshell Example *14*

5.2 Načítání externích aktivit s využitím reflexe 16

6 Návrh řešení **18**

6.1 Aplikace využívající Groovy 18

6.1.1 Struktura modulu *18*

6.1.2 Struktura aplikace *20*

6.2 Aplikace využívající Javu 22

6.2.1 Struktura modulu *22*

6.2.2 Struktura aplikace *22*

7 Implementace **23**

7.1 Aplikace GyModular 23

7.1.1 Třídy logiky načítání modulů *23*

7.1.2 Třídy grafického uživatelské rozhraní *29*

7.1.3 Závěr *32*

7.2 Aplikace ApkModular 33

7.2.1 Třídy logiky načítání modulů *33*

7.2.2 Třídy grafického uživatelského rozhraní *36*

7.2.3 Závěr *36*

7.3 Srovnání výkonu aplikací 37

8 Shrnutí výsledků **38**

9 Závěr a doporučení **39**

10 Citovaná literatura **40**

Seznam obrázků

Obrázek 1 (18) – Schéma kompilace a spuštění Java a Groovy tříd 13
Obrázek 2 – Snímek obrazovky aplikace grooidshell-example...15
Obrázek 3 – Struktura aplikace GyModular...21
Obrázek 4 – Snímek obrazovky aplikace GyModular 30

Seznam ukázek kódu

Ukázka kódu 1 – Metoda pro načítání externích Java tříd ... 7
Ukázka kódu 2 – Modulární potomek JPanel v Javě ... 9
Ukázka kódu 3 – Modulární potomek JPanel v Groovy ... 9
Ukázka kódu 4 – Načítání aktivity reflexí ... 17
Ukázka kódu 5 – Groovy třída ClockFragment ... 20
Ukázka kódu 6 – Metoda parseClass třídy GrooidShell ... 27
Ukázka kódu 7 – Metoda defineDynamic třídy GrooidShell ... 28
Ukázka kódu 8 – Metoda loadFragment aplikace ApkModular ... 35

1 Úvod

V současném rychle se rozvíjejícím světě informací a informačních technologií je mobilita a univerzálnost výpočetních systémů stále více nezbytná. Díky miniaturizaci technologií se chytré telefony, tzv. smartphone, a tablety staly dostupnou a nedílnou součástí běžného života.

Obdobná zařízení s maximálním výkonem při minimální velikosti a hmotnosti mají ovšem využití nejen jako komunikační či zábavní zařízení v rukách běžných uživatelů. Zejména díky otevřené systémové platformě Android mohou být tyto technologie využívány v průmyslu, vědě i v armádě.

Systém Android, stojící na základech Unixových systémů, nabízí univerzální řešení řízení velkého množství hardwarových platforem, a proto byl zvolen jako cílové prostředí této práce. Jejím předmětem je vytvořit aplikaci, která může tuto univerzálnost ještě více prohloubit.

Univerzálnost a modularita jsou si v informačních technologiích předměty blízké. Modularita v rámci této práce znamená aplikaci, která by sama o sobě neměla žádné využití, kromě načítání modulů. Tyto moduly by aplikace měla načítat v reálném čase, tzn. okamžitě po vybrání souboru s modulem, bez nutnosti jakékoliv další manipulace s aplikací či modulem ze strany uživatele. Takto načtený modul se ihned spustí a provede svou funkci. Ta může být takřka jakákoliv, od zobrazení hodin či kalendáře po úkoly složitější, například využívající kameru nebo síťové připojení. Tímto je získána teoreticky neomezená univerzálnost díky modularitě.

Aplikace v platformě Android jsou založeny na jazyce Java a spouštěny díky tzv. *Dalvik Virtual Machine*. Nejjednodušeji řečeno, Dalvik VM je virtuální stroj, který spouští aplikace a kód napsaný v Javě. (1) Java, jako jeden z nejrozšířenějších programovacích jazyků, umožňuje tvorbu aplikací, které jsou univerzální nejen širokým využitím v systému Android, ale také díky možnosti v jádru tutéž aplikaci spouštět na jiných platformách, např. PC.

Modularita aplikací v Javě je poměrně běžná záležitost, jejíž realizace nepředkládá veliké překážky. Platforma Android a její specifický způsob překládání a spouštění aplikací některé překážky přináší. Tato práce se zaměří mimo jiné na způsoby překonání těchto překážek.

2 Cíl a požadavky

Cílem této práce je vytvoření aplikace pro operační systém Android verze 4.4, ideálně se zpětnou kompatibilitou až k verzi 2.2. Aplikace by měla fungovat na všech zařízeních s daným operačním systémem, tzn. bez bližších hardwarových požadavků, týkajících se výkonu zařízení. Ovládání dotykem je samozřejmostí. Je nicméně možné, že při tvorbě aplikace dojde ke zjištění nutnosti řešit konkrétní hardwarové požadavky.

Hlavní okno aplikace se bude spouštět na celou obrazovku a ve výchozím stavu bude obsahovat pouze menu pro otevření modulu. Modul bude možné otevřít odkudkoliv z paměti zařízení, přičemž bude po otevření nahrán jak do operační paměti, tak do vyhrazeného místa v paměti zařízení, odkud bude možné ho při příštím spuštění aplikace načíst automaticky.

Po načtení se vybraný modul okamžitě zobrazí v okně aplikace, kde vykoná svou funkci. Pro účely práce budou vytvořeny moduly s čistě demonstračními vlastnostmi, jako je například zobrazování aktuálního času.

V práci budou popsány všechny vyzkoušené způsoby, jejichž záměrem je takové funkcionality dosáhnout, a to i ty, které k úspěšnému výsledku nevedou. Pokud bude výsledku dosaženo, bude funkční kopie této aplikace součástí práce.

3 Uvedení do problematiky

3.1 Modularita

3.1.1 Modularita obecně

Běžný pohled na modularitu aplikací pro platformu Android je jiný, než ten, o kterém tato práce pojednává. Popisuje ho následující citace z portálu Android Developers:

Aplikační modularita: Android umožňuje spouštět aplikace podepsané stejným certifikátem v jednom procesu, je-li tak aplikací požadováno, takže je systém chápe jako jedinou aplikaci. Tímto způsobem lze aplikaci zveřejnit v modulech a uživatelé mohou aktualizovat tyto moduly jednotlivě. (2)

Tento pohled znamená, že se modulární aplikace skládá z několika samostatných a teoreticky i samostatně spustitelných aplikací. Aplikací je rozuměn archiv s příponou *apk*, který systém nainstaluje a přeloží do spustitelného kódu. Dochází tedy k překladu každého modulu samostatně a zcela nezávisle. Naopak modul, o který se v práci jedná, by měl být definován jako tzv. zásuvný modul, neboli *plug-in*. Softwarový plug-in je přídavek do programu, který přidává další funkcionalitu. Například plug-in pro Photoshop (jako třeba Eye Candy) může přidat další filtry použitelné pro manipulaci obrázků. (3)

Jedná se tedy o modul, který samostatně bez „zasunutí" do hlavní aplikace, sloužící jako spouštěcí jádro, nevykonává žádnou funkci. Jeho struktura, formát a datový typ je tedy odlišný od struktury samotné aplikace pro Android. Jedním z účelů této práce je zjistit, jaká má být struktura každého modulu, aby bylo dosaženo co nejlepší efektivity při jeho zavádění, úspory prostředků a maximální funkčnosti. To vše bez nutnosti zásadní manipulace s aplikací, do které je modul zaváděn.

3.1.2 Modularita v Javě

Základními kameny aplikace v jazyce Java, která je vyvíjena pro operační systémy osobních počítačů, jsou v průběhu vývoje třídy ve formě souborů s příponou *java*. Jedná se o textové soubory obsahující člověkem čitelný kód tak, jak ho programátor napsal. V tomto stavu nejsou spustitelné a pro vytvoření funkční aplikace je třeba je zkompilovat. O kompilaci se stará nástroj „*javac*", který je součástí *JDK* (*Java Developement Kit). JDK* je vývojové prostředí pro tvorbu aplikací, appletů a komponent s využitím programovacího jazyka Java. (4) Nástroj *javac* čte definice tříd a rozhraní napsaných v programovacím jazyce Java, a kompiluje je do spustitelného kódu v souborech typu *class*. Současně může zpracovat anotace ve zdrojových souborech Javy a třídách. (5)

Rozdíl oproti mnoha programovacím jazykům je ten, že Java kód je kompilován do tzv. Java bytekódu. Bytekód je strojový jazyk pro *Java Virtual Machine (JVM)*. Když *JVM* načte *class* soubor, získá řetěz bytekódu pro každou metodu ve třídě. Tento bytekód je pak spuštěn ve chvíli volání dané metody při běhu programu. Může být spuštěn formou interpretace, *just-in-time* kompilace nebo jakoukoliv jinou technikou, kterou vybral designer daného *JVM* (6). *JVM* je abstraktní výpočetní stroj, který má sadu instrukcí pro správu paměti. Virtuální stroje jsou obvykle používány pro implementaci nějakého programovacího jazyka. *JVM* je základní kámen programovacího jazyka Java. Je odpovědný za mezi-platformní kompatibilitu Javy a malý objem jejího přeloženého kódu. *JRE (Java Runtime Environment)* je softwarové prostředí, ve kterém je možné spustit programy zkompilované pro typickou implementaci *JVM. JRE* obsahuje kód nezbytný pro běh Java programů, dynamické připojování nativních metod, správu paměti, obsluhu výjimek a implementaci *JVM* (7).

JRE je tedy aplikační prostředí běžící na cílovém systému, které díky *JVM* univerzálně zkompilované třídy přeloží do patřičného strojového kódu a takto přeloženou aplikaci spustí. Ačkoliv z tohoto principu vyplývá ztráta výkonu aplikace (při každém spuštění dochází k jejímu opětovnému překladu), Java díky němu nabízí velké výhody z hlediska platformní nezávislosti a modularity. (7) Principem platformní nezávislosti Javy je skutečnost, že například aplikace zkompilovaná do bytekódu v operačním systému Windows může být spuštěna pomocí *JVM* na OS Linux nebo MacOS a vykonávat tam svou činnosti stejně, jako na systému, na kterém byla naprogramována a testována.

JRE je z hlediska modularity podstatné v tzv. *class loaderu*. Jedná se o prostředek obsažený v *JRE*, ke kterém lze programově přistupovat v Java kódu a dynamicky jím načítat třídy. Koncept *class loader*, jeden ze základních prvků *Java Virtual Machine*, popisuje postup konverze vybrané třídy do bitů zodpovědných za implementaci této třídy. Každý *Java Virtual Machine* obsahuje jeden *class loader*, který je v něm zabudován. (8) Díky možnosti instanciace *class loaderu*, tj. vytvoření objektu v kódu *Java* aplikace, se kterým lze programově pracovat, je tedy možné do běžící aplikace načíst libovolné třídy ve formátu *class* z prostředí nebo sítě, nebo i třídu vytvořit z kódu napsaného přímo uživatelem aplikace. Tyto externí třídy musejí být stále zkompilovány pomocí *javac*.

3.2 Aplikace na platformě Android

S ohledem na obsah práce je nutné přiblížit některé základní stavební principy aplikací pro Android. Oproti Java aplikacím pro osobní počítače mají určitá užší specifika a pravidla, kterých se musí vývojáři držet.

Základem každé Android aplikace je jedna nebo více tzv. aktivit (třída *Activity*). Aktivita je aplikační komponenta, která zobrazuje obrazovku, se kterou může uživatel interagovat za účelem vykonání nějaké činnosti, jako například vytočení telefonu, vyfocení snímku, poslání emailu nebo zobrazení mapy. Každá aktivita má dané okno, ve kterém je vykreslováno uživatelské rozhraní. Toto okno typicky vyplňuje obrazovku, ale může být i menší a plavat nad ostatními okny. (9) Jedná se o třídu, která implementuje specifické metody volané systémem a umožňuje tak uživateli k aplikaci přistupovat. Každé jedno okno aplikace je obvykle právě jedna aktivita.

Komunikace mezi aktivitami probíhá pomocí tzv. *intent*, čili záměr. Z hlediska systému Android je *intent* jasně uchopitelný objekt se specifickým datovým typem. Může přenášet různé informace v doprovodných proměnných, ale v základu je využíván zejména jako rozhodnutí o tom, která aktivita se má pozastavit a skrýt a která jiná zobrazit. *Intent* je objekt zajišťující spojení ve vykonávání oddělených komponent – jako například dvou aktivit. *Intent* reprezentuje „záměr aplikace něco udělat“. Dá se použít pro širokou škálu úkonů, ale obvykle se používá pro spuštění jiné aktivity. *Intent* může nést kolekci různých datových typů jakožto páry klíč-hodnota, nazývané *extras*. Metoda *putExtra* přijímá název klíče jako první parametr a jeho hodnotu jako druhý. (10)

Předpokládaná problematika aktivit a zpráv v souvislosti s modularitou je popsána v kapitole 3.3.

Aplikace se dále skládá z několika *xml* souborů. Nejdůležitější z nich je *AndroidManifest.xml*, kterému se rovněž blíže věnuje kapitola 3.3. V dalších *xml* souborech je definována vizuální stránka aplikace. Konkrétní obsah souborů popisujících vizuální stránku aplikace je mimo rozsah této práce.

3.3 Předpokládané problémy

Dva z předpokládaných problémů při vývoji modulární aplikace souvisí se souborem *AndroidManifest.xml*. Jedná se o soubor ve značkovacím jazyce, který právě jeden náleží k aplikaci a informuje systém o řadě jejích aspektů. Každá aplikace musí mít soubor *AndroidManifest.xml* (s přesně tímto názvem) ve svém kořenovém adresáři. Tento tzv. manifest uvádí systému Android základní informace o aplikaci, informace, které musí systém mít před spuštěním jakékoliv části kódu aplikace. (11)

Jednou z konkrétních informací uváděných v *AndroidManifest.xml* je seznam práv a pověření aplikace. Jedná se o vlastnosti systému, zařízení nebo jiných aplikací, ke kterým smí aplikace přistupovat a případně s nimi manipulovat. Při tvorbě tohoto seznamu vychází autor z předem daných možností. Jedná se například o přístup ke kontaktům v telefonu, připojení k internetu, stavu modulu GPS atp.

První problém spočívá v tom, že aplikace, do které jsou moduly zaváděny, nemůže dopředu znát pověření nutná pro běh všech zaváděných modulů. Pokud by v teoretické situaci byl zaveden modul, který potřebuje pověření, které aplikace nemá, došlo by při jeho vykonání k chybě. Jednoduchým, ale nekorektním řešením je manifest vytvořený tak, aby umožnil aplikaci přístup bez výjimky všude. Veškerá oprávnění musí schválit uživatel při instalaci a takováto aplikace by se pak jevila nedůvěryhodnou. Teoreticky by mohla třetí strana vytvořit modul cíleně škodící uživateli, kterému by nebránilo nic v práci.

Druhý problém je v dalším prvku manifestu, tj. informaci o struktuře aplikace. Manifest popisuje komponenty aplikace – aktivity, služby, přijímače dat a poskytovatele obsahu, ze kterých se aplikace skládá. Jmenuje třídy implementující jednotlivé komponenty a publikuje jejich schopnosti (například, kterou zprávu *intent* mohou vyřídit). Tyto deklarace informují systém Android o tom, o jaké komponenty se jedná a za jakých podmínek mohou být spuštěny. (11)

Podle těchto informací je třeba dopředu znát přesnou strukturu aplikace z hlediska hlavních tříd (aktivit) a zpráv mezi nimi. Pokud by ale byly zaváděné moduly ve formě nových tříd, které by byly načteny do aplikace, systém by o jejich existenci nevěděl a nemohl je používat, tudíž by nastala chybová situace. Z hlediska zpráv mezi třídami se jedná zejména o přepínání jednotlivých již spuštěných modulů při zachování stavu aplikace a ostatních modulů. Manifest nemůže uvádět, jaké zprávy budou moduly zasílat a jaká část aplikace je má přijmout.

Předpokládané řešení je možné formou předpřipravených tříd, které se načtením modulu jen „naplní“ funkcionalitou. To by znamenalo, že moduly musí bez výjimky splňovat předem danou strukturu a formát a pravděpodobně by to jejich funkcionalitu mohlo omezit.

Druhý problém, tj. nutnost deklarovat strukturu aplikace, zejména její aktivity, se také dá obejít metodou nahrazování instancí tříd, viz kapitola 5.2.

4 Analýza

4.1 Modulární aplikace pro osobní počítače

V modulární aplikaci je z důvodů efektivní instanciace vhodné načítat externí třídy, které dědí vlastnosti nějaké nativní Java třídy a spouštět je jako instance této nadřazené třídy, což ilustruje Ukázka kódu 1. Tento způsob umožňuje zobrazit uživatelské rozhraní modulu voláním předem známých metod.

Metoda *findJavaClasses* vrací seznam instancí třídy *JPanel*, je tedy nezbytně nutné, aby *class loaderem* načítané třídy byly potomky této třídy. Metoda v argumentu obdrží název složky, ve které má třídy hledat. Po kontrole validity složky na řádku 4 prohledá v ní vnořené soubory (řádek 9), a pokud jsou typu *class*, uloží si jejich název do seznamu. Na řádku 14 a 15 projde tento seznam a načte *class loaderem* třídy. Tyto pak na řádku 19 a 20 zinstanciuje jako třídu *JPanel* a vloží do seznamu k navrácení.

```
01 public ArrayList<JPanel> findJavaClasses(String directory) {
02   panels = new ArrayList<JPanel>();
03   File dir = new File(directory);
04   if (dir.exists() && dir.isDirectory()) {
05      ArrayList<Class> loadedClasses = new ArrayList<Class>();
06      ArrayList<String> classes = new ArrayList<String>();
07      try {
08         ClassLoader classLoader = new URLClassLoader(new URL[] { dir
              .toURI().toURL() });
09         for (File file : dir.listFiles()) {
10            ...
11         // Naplnění seznamu classes
12         }
13
14         for (String className : classes) {
15            loadedClasses.add(Class.forName(className, true,
                 classLoader));
16         }
17
18         for (Class cl : loadedClasses) {
19            JPanel panel = (JPanel) cl.newInstance();
20            panels.add(panel);
21         }
22         return panels;
23      } catch (Exception ex) {
24         ex.printStackTrace();
25      }
26   }
27   return null;
28 }
```

Ukázka kódu 1 – Metoda pro načítání externích Java tříd

4.1.1 Zjednodušení skrze Groovy

Pro usnadnění tvorby modulů aplikace v jazyce Java za účelem maximalizace nezávislosti na prostředcích, tj. *JDK*, byla zvolena možnost nahrazení zdrojového kódu modulů nějakou alternativou k programovacímu jazyku Java.

Jako vhodná alternativa se nabízejí skriptovací jazyky. Skriptovací jazyky jako například *Perl*, *Python*, *Rexx*, *Tcl*, *Visual Basic* a Unixový *shell* reprezentují velice odlišný styl programování oproti programovacím jazykům. Skriptovací jazyky předpokládají, že v prostředí již existuje kolekce využitelných komponent napsaných v jiných jazycích. Skriptovací jazyky nejsou určeny k psaní celých aplikací, jsou určeny primárně ke spojování komponent. (12) Konkrétním výběrem je pak jazyk *Groovy*, který je s Javou úzce spjat. Groovy je obratný a dynamický jazyk pro *Java Virtual Machine*. Staví na síle Javy, ale má silné dodatečné prvky inspirované jazyky jako je *Python, Ruby* a *Smalltalk.* Integruje všechny existující Java třídy a knihovny a je kompilován přímo do Java bytekódu, takže je možné ho použít kdekoliv, kde je možné použít Javu. (13)

Ukázka kódu 2 ilustruje třídu modulu v Javě načítaného metodou, kterou přibližuje Ukázka kódu 1, Ukázka kódu 3 pak totožnou třídu v Groovy.

Tato varianta byla zvolena, protože Groovy skripty a třídy mohou přímo přistupovat k třídám Javy, využívat jejich metody a dědit po nich. Kromě toho ale nabízejí širší možnosti a jednodušší kód, který je možné psát kdekoliv bez nutnosti užití speciálního software, kromě textového editoru. I Javu je možné psát jen v textovém editoru, ale stále je pro její integraci potřeba kompilace pomocí nástroje *javac* a tedy i *JDK*.

K integraci Groovy je k dispozici knihovna pro Javu, která umožní aplikaci psané v Javě používat *GroovyClassLoader*. Ten překládá Groovy třídy (soubory ve formátu **.groovy*) do bytekódu a umožňuje *JVM* je spouštět. Moduly pak aplikace načítá přímo ve formátu *groovy*, což je z pohledu uživatele ekvivalent *java* a nikoliv *class*, takže je jejich kód pro uživatele aplikace čitelný. To sice znamená ztrátu „zapouzdřenosti" pro vývojáře modulu, ale zároveň nabízí transparentnost z hlediska bezpečnosti a kontroly kompatibility.

Samotná instanciace třídy pak po přeložení pomocí *GroovyClassLoader* probíhá totožně, jako kdyby se jednalo o normální třídu Javy.

```
import javax.swing.JLabel;
import javax.swing.JPanel;


public class JavaPanel extends JPanel {

	public JavaPanel() {
		JLabel lbl = new JLabel("Java text!");
		add(lbl);
	}
}
```

Ukázka kódu 2 – Modulární potomek JPanel v Javě

```
import javax.swing.JLabel
import javax.swing.JPanel


class GroovyPanel extends JPanel {

	GroovyPanel() {
		JLabel lbl = new JLabel("Groovy text!")
		add(lbl)
	}
}
```

Ukázka kódu 3 – Modulární potomek JPanel v Groovy

4.2 Problematika kompilace třídy na platformě Android

Postup, který ilustruje Ukázka kódu 1, je aplikovatelný pouze na systémy osobních počítačů, protože ačkoliv Android vychází z Linuxových systémů a aplikace na něj jsou psány v Javě, je svázán užšími specifikacemi. První problém při snaze aplikovat tento postup nastává kvůli faktu, že s třídami ve formě *class* souborů zkompilovanými nástrojem *javac* není možné v Androidu přímo pracovat. Důvodem je specifická forma *JVM*. V Androidu se *JRE* popsané v kapitole 3.1.2 nevyskytuje. Místo něj se o překlad aplikací do strojového kódu stará tzv. *Dalvik VM* popsaný v kapitole 1. Ten je kromě specifického způsobu překladu typický také svou kompaktností, se kterou jde ruku v ruce omezení možností ve srovnání s běžným *JRE*.

Při kompilaci aplikace pro systém Android dochází ve finále ke kompilaci a zabalení tříd do souboru typu *dex*. V glosáři stránek *Android Developers* je *dex* popsán následovně:

> Soubor zkompilované Android aplikace.
>
> Programy Androidu jsou kompilovány do *dex (Dalvik Executable)* souborů, které jsou dále zazipovány do jednoho *apk* souboru v zařízení. *Dex* soubory mohou být vytvořeny automatickým překladem zkompilovaných aplikací napsaných v programovacím jazyce Java. (14)

Dalvik Executable je tedy další krok mezi kompilací třídy do *class* a jejím spuštěním. O jeho vytvoření se stará kompilační nástroj vývojového prostředí aplikace, proto se úloha načtení externí třídy komplikuje. Použití Groovy tuto skutečnost nijak neovlivňuje.

Při použití běžného *class loaderu* nebo *Groovy class loaderu* vždy dojde k selhání, přičemž konkrétní bod tohoto selhání se liší podle verze systému a u Groovy také podle knihoven k tomu použitých. V praxi byla například vyzkoušena varianta standardních Groovy knihoven pro Javu, které bez problému fungovaly v příkladu, který ilustruje Ukázka kódu 1. Přičemž při kompilaci aplikace s cílovou platformou API19, neboli Android 4.4 KitKat, došlo k selhání při snaze aplikace instanciovat načtenou třídu, protože zkompilovaný *class* kód byl pro *Dalvik* nečitelný. Při pozdějším použití *Groovy class loaderu* z knihovny *Grooid* (viz kapitola 5.1) a cílovém API21 proces kompilace třídy na začátku selhal s chybovým hlášením „General error during class generation: can't load this type of class file“, neboli „Obecná chyba při generaci třídy: nelze načíst tento typ souboru třídy.“

Správný postup je využití třídy *DexClassLoader*, což je *class loader* který načítá třídy z *jar* nebo *apk* (archiv Java aplikace a archiv Android aplikace) souborů obsahujících záznam *classes.dex*. (15) Řešení problému je tedy v odpovědi na otázku jak vytvořit takovýto *jar* nebo *apk* – lze to buď dynamicky, viz kapitola 5.1.2, nebo s použitím vývojového prostředí, viz kapitola 6.2.

4.3 Dynamické načítání externích aktivit

Byla vytvořena aplikace pro otestování možností dynamického načítání aktivit. Na základě pokusů s touto aplikací byly odvozeny následující závěry.

Při použití instance třídy *DexClassLoader*, které je předán odkaz na soubor *apk* nebo *jar* s aktivitou k načtení, a následném volání metody *loadClass*, které je předán název třídy aktivity, dojde k chybovému hlášení:

„*android.content.ActivityNotFoundException: Unable to find explicit activity class {*cesta ke třídě*}; have you declared this activity in your AndroidManifest.xml?*" neboli „Výjimka – aktivita nenalezena: nebylo možné nalézt explicitní třídu aktivity; deklarovali jste tuto aktivitu v AndroidManifest.xml?"

Hlášení napovídá, že řešením je deklarace aktivity v *AndroidManifest.xml* – pro úspěšnou kompilaci aplikace není potřeba, aby deklarovaná třída aktivity v aplikaci existovala. Pokusem bylo zjištěno, že je také potřebná existence jejího balíčku v adresářové sktruktuře aplikace. Pokud je tedy v *AndroidManifest.xml* deklarována aktivita s identickým názvem, jako ta, která má být načtena, je tento problém odstraněn. Aktivitu ale stále není možné načíst, kvůli následující chybě:

„java.lang.RuntimeException: Unable to instantiate activity ComponentInfo {cz.noharmdan.apkmodular/cz.noharmdan.simpleclockapp.MainActivity}: java.lang.ClassNotFoundException: Didn't find class "cz.noharmdan.simpleclockapp.MainActivity" on path: DexPathList[[zip file "/data/app/cz.noharmdan.apkmodular-2.apk"], nativeLibraryDirectories = [/data/app-lib/cz.noharmdan.apkmodular-2, /system/lib]]"

Aktivita, kterou se aplikace snaží načíst, se nachází v *apk* souboru, uloženém uživatelem v paměti zařízení, v balíčku *cz.noharmdan.simpleclockapp.MainActivity*, je ale hledána pod cestou */data/app/cz.noharmdan.apkmodular-2.apk*, což je cesta k archivu spuštěné aplikace v uživateli nepřístupné části paměti zařízení. Tato chyba je vyvolána v instanci třídy *BaseDexClassLoader*, která je předkem každé instance *DexClassLoader*. Tato třída při dynamickém načítání nejdříve zkontroluje, jestli třída stejného typu jako načítaná již existuje, a pokud ne, vyhledá ji v balíčku, na který si vytváří po svém vytvoření odkaz. (16) Dynamická třída není načtena okamžitě – je na ni v patřičném *class loaderu* vytvořen odkaz, ale k samotnému získání kódu dojde až ve chvíli, kdy je třídu nutné instanciovat. V tomto bodě dochází k problému. Volání načtení třídy je přesměrováno na rodičovskou třídu daného *class loaderu*, což je v tomto případě instance výchozí třídy *ClassLoader* patřící hlavní aktivitě testovací aplikace, a tedy jejím balíčkům. Protože je aktivita deklarována v *AndroidManifest.xml* a její balíček v aplikaci existuje, je automaticky vyhledávána v něm a nikoliv v balíčku v archivu zadaného *apk* souboru.

Řešením by bylo změnit cílovou cestu pro hledání aplikace v tomto výchozím *class loaderu*, tj. změna *DexPathList* zmíněného v chybovém hlášení, popř. změna samotné instance třídy *ClassLoader* přiřazené k aplikaci. Ani jedna z těchto variant není přímým přístupem možná. Možné řešení popisuje kapitola 5.2.

5 Existující řešení

5.1 Spouštění dynamických Groovy skriptů

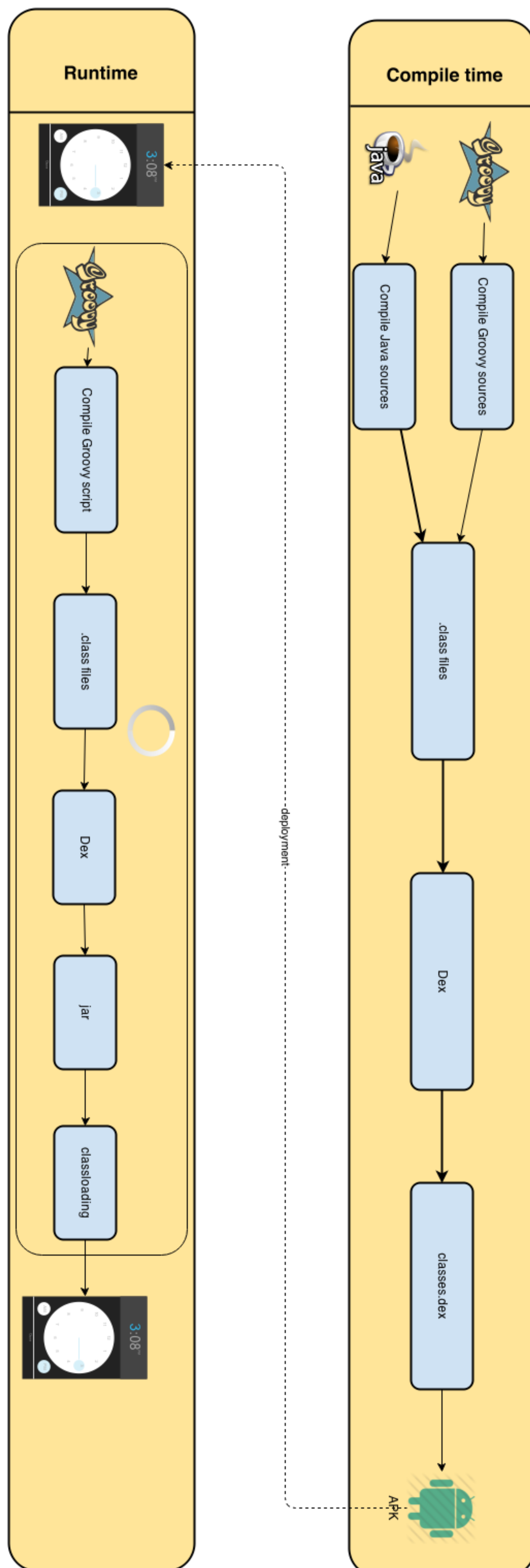
5.1.1 Knihovny pro podporu Groovy

Řešením problematiky dynamického načítání tříd skrze třídu *dexClassLoader* s využitím Groovy se zabývá francouzský vývojář Cédric Champeau, který rozšířil Java knihovny pro podporu Groovy o knihovny speciálně navržené pro systém Android.

Prvotním cílem bylo vytvoření knihoven, které by umožnily kompilaci Groovy tříd současně s Java třídami, nebo i bez nich, při kompilaci aplikace ve vývojovém prostředí. Tyto knihovny jsou k dispozici jako „*Gradle Plugin for Groovy on Android*" (17). Dá se s nimi nahradit kompletní zdrojový kód Java aplikace kódem ve formátu jazyka Groovy.

Pro efektivní využití jazyka Groovy je třeba i možnost spouštět jeho skripty dynamicky, za běhu aplikace. Problém spočívá ve faktu, že každá část kódu (a tedy i Groovy skript) musí být součástí třídy spustitelné pomocí *Dalvik VM*. Nalezeným řešením je úprava *Groovy class loaderu* takovým způsobem, aby bytekód přeložený ze skriptu (potažmo *groovy* souboru) nepředával ke spuštění *JVM*, ale přeložil ho ještě jednou do patřičného formátu pro *Dalvik*, zabalil do *jar* souboru a z něj kód načetl jako třídu, kterou již je možné zinstanciovat. Tento proces vhodně ilustruje schéma z článku na blogu Cédrica Champeau, viz Obrázek 1.

O správnou interpretaci dynamicky zadaného skriptu se stará knihovna *Grooid* od Cédrica Champeau, která se dá použít samostatně bez Groovy pluginu pro Android – aplikace pak musí být psána v Javě, ale je schopna dynamicky spouštět Groovy skripty.



Obrázek 1 (35) – Schéma kompilace a spuštění Java a Groovy tříd

5.1.2 Grooidshell Example

Pro demonstraci využití své knihovny pro podporu Groovy v Android vytvořil Cédric Chamepeau ukázkovou aplikaci *grooidshell-example* (18), která kromě kompilace tříd ve zdrojovém kódu z jazyka Groovy předvádí dynamické spouštění krátkých Groovy skriptů. Kód aplikace je dostupný pod licencí Apache verze 2.0. (19)

Nejpodstatnějšími částmi aplikace jsou dvě třídy, které umožňují dynamické spouštění skriptů. První z nich je *GrooidClassLoader*. Jedná se o potomka třídy *GroovyClassLoader* z knihoven pro podporu Groovy v Javě, který zajišťuje přerušení instanciace načtené třídy, aby mohla být předána *Dalvik VM*. Druhou je pak *GrooidShell*, třída, která vytváří patřičné prostředí a řídí proces dynamického spouštění skriptu.

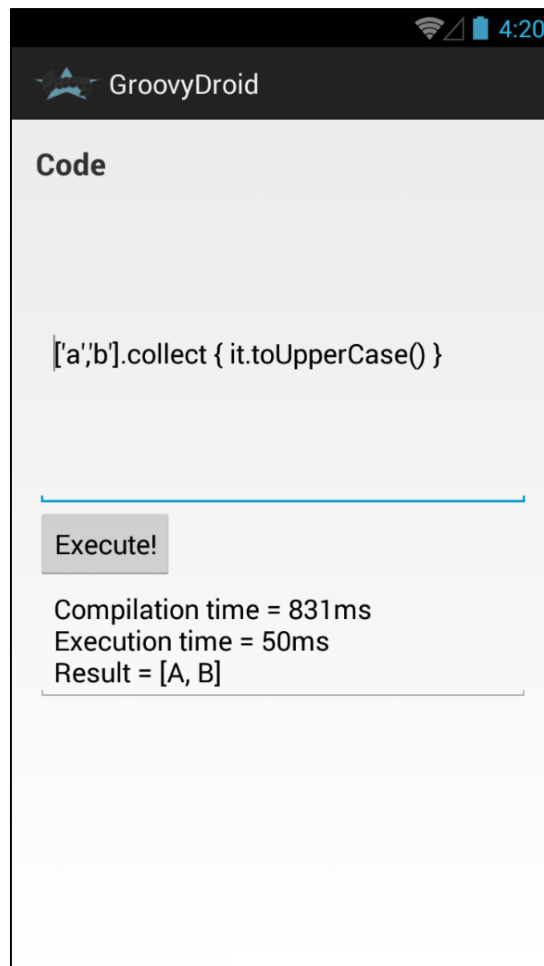
Tento proces se skládá z několika kroků:

1. Po zadání skriptu uživatelem se připraví prázdný soubor formátu *dex* s vhodnými parametry (specifikace kompatibilní verze Android API).
2. Vytvoří se speciální post-procesor bytekódu, jehož účelem je přijmout bytekód formou jednorozměrného pole bytů po přeložení do *class* formátu, přeložit ho do formátu pro *dex* soubor a do něj ho následně uložit.
3. Tento post-procesor je předán jako výchozí post-procesor pro novou instanci *GrooidClassLoader*, která následně zpracuje text požadovaného skriptu, zadaného uživatelem.
4. Takto naplněný *dex* soubor je načten do bytekódu ve formátu srozumitelném pro *Dalvik* a předán k dalšímu zpracování.
5. Kódem *dex* souboru je naplněn *jar* soubor, který je pak načten novou instancí třídy *DexClassLoader* – do paměti aplikace je uložena instance obecné třídy obsahující spustitelný kód skriptu.
6. Načtená třída je přetypována na třídu Groovy skriptu a je spuštěn její kód.
7. Výsledek kódu je navrácen uživateli v čitelné podobě spolu s informací o délce trvání procesu.

Problém tohoto přístupu je v jeho nedostatečné efektivitě z hlediska výkonu. Jednotlivé kroky se mohou jevit jako poměrně jednoduché, ale vyžadují množství procesního výkonu zařízení a od něj se odvíjející doby zpracování.

Krátký skript byl, jak ilustruje Obrázek 2, zkompilován a spuštěn dohromady za necelou jednu sekundu. Použité zařízení byla virtuální kopie telefonu Samsung Galaxy S3, se systémem Android 4.3, 1024MB operační paměti a přímým přístupem k čtyřjádrovému procesoru Intel Core i7-4710HQ s frekvencí 2,5GHz. Při kompilaci rozsáhlých tříd nebo jejich sdružení, které mohou mít od desítek do stovek řádků kódu, může požadovaná doba stoupat. Teoreticky se pak může aplikace stát neefektivní, až v praxi nepoužitelnou.

S využitím kódu z aplikace *grooidshell-example* byla vytvořena aplikace, která umožňuje načítání modulů ve formě celých externích Groovy tříd.



Obrázek 2 – Snímek obrazovky aplikace grooidshell-example

5.2 Načítání externích aktivit s využitím reflexe

Řešení problému popsaného v kapitole 4.3 je možné skrze Java reflexi. Reflexe umožňuje prohlížet třídy, rozhraní, pole (proměnné) a metody za běhu aplikace, bez toho, aby byly známy názvy těchto tříd, metod atp. během kompilace. Je skrze ni také možné instanciovat nové objekty, volat metody a získávat nebo nastavovat hodnoty proměnných. (20)

Příkladem využití reflexe pro načítání aktivit z externích *apk* souborů je Ukázka kódu 4, jejímž autorem je vývojář Marshall Culpepper z USA. (21) Cílem této metody je nahrazení výchozí instance třídy *ClassLoader* aplikace novou instancí třídy *DexClassLoader*, která vždy vyhledává třídy v zadané lokaci. Metoda *setAPKClassLoader* na řádku 19-21 pomocí reflexe získá instanci hlavního vlákna aplikace, z něj pak instanci třídy reprezentující mapování balíčků aplikace (řádek 23-24). Na řádku 26-28 z tohoto mapování získá odkaz na instanci současného balíčku, ve kterém se nachází spuštěná aktivita. Tento objekt s sebou nese informaci o výchozí instanci *ClassLoader* v poli *mClassLoader*. Metoda dále na řádku 30-32 získá odkaz na toto pole, zpřístupní ho pro úpravy a přepíše na odkaz na *DexClassLoader*, kterému byla v parametru předána cesta k *apk* souboru s aktivitou.

Na této instanci *DexClassLoader* je na řádku 7 zavolána metoda *loadClass*, která nalezne požadovanou třídu a uloží si o ni informace nutné k načtení. Ve chvíli instanciace pak není volán výchozí *class loader*, který by třídu hledal v balíčku aplikace, ale opět instance *DexClassLoader*, která třídu vyhledá v zadaném souboru a finálně načte.

Takto získaná instance aktivity je na řádku 8-9 pomocí *intent* bez problému spuštěna.

```
01 public class ClassLoaderActivity extends Activity   {
02   public void onCreate(Bundle savedInstanceState) {
03       ClassLoader dexLoader = new DexClassLoader("/path/to/file.jar",
             getCacheDir().getAbsolutePath(), null, getClassLoader());
04       setAPKClassLoader(dexLoader);
05
06       try {
07           Class<?> activityClass = dexLoader.loadClass("com.company.MyActivity");
08           Intent intent = new Intent(this, activityClass);
09           startActivity(intent);
10       } catch (ClassNotFoundException e) {
11           e.printStackTrace();
12       }
13       finish();
14   }
15
16   private void setAPKClassLoader(ClassLoader classLoader) {
17       try {
18           // získá hlavní vlákno aktivity
19           Field mMainThread = getField(Activity.class, "mMainThread");
20           Object mainThread = mMainThread.get(this);
21           Class threadClass = mainThread.getClass();
22           // získá mapu balíčků
23           Field mPackages = getField(threadClass, "mPackages");
24           HashMap<String,?> map = (HashMap<String,?>) mPackages.get(mainThread);
25           // získá balíček aktivity
26           WeakReference<?> ref = (WeakReference<?>) map.get(getPackageName());
27           Object apk = ref.get();
28           Class apkClass = apk.getClass();
29           // získá odkaz na class loader
30           Field mClassLoader = getField(apkClass, "mClassLoader");
31           // nastaví class loader
32           mClassLoader.set(apk, classLoader);
33       } catch (IllegalArgumentException e) {
34           e.printStackTrace();
35       } catch (IllegalAccessException e) {
36           e.printStackTrace();
37       }
38   }
39
40   private Field getField(Class<?> cls, String name) {
41      // získá všechny pole třídy cls
42       for (Field field: cls.getDeclaredFields()) {
43           // zpřístupní pole pro úpravy
44           if (!field.isAccessible()) field.setAccessible(true);
45           // pokud je nalezeno hledané pole, vrátí na něj odkaz
46           if (field.getName().equals(name)) return field;
47       }
48       return null;
49   }
50 }
```

Ukázka kódu 4 – Načítání aktivity reflexí

6 Návrh řešení

6.1 Aplikace využívající Groovy

Cílem vytvořené aplikace bylo upravit třídy *GrooidClassLoader* a *GrooidShell* z aplikace *grooidshell-example* tak, aby namísto spouštění skriptu zadaného uživatelem obstarávaly načítání a instanciaci externích Groovy tříd.

6.1.1 Struktura modulu

Podobně jako ve variantě pro osobní počítače, viz kapitola 4.1, je pro efektivní využití modulu nutné, aby dědil vlastnosti nějaké nativní Android třídy, v tomto případě třídy reprezentující nějakou část uživatelského rozhraní aplikace.

Jako předek třídy modulu byla zvolena třída *Fragment*, jejíž potomci nemají být v manifestu aplikace nijak popsáni, tudíž nemůže docházet ke konfliktu při jejich externím načítání. Fragment je část uživatelského rozhraní nebo chování aplikace, která může být vložena do aktivity. Třída *Fragment* může být použita mnoha způsoby k dosažení širokého spektra výsledků. V jádře reprezentuje specifickou operaci nebo uživatelské rozhraní, které běží v obalující aktivitě. Fragment je úzce spjat s aktivitou, do které je vložen, a nemůže bez ní být použit. Ačkoliv fragment definuje svůj vlastní životní cyklus, tento cyklus je závislý na jeho aktivitě: pokud je aktivita zastavená, žádné v ní vnořené fragmenty nemohou být spuštěny a pokud je aktivita zrušena, její fragmenty jsou také zrušeny. (22)

Při dědění po třídě *Fragment* je nutné řešit několik problémů.

Fragment vygenerovaný v prostředí *Android Studio* obsahuje rozhraní *OnFragmentInteractionListener*. Toto rozhraní má za úkol propojit aktivity a fragment skrze zachytávání vstupu uživatele. V standardní situaci je nutné, aby aktivita, do které je fragment vložen, implementovala toto rozhraní. Tato aktivita má pak možnost přistupovat ke svému obsahu v závislosti na vstupu uživatele do fragmentu. Například dotyk nějakého prvku fragmentu vyvolá metodu definovanou v rozhraní, která je tedy v aktivitě implementována, takže na ní aktivita dle implementace zareaguje. Výhoda tohoto principu spočívá ve zjednodušení správy fragmentů a předávání informací z fragmentu do aktivity nebo do dalších fragmentů. Problémem je skutečnost, že aplikace při své kompilaci nemůže „znát“ obsah případných modulů a tudíž ani jejich rozhraní, které by musela implementovat.

Řešením je odstranění rozhraní a všeho s ním souvisejícího z fragmentu. Pro zachytávání vstupu uživatele do fragmentu je vhodné po instanciaci, tj. v přetěžované metodě *onStart*, volat metodu *getView* a do navráceného kořenového prvku uživatelského rozhraní nastavit *listener* naplněný patřičným chováním pro zachytávání vstupu. Tímto je fragment odříznut od volání metod aktivity nebo jiných fragmentů, kromě me-

tod definovaných v původním předku aktivity, tj. samotné třídě *Acitivity*, které nezbytně každá aktivita dědí – každý fragment může přistupovat ke své obklopující aktivitě metodou *getActivity*. Naopak výhodou tohoto řešení je oddělení modulu od ostatních prvků aplikace z hlediska bezpečnosti.

Další problém se týká omezení vyplývajících ze struktury Android aplikace, která je definována *xml* soubory. Vzhled každého prvku uživatelského rozhraní (tj. aktivity, fragmentu) je popsán vlastním *xml* souborem. Pro programový přístup k prvkům popsaným v tomto souboru je používána vývojovým prostředím automaticky generovaná třída *R*. Každá aplikace má svou vlastní třídu *R*, skrz jejíž atributy přistupuje ke svým *xml* souborům. (23) Tato třída je pokaždé svým obsahem jiná. Není tedy možné ji v modulu importovat a použít. Třída R navrací podle názvu elementu v *xml* jeho identifikátor ve formě celého kladného čísla. Z tohoto faktu vyplývá výhoda bezpečnosti, protože modul nemůže přistupovat k *xml* souborům aplikace podle jejich názvů. Zároveň ale vzniká problém s definicí vzhledu uživatelského rozhraní modulu. To musí být v přetěžované metodě *onCreateView* vytvořeno programově. Výchozí volání *inflater.inflate* nelze použít, protože v argumentu dostává odkaz na atributy třídy *R*, ke které nemá modul přístup. To také znamená, že případný *xml* soubor s vzhledem modulu není možné v *R* najít, takže aplikace nemůže přímo přistupovat k prvkům vzhledu modulu. Přístup by byl možný, pokud by byla modulární aplikace distribuována se seznamem vhodných elementů a jejich číselnými identifikátory. Volané metody pro přístup k *xml* souborům v modulu by pak v argumentu měly přímo tyto identifikátory, nikoliv atributy třídy *R*.

Při dodržení zásad vyplývajících z těchto skutečností je pak možné sestavit třídu fragmentu vhodnou k použití jako externí modul. Příkladem takové třídy, psané v jazyce Groovy, která má pouze funkci zobrazování analogových hodin, je třída *ClockFragment*, jejíž obsah ilustruje Ukázka kódu 5.

```
import android.os.Bundle
import android.support.v4.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.AnalogClock
import android.widget.LinearLayout

class ClockFragment extends Fragment {

	ClockFragment() {
		// Požadovaný prázdný konstruktor
	}

	@Override
	View onCreateView(LayoutInflater inflater, ViewGroup container,
			Bundle savedInstanceState) {
		LinearLayout ll = new LinearLayout(getActivity())
		ll.setOrientation(LinearLayout.VERTICAL)
		ll.addView(new AnalogClock(getActivity()))

		return ll
	}
}
```

Ukázka kódu 5 – Groovy třída ClockFragment

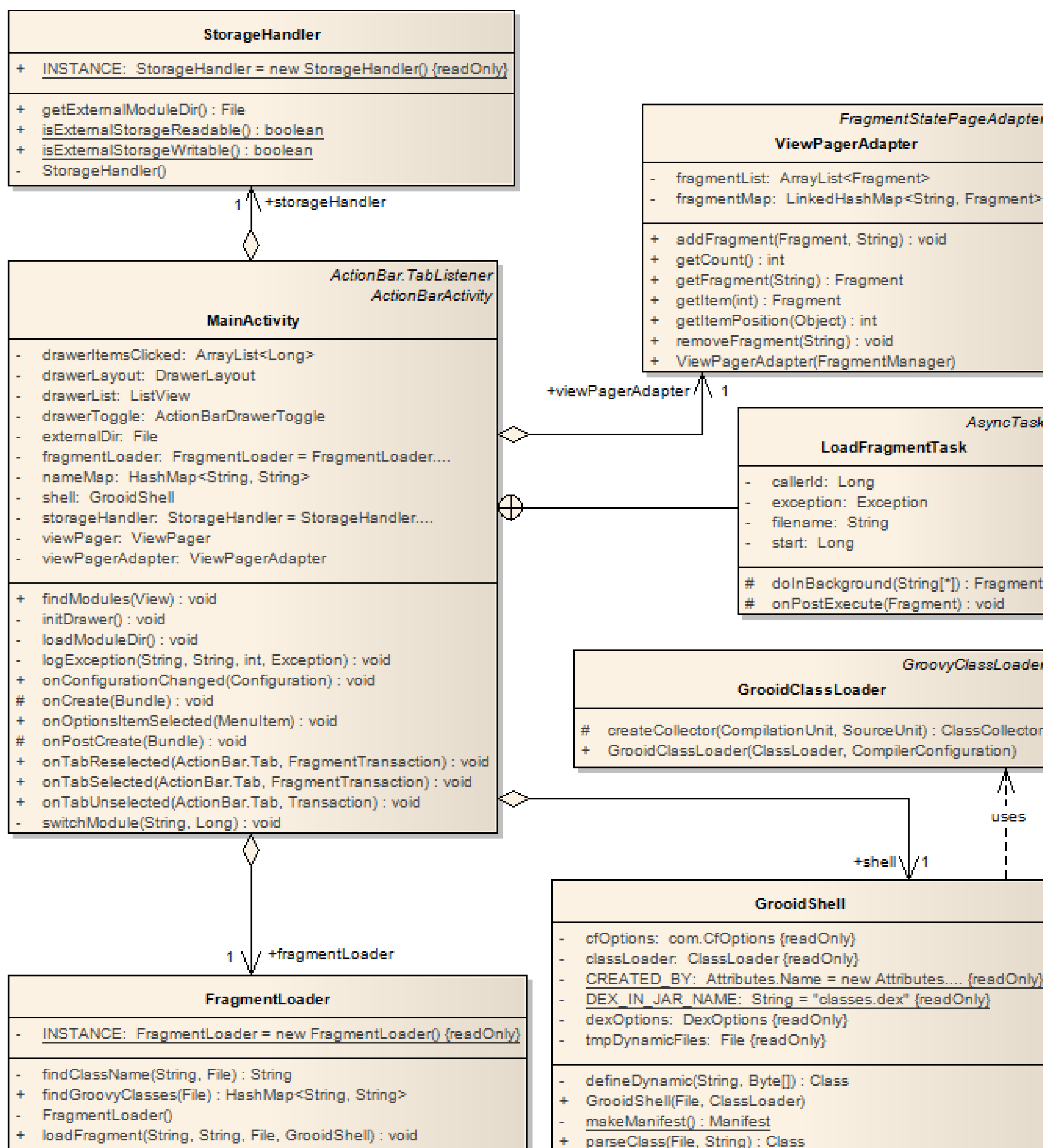
6.1.2 Struktura aplikace

Návrh struktury tříd aplikace popisuje Obrázek 3.

Aplikace je psána v Javě, nikoliv v Groovy, a proto používá jen knihovnu *Grooid* pro dynamickou kompilaci jazyka Groovy. Přesto je kvůli rozsahu této knihovny potřeba řešit problém s omezením aplikace na 65 536 metod, který je na *Android Developers* popsán následovně:

Soubory aplikace pro Android (*APK*) obsahují spustitelné soubory s bytekódem ve formě souborů *Dalvik Executable* (*DEX*), které obsahují zkompilovaný kód použitý pro běh aplikace. Specifikace *Dalvik Executable* limitují počet metod, které mohou být zahrnuty uvnitř jednoho *DEX* souboru na 65 536, včetně metod *Android Framework*, metod knihoven a metod v kódu aplikace. Překročení tohoto limitu vyžaduje konfiguraci procesu sestavení spustitelné aplikace tak, aby vygeneroval více než jeden *DEX* soubor – známé jako *multidex* konfigurace. (24)

V článku jsou dále popsány způsoby řešení. V aplikaci *GyModular* je použita knihovna *com.android.support:multidex:1.0.0* a spustitelná třída aplikace je v *AndroidManifest.xml* definována jako *android.support.multidex.MultiDexApplication* z této knihovny.



Obrázek 3 – Struktura aplikace GyModular

6.2 Aplikace využívající Javu

Pro demonstraci obdobného řešení, jako je popsáno v kapitole 6.1, ale bez použití jazyka Groovy, byla vytvořena aplikace podle následujícího návrhu.

6.2.1 Struktura modulu

Jako třída modulu byl zvolen *Fragment*. Obecná pravidla pro vnitřní strukturu funkčního modulu jsou shodná jako v kapitole 6.1.1, s několika rozdíly.

Modul musí být v tomto případě součástí archivu aplikace pro Android, tj. *apk,* nebo pro ostatní systémy využívající Javu, tj. *jar*. Pro účely práce byla zvolena varianta archivu *apk.* Je možné ho takto vytvořit jako součást libovolné aplikace v libovolném vývojovém prostředí, podporujícím vývoj aplikací pro Android. Pro jednoduchost načítání musí třída fragmentu být pojmenována „*ModuleFragment.java*“ a musí být v kořenovém adresáři aplikace.

Fragment může přistupovat k *xml* souborům aplikace, do které je zaváděn, skrze třídu *R*. Protože je fragment kompilován s nějakou aplikací, dochází při překladu jeho kódu k převedení těchto odkazů na číselné identifikátory. Pokud by se pak takový identifikátor shodoval s identifikátorem nějakého prvku aplikace, do které je fragment zaváděn, může k tomuto prvku být ve fragmentu přistupováno. Tato situace není pro praktické využití příliš pravděpodobná – za předpokladu, že by součástí modulární aplikace nebyl seznam číselných identifikátorů prvků, se kterými mohou moduly pracovat.

6.2.2 Struktura aplikace

Aplikace v designu vychází z modelu popsaného v kapitole 6.1.2. V aplikaci nejsou zahrnuty knihovny *Grooid* a třídy *GrooidShell* a *GrooidClassLoader*. (viz Obrázek 3) Rozdíl je dále v obsahu třídy *FragmentLoader*. Metoda *findGroovyClasses* je nahrazena metodou *findApkClasses* a metodě *loadFragment* není v argumentu předávána instance třídy *GrooidShell*, ale *ClassLoader* hlavní aktivity a složka pro dočasné soubory. Konkrétní implementace třídy *FragmentLoader* je popsána v části FragmentLoader kapitoly 7.2. Spustitelná třída aplikace nemusí být v *AndroidManifest.xml* definována jako *android.support.multidex.MultiDexApplication*.

7 Implementace

Aplikace byly vyvinuty v prostředí *Android Studio. Android Studio* je oficiální vývojové prostředí pro vývoj aplikací pro Android, založené na *IntelliJ IDEA*. (25) *IntelliJ IDEA* je vývojové prostředí pro firemní, mobilní a webový vývoj v jazycích Java, Scala a Groovy. (26) Aplikace jsou kompilovány pro cílové Android API 21 (systém Android 5.0) se zpětnou kompatibilitou k verzi API 14 (Android 4.0).

7.1 Aplikace GyModular

Aplikace umožňuje načítat moduly ve formě tříd fragmentů v jazyce Groovy, viz kapitola 6.1.

7.1.1 Třídy logiky načítání modulů

Klíčovými prvky aplikace jsou třídy *GrooidClassLoader*, popsaný v kapitole 5.1.2 a upravená třída *GrooidShell*. Metody třídy *GrooidShell* jsou volány třídou *FragmentLoader.*

FragmentLoader

Během chodu aplikace je spuštěna vždy právě jedna instance této třídy. Nabízí tři metody, *findGroovyClasses*, *findClassName* a *loadFragment*.

Metoda *findGroovyClasses:*

1. Metoda prohledá v argumentu zadanou instanci třídy *file* jako adresář.
2. Pokud tento adresář v paměti zařízení neexistuje, nebo se nejedná o adresář, ale o soubor, dojde k výjimce a vyhledávání se ukončí.
3. Pokud jsou tyto podmínky splněny, metoda v adresáři vyhledá všechny soubory s příponou *groovy* a uloží seznam jejich názvů.
4. Tento seznam následně metoda projde a v každém souboru vyhledá metodou *findClassName* název třídy.
5. Ke každému názvu souboru přiřadí patřičný název třídy a tyto páry uloží do konstrukce *HashMap,* která je návratovou hodnotou metody.

Metoda *findClassName:*

1. Vyhledávání názvů tříd probíhá procházením kódu v souboru po řádcích, dokud není nalezen řádek obsahující řetězec *„class “*, tedy výraz *class* a znak mezery.
2. V takovém případě je vyjmut řetězec následující za tímto řetězcem až do dalšího výskytu mezery.
3. Tento řetězec je považován za název třídy. Pokud není nalezen řádek splňující podmínku, navrácený řetězec obsahuje prázdnou hodnotu a příslušný název souboru je ze seznamu názvů vymazán.

Tímto jsou rovnou eliminovány soubory, které neobsahují deklaraci třídy, a zároveň je umožněno načítání tříd s duplicitním názvem, jsou-li uloženy v souborech s rozdílným názvem.

Metoda *loadFragment:*

1. Metoda obdrží v argumentu název souboru modulu, název třídy modulu, odkaz na adresář s moduly a instanci třídy *GrooidShell*.
2. Stejně jako *findGroovyClasses* zkontroluje validitu zadaného adresáře a případně vyhodí výjimku.
3. Pokud je adresář validní, metoda zavolá metodu *parseClass* instance třídy *GrooidShell*, jejíž návratovou hodnotou je instance obecné třídy *Class*.
4. Pokud v průběhu volání metody *parseClass* dojde k chybě, která zapříčiní prázdnou návratovou hodnotu, metoda *loadFragment* vyhodí výjimku a přeruší svůj běh.
5. Následně dojde ke kontrole, jestli je možné získanou instanci C*lass* přetypovat na instanci třídy *Fragment*.
6. Pokud ne, je vyhozena výjimka a metoda přeruší svůj běh, pokud ano, dojde k přetypování na třídu *Fragment* a zavolání metody *newInstance* na této třídě.
7. Tím je instance modulu načtena jako fragment a připravena k přidání do uživatelského rozhraní aplikace.

GrooidShell

Konstruktoru třídy *GrooidShell* je v parametru předán adresář pro ukládání dočasných souborů a *ClassLoader* náležící hlavní aktivitě aplikace. V konstruktoru dochází k nastavení formátu *dex* a *class* souborů pro načítané moduly – zápis proměnných *dexOptions* a *cfOptions*. Proměnné *dexOptions* je nastaven parametr *targetApiLevel* na *API_NO_EXTENDED_OPCODES.* To zajišťuje širší kompatibilitu dynamicky načtených tříd se staršími verzemi Dalvik VM, přítomnými v systémech Android verze 3.0 a starší, které neumějí zpracovávat rozšířené programové instrukce, tzv. *extended opcodes*. (27) Proměnné *cfOptions* jsou nastaveny parametry zpracování *class* souboru po jeho kompilaci z *groovy* souboru. Příznaky pro uložení čísel řádků z původního souboru, pro uložení lokálních informací o proměnných a pro striktní kontrolu názvu třídy proti názvu jejího *class* souboru jsou nastaveny pozitivně, příznaky pro optimalizaci metod podle předepsaných souborů a pro ukládání statistik o kompilaci jsou nastaveny negativně.

Metoda *parseClass,* viz Ukázka kódu 6**,** zajišťuje načtení třídy obdobně jako v Grooidshell Example. Metoda nahrazuje metodu *evaluate(String scriptText)* z *grooidshell-example*, v jejímž argumentu je předáván řetězec s textem vyhodnocovaného Groovy skriptu.

1. Řádek 8 – metodě *parseClass* je předáván soubor s modulem a název třídy modulu.
2. Řádek 10 – metoda vytvoří *dex* soubor podle parametrů *dexOptions* nastavených v konstruktoru.

3. Řádek 13-29 – metoda připraví post-procesor bytekódu pro předání instanci třídy *GrooidClassLoader*, který obstarává uložení přeloženého kódu třídy do vytvořeného *dex* souboru.
   K překladu dochází voláním metody *translate* na instanci třídy *CfTranslator*, které je v parametru předáno pole bytů pro uložení přeložené třídy, název jejího *class* souboru a proměnné *cfOptions* a *dexOptions*. Výchozí post-procesor děděný po třídě *GroovyClassLoader* přeložený kód neukládá do *dex* souboru, ale vrací v proměnné *bytes*. Takto navrácený bytekód v proměnné *bytes* je výchozí třídou *GroovyClassLoader* dále zpracováván, což není žádoucí, protože pak nedochází k jeho úpravě pro Dalvik VM a není možné ho na systému Android spustit. Třída *GrooidClassLoader* tento proces přerušuje.
4. Řádek 32 – metoda vytvoří instanci *GrooidClassLoader* jejímuž konstruktoru je předán připravený post-procesor a *ClassLoader* získaný z aktivity.
5. Řádek 36 – metoda zavolá metodu *parseClass* instance *GrooidClassLoader*, která s využitím post-procesoru uloží kód třídy do *dex* souboru. V tomto kroku může dojít k chybě, pokud byl soubor třídy mezitím přesunut nebo odstraněn, popř. pokud jeho obsah neodpovídá obecné struktuře Groovy třídy. V případě chyby je načítání přerušeno, ale aplikace pokračuje v běhu.
6. Řádek 45 – metoda na *dex* souboru zavolá metodu *toDex*, která vrátí sekvenci bytů obsahujících kód třídy ve formátu pro *Dalvik VM*.
7. Řádek 50 – návratová hodnota metody *parseClass* je instance třídy *class*, která je získána voláním metody *defineDynamic*, které je v parametru předán název třídy modulu a bytekód získaný metodou *toDex*.

Metoda defineDynamic, viz Ukázka kódu 7, využívá klíčovou třídu DexClassLoader:

1. Řádek 8 – metoda vytvoří ve složce pro dočasné soubory nový *jar* soubor s náhodně vygenerovaným, unikátním názvem.
2. Řádek 10-11 – následně vytvoří datové proudy pro zápis do vytvořeného *jar* souboru. Proudu *JarOutputStream* je předán *xml* manifest vytvořený metodou *makeManifest*, který nese informace o verzi a názvu *classes.dex* souboru, který bude *jar* soubor obsahovat.
3. Řádek 13-16 – v *jar* souboru metoda připraví prostor pro položku *classes.dex* patřičné délky podle předaného bytekódu a tu jím naplní.
4. Řádek 17-21 – takto naplněný *jar* soubor metoda zapíše do složky pro dočasné soubory a uzavře datové proudy.
5. Řádek 22 – metoda vytvoří instanci třídy *DexClassLoader*, které předá odkaz na vytvořený *jar* soubor, na složku s dočasnými soubory a na *ClassLoader* aktivity aplikace.
6. Řádek 23 - metoda navrátí instanci třídy *class* získanou zavoláním metody *loadClass* na instanci *DexClassLoader*, jejímž parametrem je název

třídy původně předaný metodě *parseClass*. V průběhu metody může dojít k chybě při zápisu nebo čtení v dočasném adresáři, pokud k němu aplikace například ztratí přístup, nebo k chybě, kdy se zadaný název třídy neshoduje s třídou v *classes.dex*. Ta během vývoje aplikace mohla nastat, pokud název souboru modulu (např. *Modul.groovy*) nebyl totožný s názvem třídy v něm obsaženém – slovem za výrazem *class* ve zdrojovém kódu. Použitým řešením je vyhledání názvu třídy uvnitř souboru modulu a používání názvu souboru jen jako identifikátoru. Tento způsob mírně zvýšil nároky na výkon a prodloužil čas vyhledávání modulů.

```
01 /**
02 * Načte externí Groovy třídu.
03 *
04 * @param filePath soubor třídy
05 * @param className název třídy
06 * @return načtená třída
07 */
08 public Class parseClass(File filePath, String className) {
09      // Dex pro uložení bytecode
10      final DexFile dexFile = new DexFile(dexOptions);
11      Log.d("parseClass", "className: " + className);
12      // Konfigurace pro class loader
13      CompilerConfiguration config = new CompilerConfiguration();
14      config.setBytecodePostprocessor(new BytecodeProcessor() {
15          /**
16           * Úprava metody, ukládá bytecode do dexFile
17           * @param s název třídy
18           * @param bytes bytecode třídy
19           * @return bytecode pro běžný JVM, použitelný výsledek je v dexFile
20           */
21          @Override
22          public byte[] processBytecode(String s, byte[] bytes) {
23              // Vytvoří objekt třídy pro uložení do dex souboru podle výše
24              // uvedených dex a cf možností
25              ClassDefItem classDefItem = CfTranslator
                    .translate(s + ".class", bytes, cfOptions, dexOptions);
26              dexFile.add(classDefItem);
27              return bytes;
28          }
29      });
30
31      // Class loader, který se postará o vykonání metody processBytecode
32      GrooidClassLoader gcl = new GrooidClassLoader(classLoader, config);
33
34      try {
35          // Parsování - volání processBytecode a uložení třídy do dexFile
36          gcl.parseClass(filePath);
37      } catch (Throwable e) {
38          Log.e("GrooidShell", "Dynamic loading failed!", e);
39          return null;
40      }
41
42      byte[] dalvikBytecode;
43      try {
44          // Převede bytecode v dexFile do formátu pro Dalvik
45          dalvikBytecode = dexFile.toDex(new OutputStreamWriter(
                    new ByteArrayOutputStream()), false);
46      } catch (IOException e) {
47          Log.e("GrooidShell", "Unable to convert to Dalvik", e);
48          return null;
49      }
50      return defineDynamic(className, dalvikBytecode);
51  }
```

Ukázka kódu 6 – Metoda parseClass třídy GrooidShell

```
01 /**
02  * Vytvoří z bytecode ve formátu pro Dalvik VM instanci třídy.
03  * @param className jméno třídy
04  * @param dalvikBytecode bytekód třídy
05  * @return instance třídy
06  */
07 private Class defineDynamic(String className, byte[] dalvikBytecode) {
08     File tmpDex = new File(tmpDynamicFiles,
             UUID.randomUUID().toString() + ".jar");
09     try {
10         FileOutputStream fos = new FileOutputStream(tmpDex);
11         JarOutputStream jar = new JarOutputStream(fos, makeManifest());
12         JarEntry classes = new JarEntry(DEX_IN_JAR_NAME);
13         classes.setSize(dalvikBytecode.length);
14         jar.putNextEntry(classes);
15         jar.write(dalvikBytecode);
16         jar.closeEntry();
17         jar.finish();
18         jar.flush();
19         fos.flush();
20         fos.close();
21         jar.close();
22         DexClassLoader loader = new DexClassLoader(tmpDex.getAbsolutePath(),
               tmpDynamicFiles.getAbsolutePath(), null, classLoader);
               // načítá class z dex
23         return loader.loadClass(className); // finální načtení třídy
24     } catch (Throwable e) {
25         Log.e("DynamicLoading", "Unable to load class", e);
26     } finally {
27         tmpDex.delete();
28     }
29     return null;
30 }
```

Ukázka kódu 7 – Metoda defineDynamic třídy GrooidShell

7.1.2 Třídy grafického uživatelské rozhraní

LoadFragmentTask

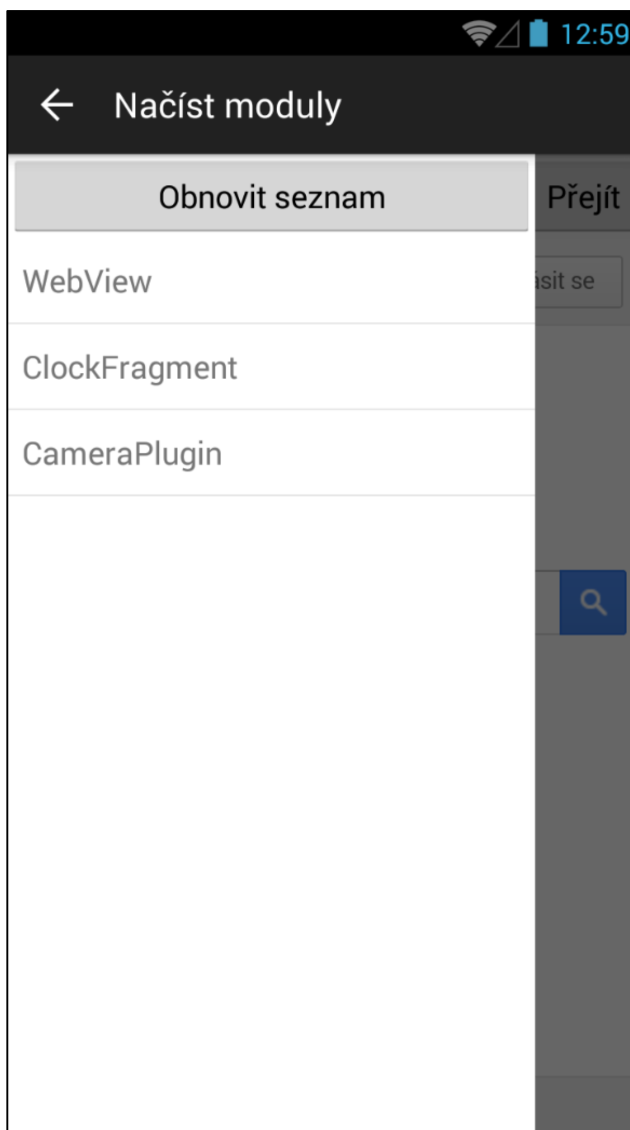
Třída *LoadFragmentTask* je potomkem třídy *AsyncTask*. Třída *AsyncTask* umožňuje vykonávat operace na pozadí a publikovat výsledky ve vlákně uživatelského rozhraní aplikace bez nutnosti manuální manipulace s vlákny. (28) Třída *LoadFragmentTask* obstarává volání metod tříd logiky načítání modulů na pozadí, čímž zajišťuje plynulý chod vlákna uživatelského rozhraní aplikace.

MainActivity

Aktivitou zajišťující grafické uživatelské rozhraní aplikace je třída *MainActivity*. Třída *MainActivity* je potomkem třídy *ActionBarActivity*, což třídě umožňuje používat *ActionBar*. *ActionBar* je primární panel nástrojů v aktivitě, který může zobrazovat titulek aplikace, prvky pro navigaci na úrovni aplikace a další interaktivní objekty. (29) V třídě *MainActivity* obstarává *ActionBar* přístup do postranního vysouvacího panelu, tzv. šuplíku (*drawer*), pro vyhledávání a výběr modulů (Obrázek 4). Tuto funkcionalitu obstarávají metody *onOptionsItemSelected, onPostCreate, onConfigurationChanged* a *initDrawer*. První tři z těchto metod jsou přetěžované metody třídy *ActionBarActivity* a obstarávají správné vykreslování šuplíku. Metoda *initDrawer* nastaví šuplíku *listener*, který obstarává volání metody *switchModule* pro načtení nebo odebrání modulu po klepnutí na jeho název. V posledním kroku metoda *initDrawer* nastaví chování pro *actionBar* ve chvíli otevření nebo zavření šuplíku – dochází k překreslení textu a tlačítka pro otevření/zavření.

Třída *MainActivity* implementuje rozhraní *ActionBar.TabListener*, které umožňuje zobrazovat každý načtený modul jako jednu stránku aplikace, mezi kterými je možný přesun přetažením do strany. Správu jednotlivých stránek obstarává třída *ViewPagerAdapter.*

Metoda *loadModuleDir* třídy *MainActivity* je volána po její instanciaci, v metodě *OnCreate*. Metoda do proměnné *externalDir* uloží cestu k složce s moduly – hodnotu navrácenou voláním metody *getExternalModuleDir* třídy *StorageHandler.* Pokud je hodnota *externalDir* prázdná, aplikace se ukončí, pokud ne, metoda vyhledá moduly voláním metody *findModules*. Metoda *findModules* zavolá metodu *findGroovyClasses* třídy *FragmentLoader*, získaný seznam párů názvů souborů modulů a názvů tříd uloží a zavolá vykreslení názvů souborů v šuplíku.



Obrázek 4 – Snímek obrazovky aplikace GyModular

Metodě *switchModule* je v argumentu předán název třídy modulu a číselný identifikátor položky v šuplíku, která byla zvolena:

1. Metoda zkontroluje, jestli předaný identifikátor není v seznamu označujícím aktivní položky. Pokud ano, metoda přeruší svůj běh. Pokud ne, metoda zkontroluje, jestli je modul již načten a pokud ano, odebere jej.
2. Pokud modul není načten, metoda na nové instanci vnitřní třídy *LoadFragmentTask* zavolá metodu *execute* kterou třída dědí po svém předkovi. Metodě *execute* je v argumentu předán název třídy a identifikátor položky v šuplíku.
3. Instance třídy *LoadFragmentTask* ve vlákně na pozadí načte modul voláním metody *loadFragment* třídy *FragmentLoader*. V tomto kroku odchytává chyby vyhazované během metody *loadFragment* a případně ukončí běh vlákna a zároveň upozorní uživatele voláním metody *logException*, která zobrazí uživateli zprávu popisující výjimku.
4. Po dokončení činnosti ve vlákně na pozadí odeber instance třídy *LoadFragmentTask* identifikátor vybrané položky v šuplíku ze seznamu aktivních, čímž jej zpřístupní k opětovnému výběru.
5. Pokud proběhlo načtení bez chyby, je načtený modul přidán do grafického uživatelského rozhraní voláním metody *addFragment* instance třídy *ViewPagerAdapter*.

StorageHandler

Metoda *getExternalModuleDir* ověří přístup do externího úložiště zařízení. Pokud má přístup k zápisu i čtení, najde složku s moduly a vrátí odkaz na ní, případně ji nejdříve vytvoří. Pokud nemá přístup, vrací prázdnou hodnotu. Uživatel je o stavu složky s moduly informován zprávou na obrazovce zařízení. K ověření přístupu do externího úložiště využívá třída metody *isExternalStorageReadable* a *isExternalStorageWritable*, převzaté z Android Developers. (30)

ViewPagerAdapter

Jedná se o potomka třídy *FragmentStatePagerAdapter*, který je implementací třídy *PagerAdapter,* používající fragmenty jako jednotlivé stránky. Tato třída zároveň obstarává ukládání a obnovování stavu fragmentů. Tato verze stránkování je vhodnější pro případy, kde je velké množství stran. Ve chvíli, kdy nejsou strany viditelné uživateli, je jejich fragment zničen a je zachován pouze jeho uložený stav. To umožňuje využívat malé množství paměti na úkor možné větší výpočetní náročnosti při listování mezi stránkami. (31) Tato varianta byla zvolena s ohledem na nepředpokládatelné množství zobrazovaných modulů. Metody *getItem, getCount* a *getItemPosition* jsou přetěžované metody předka, které pracují s indexovaným seznamem stránek uloženým v proměnné *fragmentList* a jsou automaticky volány aplikací při změně stavu stránek. Metody *getFragment, addFragment* a *removeFragment* rozšiřují adaptér o možnost správy stránek podle názvů tříd fragmentů modulů a jsou volány metodou *switchModule* třídy *MainActivity*. Metodou *getFragment* je kontrolována existence stránky fragmentu, metodou

removeFragment je stránka a instance fragmentu odstraněna z uživatelského rozhraní a seznamů *fragmentList* a *fragmentMap,* a metodou *addFragment* přidána. Zároveň vždy dochází k překreslení uživatelského rozhraní aplikace s aktuálně zobrazenými moduly.

7.1.3 Závěr

Z konečného řešení aplikace *GyModular* vyplývá několik možných chybových situací:

- Aplikace se může stát nepoužitelnou, pokud nemá přístup do úložiště zařízení. Pokud o tento přístup přijde během načítání modulu, nebo je v takovou chvíli soubor modulu odstraněn, dojde k výjimce, která toto načítání přeruší, ale nepřeruší běh samotné aplikace.
- Pokud vnitřní struktura modulu neodpovídá struktuře obecné třídy, resp. fragmentu, při jeho načítání dojde k výjimce, která toto načítání přeruší.
- Problém způsobují oprávnění aplikace deklarované v *AndroidManifest.xml*. Pokud se modul pokusí použít součást systému, ke které nemá aplikace předem deklarovaný přístup, dojde k chybě, která způsobí pád aplikace. Takovou akci může modul vykonat kdykoliv během svého běhu, takže aplikace může havarovat kdykoliv, bez pro uživatele zjevné příčiny. Aplikaci byla proto deklarována tato oprávnění: zápis a čtení externího úložiště zařízení, přístup k pozici zařízení, přístup ke kameře zařízení, přístup k internetu, přístup k bluetooth a přístup k NFC.
- Deklarace oprávnění v *AndroidManifest.xml* mohou způsobit problémy z hlediska bezpečnosti aplikace. Před načtením modulu není možné zkontrolovat jeho kód na potenciálně nebezpečný obsah. Takový obsah se tedy může spustit okamžitě po načtení modulu a uškodit uživateli.
- Nejzávažnější problém vznikne v situaci, kdy kód modulu svou strukturou umožňuje přetypování na fragment, ale obsahuje chyby. Tyto chyby se mohou projevit kdykoliv během jeho běhu a nedají se předpokládat. Jedná se například o přístup k nedeklarovaným proměnným nebo volání neexistujících metod. Jejich výskyt je možný kvůli tomu, že soubory modulů nejsou před načtením zkompilované žádným vývojovým prostředím, takže nedochází ke kontrole kódu. Není ani možné tyto chyby odchytávat a případně běh modulu přerušit, protože se nedá předpokládat, kdy k jaké chybě může dojít. Následkem takové chyby je pak pád aplikace, který se může vyskytnout bez příčiny zjevné pro uživatele. Z tohoto důvodu není v aplikaci zahrnuta plánovaná funkcionalita automatického zavádění modulů po restartu aplikace.

Aplikace byla v průběhu vývoje testována mj. na systémech Android 2.2 (API 8) a 2.3.7 (API 9), na kterých bylo možné ji spustit. Problém na těchto verzích systému nastal při snaze přeložit a zkompilovat Groovy kód modulu. V případě rozsáhlejšího kódu došlo dle chybových hlášení k přetečení zásobníku, do kterého byl ukládán zpracovávaný kód. V případě malého modulu (analogové hodiny) došlo k chybě:

„Could not find class 'java.lang.invoke.CallSite', referenced from method org.codehaus.groovy.classgen.asm.indy.InvokeDynamicWriter.<clinit>“

Dalvik VM v Android API 8 a API 9 tedy pravděpodobně neobsahuje prostředky potřebné ke kompilaci Groovy kódu. Pro testování na API 10-13 se nepodařilo získat zařízení s požadovanou verzí systému. Tyto verze systému nejsou podle oficiálních srovnání společnosti Google používány, nebo mají podíl na trhu menší, než 0,1%. (32) Z tohoto důvodu bylo určeno pro maximální zpětnou kompatibilitu API 14, neboli systém Android 4.0.

Přístup s využitím Groovy tříd nezkompilovaných před jejich načtením zaručuje transparentnost modulů z pohledu uživatele, který může zobrazit jejich kód a zkontrolovat případnou škodlivost. Je ale náchylný k chybovým situacím, které nemůže uživatel vždy předpokládat. Načítání modulů tímto způsobem je pomalejší a náročnější na výpočetní výkon (viz kapitola 7.3), než způsob popsaný kapitole 7.2.

7.2 Aplikace ApkModular

Aplikace umožňuje načítat moduly ve formátu fragmentů zkompilovaných do archivů Android aplikace (*apk*), viz kapitola 6.2.

7.2.1 Třídy logiky načítání modulů

FragmentLoader

Během chodu aplikace je spuštěna vždy právě jedna instance této třídy. Nabízí dvě metody, *findApkClasses* a *loadFragment*. Metoda *findApkClasses* prohledá v argumentu zadanou instanci třídy *file* jako adresář. Pokud tento adresář v paměti zařízení neexistuje, nebo se nejedná o adresář, ale o soubor, dojde k výjimce a vyhledávání se ukončí. Pokud jsou tyto podmínky splněny, metoda v adresáři vyhledá všechny soubory s příponou *apk* a navrátí seznam jejich názvů.

Metoda *loadFragment*, viz Ukázka kódu 8:

1. Řádek 12 – metoda obdrží v argumentu název souboru modulu, odkaz na adresář s moduly, odkaz na adresář pro dočasné soubory a instanci třídy *ClassLoader*.
2. Řádek 14 – metoda, stejně jako *findApkClasses,* zkontroluje validitu zadaného adresáře a případně vyhodí výjimku.
3. Řádek 15 - pokud je adresář validní, metoda vytvoří instanci třídy *DexClassLoader*, které je v parametru předán odkaz na soubor *apk* modulu – vytvořený složením názvu souboru a adresáře s moduly, adresář pro dočasné soubory a instance třídy *ClassLoader*.
4. Řádek 17 – na této instanci třídy *DexClassLoader* je dále zavolána metoda *loadClass*, jejímž parametrem je plný název třídy fragmentu, tj. včetně balíčku uvnitř jeho aplikace. Výsledek volání je instance obecné třídy *Class*. V průběhu volání metody *loadClass* může dojít k výjimce typu *ClassNotFoundException*, pokud nebyla třída v archivu nalezena. V takovém případě metoda *loadFragment* vyhodí výjimku a přeruší svůj běh.

5. Řádek 20-26 – dojde ke kontrole, jestli je možné získanou instanci C*lass* přetypovat na instanci třídy *Fragment*. Pokud ne, je vyhozena výjimka a metoda přeruší svůj běh, pokud ano, dojde k přetypování na třídu *Fragment* a zavolání metody *newInstance* na této třídě.
6. Řádek 27 – navrácená instance modulu, načtena jako fragment, je připravena k přidání do uživatelského rozhraní aplikace.

Aplikace GyModular (kapitola 7.1) používá třídu *Fragment* z balíčku *android.support.v4.app*, který umožňuje zpětnou kompatibilitu s Android API 8 (Android 2.2). Při použití této třídy v aplikaci ApkModular došlo při pokusu o její načtení vždy k výjimce *„java.lang.IllegalAccessError: Class ref in pre-verified class resolved to unexpected implementation"*, neboli „chyba neplatného přístupu: reference v před-verifikované třídě vedly k nečekané implementaci". Chybu pravděpodobně způsobuje konflikt v balíčku *android.support.v4.app* použitém v modulární aplikaci a v aplikaci modulu a to ať už mají balíčky stejnou verzi, nebo ne. Předpokládaným řešením bylo nekompilovat tento balíček do aplikace modulu – měl by pak být při načtení použit balíček z modulární aplikace. Toto řešení ale selhalo se stejnou chybou.

Funkčním řešením bylo použití třídy *Fragment* z balíčku *android.app* jak v modulární aplikaci, tak v aplikaci modulu – vzájemná shodnost je nezbytná. Toto řešení ale omezuje zpětnou kompatibilitu na Android API 11 (Android 3.0).

```
01 /**
02  * @param fileName          název souboru třídy
03  * @param moduleDirectory   složka s moduly
04  * @param cacheDir          dočasný pracovní adresář
05  * @param classLoader       classLoader hlavní aktivity aplikace
06  * @return načtený modul - instance třídy Fragment
07  * @throws FileNotFoundException  pokud nebyla nalezena složka s moduly
08  * @throws InstantiationException pokud není možné přetypovat třídu na
    fragment
09  * @throws IllegalAccessException pokud není možné třídy zinstanciovat
10  * @throws ClassNotFoundException pokud nebyla v cílovém souboru nalezena
    třída modulu
11  */
12 public Fragment loadFragment(String fileName, File moduleDirectory,
     File cacheDir, ClassLoader classLoader) throws FileNotFoundException,
     InstantiationException, IllegalAccessException, ClassNotFoundException {
13 
14    if (moduleDirectory.exists() && moduleDirectory.isDirectory()) {
15        ClassLoader dexLoader = new DexClassLoader(moduleDirectory.getPath()
            + "/" + fileName, cacheDir.getAbsolutePath(), null, classLoader);
16 
17        Class loadedClass = dexLoader.loadClass("ModuleFragment");
18 
19        Fragment fragment;
20        if (Fragment.class.isAssignableFrom(loadedClass)) {
21             fragment = (Fragment) loadedClass.newInstance();
22             Log.d("loadFragment", "fragment: " + fragment.toString());
23        } else {
24             throw new InstantiationException(loadedClass.toString() +
               " is not assignable to a Fragment!");
25        }
26        return fragment;
27 
28    } else {
29        throw new FileNotFoundException("Given module directory ( " +
               moduleDirectory + " ) doesn't exist or isn't a directory!");
30    }
31 }
```

Ukázka kódu 8 – Metoda loadFragment aplikace ApkModular

7.2.2 Třídy grafického uživatelského rozhraní

Třídy grafického uživatelského rozhraní jsou až na dále popsané rozdíly shodné s třídami popsanými v kapitole 7.1.2.

LoadFragmentTask

Ve volání metody *loadFragment* třídy *FragmentLoader* předává *ClassLoader* obalující aktivity získaný voláním *getClassLoader* a složku pro dočasné soubory získanou voláním *getCacheDir*.

MainActivity

Místo mapy s páry „název souboru modulu – název třídy modulu“ pracuje pouze se seznamem názvů souborů.

ViewPagerAdapter

Kvůli nahrazení balíčku *android.support.v4.app,* obsahujícího třídu *Fragment,* za balíček *android.app* (kapitola 7.2.1) je nutné, aby třída *ViewPagerAdapter* dědila po třídě *FragmentStatePagerAdapter* z balíčku *android.support.v13.app*. Původní třída *FragmentStatePagerAdapter* z balíčku *android.support.v4.app* nepodporuje práci s fragmenty z balíčku *android.app*.

7.2.3 Závěr

Z konečného řešení aplikace *ApkModular* vyplývá několik možných chybových situací:

- Aplikace se může stát nepoužitelnou, pokud nemá přístup do úložiště zařízení. Pokud o tento přístup přijde během načítání modulu, nebo je v takovou chvíli soubor modulu odstraněn, dojde k výjimce, která toto načítání přeruší, ale nepřeruší běh samotné aplikace.
- Pokud vnitřní struktura třídy modulu neodpovídá struktuře fragmentu, při jeho načítání dojde k výjimce, která toto načítání přeruší.
- Problém způsobují oprávnění aplikace deklarované v *AndroidManifest.xml*. Pokud se modul pokusí použít součást systému, ke které nemá aplikace předem deklarovaný přístup, dojde k chybě, která způsobí pád aplikace. Takovou akci může modul vykonat kdykoliv během svého běhu, takže aplikace může havarovat kdykoliv, bez příčiny zjevné pro uživatele. Aplikaci byla proto deklarována tato oprávnění: zápis a čtení externího úložiště zařízení, přístup k pozici zařízení, přístup ke kameře zařízení, přístup k internetu, přístup k bluetooth a přístup k NFC.
- Deklarace oprávnění v *AndroidManifest.xml* mohou způsobit problémy z hlediska bezpečnosti aplikace. Před načtením modulu není možné zkontrolovat jeho kód na potenciálně nebezpečný obsah. Takový obsah se tedy může spustit okamžitě po načtení modulu a uškodit uživateli.

Pro testování na API 10-13 (Android 3.0-3.2) se nepodařilo získat zařízení s požadovanou verzí systému. Tyto verze systému nejsou podle oficiálních srovnání společnosti Google používány, nebo mají podíl na trhu menší, než 0,1%. (32) Z tohoto důvodu bylo určeno pro maximální zpětnou kompatibilitu API 14, neboli systém Android 4.0.

Chybová situace, při které kód modulu obsahuje chyby tak, jak je popsáno v kapitole 7.1.3, by neměla nastat, protože kód modulu je zkontrolován kompilátorem vývojového prostředí, ve kterém je vyvíjen.

Přístup s využitím Java tříd zkompilovaných do archivu *apk* zaručuje vyšší stabilitu aplikace a redukuje uživatelem nepředpokládatelné chybové stavy. Zároveň snižuje transparentnost modulů z pohledu uživatele, který nemůže zobrazit jejich kód a zkontrolovat případnou škodlivost. Načítání modulů tímto způsobem je rychlejší a méně náročné na výpočetní výkon (viz kapitola 7.3), než způsob popsaný v kapitole 7.1.

7.3 Srovnání výkonu aplikací

Pro srovnání rychlosti načítání tříd aplikací *GyModular* a *ApkModular* byla vytvořena třída modulu identická pro obě aplikace. Modul byl naplněn ukázkovým kódem z webových stránek http://www.nealjohan.com, konkrétně příkladem *CarGoatSim.java*, (33) který demonstruje tzv. „Monty Hall Problem“. (34)

Výsledný fragment měl 117 řádků kódu. Obě aplikace měřily rychlost načítání od okamžiku zavolání metody *loadFragment* do vykreslení fragmentu s přesností na nanosekundy. Za nezměněných podmínek bylo v každé aplikaci provedeno 5 pokusů načtení modulu. Průměrný čas načítání v aplikaci *GyModular* byl přibližně 1,39134 sekundy, průměrný čas načítání v aplikaci *ApkModular* byl přibližně 0,02489 sekundy.

8 Shrnutí výsledků

V práci byly prozkoumány tři varianty tvorby modulární aplikace na platformě Android a modulů pro ni. Dvě varianty používají jako modul potomka třídy *Fragment*. První z těchto variant využívá pro načtení modulu kompilaci skriptovacího jazyka Groovy za běhu modulární aplikace. Mezi klady této varianty patří transparentnost modulů a možnost jejich snadné tvorby, záporem je vysoká pravděpodobnost neočekávatelných chybových stavů aplikace se zavedenými moduly a malá rychlost jejich zavádění. Druhá varianta využívá modulů zkompilovaných do souborů typu *apk* v libovolném prostředí pro vývoj aplikací pro systém Android. Výhodou takto připravených modulů je vysoká rychlost jejich zavádění a skutečnost, že došlo při jejich tvorbě ke kontrole kódu, takže jsou chybové situace vznikající za jejich běhu méně pravděpodobné. Nevýhodou je ztráta transparentnosti modulu z pohledu uživatele a jejich obtížnější tvorba. Nevýhodou obou těchto variant je nekompatibilita se systémy Android staršími než verze 4.0 a omezení vyplývající z nutnosti v modulární aplikaci jako takové deklarovat její přístupová práva, a tím pádem i přístupová práva modulů, které je potřeba při jejich tvorbě dodržet.

Třetí varianta umožňuje zavádění a spouštění potomků třídy *Activity* s využitím Java reflexe. Tento způsob nebyl zvolen pro tvorbu kompletní modulární aplikace, protože narušuje základní prvky bezpečnosti systému Android a využívá principy, které nejsou obecně považovány za vhodné pro zahrnutí do Java aplikací, není-li to nezbytně nutné.

9 Závěr a doporučení

Při tvorbě této práce bylo nutné řešit velké množství komplikací, ze kterých vyplývá, že systém Android není v současnosti vhodná platforma pro tvorbu modulárních aplikací – v systému neexistuje žádná přímá podpora takové funkcionality a to ani formou informací v dokumentaci prostředků pro vývoj aplikací na tento systém. Možností tvorby modulární aplikace pro systém Android i přesto existuje několik a tato práce popisuje jen zlomek z jejich množství.

Pro další zkoumání by bylo vhodné soustředit se na způsoby zefektivnění načítání modulů formou kompilace při načítání – například z nějakého skriptovacího jazyka, nebo na možnosti načítání aktivit bez využití reflexe. Zároveň je třeba věnovat pozornost otázkám bezpečnosti a zajištění stabilního běhu modulů.

10 Citovaná literatura

Překlad vlastní, pokud není uvedeno jinak.

1. **Hildenbrand, Jerry.** Android A to Z: What is Dalvik. *Androidcentral.* [Online] 5. Leden 2012. [Citace: 13. červenec 2014.] http://www.androidcentral.com/android-z-what-dalvik.

2. **Google Inc.** Signing Your Applications. *Android Developers.* [Online] [Citace: 13. červenec 2014.] http://developer.android.com/tools/publishing/app-signing.html.

3. **Christensson, Per.** Plug-in. *TechTerms.* [Online] [Citace: 13. červenec 2014.] http://www.techterms.com/definition/plugin.

4. **Oracle Corporation.** Java SE Development Kit 8 Downloads. *Oracle Technology Network.* [Online] [Citace: 1. Leden 2015.] http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html.

5. —. javac - Java programming language compiler. *ORACLE Help Center.* [Online] [Citace: 6. Leden 2015.] http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html.

6. **Venners, Bill.** Bytecode basics. *JavaWorld.* [Online] 1. Září 1996. [Citace: 1. Leden 2015.] http://www.javaworld.com/article/2077233/core-java/bytecode-basics.html.

7. **Oracle Corporation.** Java Programming Environment and the Java Runtime Environment (JRE). *Oracle Help Center.* [Online] [Citace: 1. Leden 2015.] http://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqd/index.html.

8. **JavaWorld.** The basics of Java class loaders. *JavaWorld.* [Online] 1. Říjen 1996. [Citace: 5. Leden 2015.] http://www.javaworld.com/article/2077260/learn-java/learn-java-the-basics-of-java-class-loaders.html.

9. **Google Inc.** Activities. *Android Developers.* [Online] [Citace: 10. srpen 2014.] http://developer.android.com/guide/components/activities.html.

10. —. Starting Another Activity. *Android Developers.* [Online] [Citace: 13. srpen 2014.] http://developer.android.com/training/basics/firstapp/starting-activity.html.

11. —. App Manifest. *Android Developers.* [Online] [Citace: 13. červenec 2014.] http://developer.android.com/guide/topics/manifest/manifest-intro.html.

12. **Ousterhout, John K.** Scripting: Higher Level Programming for the 21st Century. *Tcl Developer Xchange.* [Online] Březen 1998. [Citace: 5. Leden 2015.] http://www.tcl.tk/doc/scripting.html.

13. **Bloschetsov, Andrey et al.** Groovy Language Documentation. *Groovy.* [Online] 19. Prosinec 2014. [Citace: 1. Leden 2015.] http://beta.groovy-lang.org/docs/latest/html/documentation/.

14. **Google Inc.** Glossary. *Android Developers.* [Online] [Citace: 5. Leden 2015.] https://developer.android.com/guide/appendix/glossary.html.

15. —. DexClassLoader. *Android Developers.* [Online] [Citace: 1. Leden 2015.] http://developer.android.com/reference/dalvik/system/DexClassLoader.html.

16. **Anonymní.** BaseDexClassLoader.java. *android Git repositories.* [Online] [Citace: 10. Březen 2015.] https://android.googlesource.com/platform/libcore-snapshot/+/ics-mr1/dalvik/src/main/java/dalvik/system/BaseDexClassLoader.java.

17. **Champeau, Cédric.** me.champeau.gradle:gradle-groovy-android-plugin. [Online] [Citace: 5. Leden 2015.] https://maven-repository.com/artifact/me.champeau.gradle/gradle-groovy-android-plugin.

18. —. grooidshell-example. [Online] 25. Červenec 2014. [Citace: 1. Ledna 2015.] https://github.com/melix/grooidshell-example.

19. **The Apache Software Foundation.** Apache License, Version 2.0. *The Apache Software Foundation.* [Online] Leden 2004. [Citace: 10. Únor 2015.] http://www.apache.org/licenses/LICENSE-2.0.

20. **Jenkov, Jakob.** Java Reflection Tutorial. *Software Development & Entrepreneurship Tutorials.* [Online] [Citace: 10. Březen 2015.] http://tutorials.jenkov.com/java-reflection/index.html.

21. **Culpepper, Marshall.** A reflection hack to override the APK ClassLoader so you can launch Activities in an external JAR. *GitHUB Gist.* [Online] 22. Únor 2011. [Citace: 10. Březen 2015.] https://gist.github.com/marshall/839003.

22. **Google Inc.** Fragment. *Android Developers.* [Online] [Citace: 10. Leden 2015.] http://developer.android.com/reference/android/app/Fragment.html.

23. —. Accessing Resources. *Android Developers.* [Online] [Citace: 11. Duben 2015.] http://developer.android.com/guide/topics/resources/accessing-resources.html.

24. —. Building Apps with Over 65K Methods. *Android Developers.* [Online] [Citace: 11. Únor 2015.] https://developer.android.com/tools/building/multidex.html.

25. —. Android Studio Overview. *Android Studio.* [Online] [Citace: 11. Únor 2015.] http://developer.android.com/tools/studio/index.html.

26. **Anonymní.** JetBrains. *IntelliJ IDEA - The Most Intelligent Java IDE.* [Online] [Citace: 11. únor 2015.] https://www.jetbrains.com/idea/.

27. **Bornstein, Dan.** Diff - f4955a1^! - platform/dalvik - Git at Google. *android Git repositories.* [Online] 16. Březen 2011. [Citace: 3. Březen 2015.] https://android.googlesource.com/platform/dalvik/+/f4955a1%5E%21/.

28. **Google Inc.** AsyncTask. *Android Developers.* [Online] [Citace: 10. Březen 2015.] http://developer.android.com/reference/android/os/AsyncTask.html.

29. —. ActionBar. *Android Developers.* [Online] [Citace: 2. Březen 2015.] https://developer.android.com/reference/android/support/v7/app/ActionBar.html.

30. —. Storage Options. *Android Developers.* [Online] [Citace: 2. Březen 2015.] http://developer.android.com/guide/topics/data/data-storage.html.

31. —. FragmentStatePagerAdapter. *Android Developers.* [Online] [Citace: 2. Březen 2015.] http://developer.android.com/reference/android/support/v4/app/FragmentStatePagerAdapter.html.

32. —. Dashboards. *Android Developers.* [Online] 6. Duben 2015. [Citace: 13. Duben 2015.] https://developer.android.com/about/dashboards/index.html.

33. **Johan, Neil.** www.neiljohan.com/java/misc/CarGoatSim.java. *Neil Johan's Website.* [Online] [Citace: 10. Březen 2015.] http://www.neiljohan.com/java/misc/CarGoatSim.java.

34. **Wolfram Research, Inc. .** Monty Hall Problem. *Wolfram MathWorld.* [Online] [Citace: 10. Březen 2015.] http://mathworld.wolfram.com/MontyHallProblem.html.

35. **Champeau, Cédric.** Groovy on Android, technical details. *Cédric Champeau's blog.* [Online] 10. Červen 2014. [Citace: 5. Leden 2015.] http://melix.github.io/blog/2014/06/grooid2.html.