VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

ANDROID APLIKACE PRO GIT S PODPOROU GIT-LFS A GIT-ANNEX

THESIS TITLE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **PETR MAREK**
AUTHOR

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

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Marek Petr**
Program: Informační technologie
Název: **Android aplikace pro Git s podporou git-lfs a git-annex**
Android Application for Git with git-lfs and git-annex Support
Kategorie: Uživatelská rozhraní

Zadání:

1. Seznamte se s Git a jeho rozšířeními git-lfs a git-annex. Prozkoumejte existující aplikace (nejen pro operační systém Android) pro ovládání repositářů Git, git-lfs a git-annex, soustřeďte se zejména na aplikace s grafickým uživatelským rozhraním.
2. Navrhněte aplikaci pro operační systém Android, která umožní ovládat Git repositáře s podporou git-lfs a git-annex. Zaměřte se na uživatelskou přívětivost a minimalizaci velikosti repositářů ("shallow clone", ignorování a odstraňování nepotřebných souborů, aj.). Řešte také problémy kompatibility s úložištěm (např. podpora symbolických odkazů).
3. Po konzultaci s vedoucím aplikaci pro operační systém Android implementujte.
4. Řešení otestujte, vyhodnoťte a diskutujte výsledky. Výsledný software publikujte jako open-source.

Literatura:

- Ľuboslav Lacko. Vývoj aplikací pro Android. Computer Press, Brno, 2015. ISBN 978-80-251-4347-6.
- Scott Chacon. Pro Git. CZ.NIC, Praha, 2009. ISBN 978-80-904248-1-4

Pro udělení zápočtu za první semestr je požadováno:

- Body 1, 2 a započatá práce na bodu 3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Rychlý Marek, RNDr., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 14. května 2020
Datum schválení: 21. října 2019

Abstrakt

Tato práce má za cíl implementovat aplikaci umožňující použití programu Git a jeho rozšíření Git LFS a Git Annex na operačním systému Android. Poslouží zejména vývojářům k usnadnění práce s GIT a velkými soubory na tomto operačním systému. Uživatelské rozhraní je proto navrženo jako maximálně transparentní, za účelem efektivního řešení problémů vznikajících při použití Git.

Abstract

This thesis aims to design and develop an Android application implementing Git and its extensions Git LFS and Git Annex to the Android operation system. Its target audience is mainly developers looking for an effective way to work with Git and large files on an this operating system. Its user interface is therefore designed to provide a transparent environment, which makes it effective in solving Git related problems.

Klíčová slova

Android, Java, mobilní aplikace, Android Studio, Git, Git LFS, Git Annex, MVVM, LiveData, Room

Keywords

Android, Java, mobile application, Android Studio, Git, Git LFS, Git Annex, MVVM, LiveData, Room

Citace

MAREK, Petr. *Android aplikace pro Git s podporou git-lfs a git-annex*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce RNDr. Marek Rychlý, Ph.D.

Android aplikace pro Git s podporou git-lfs a git-annex

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana RNDr. Marka Rychlého Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Petr Marek
4. května 2020

Poděkování

Poděkovat bych chtěl panu RNDr. Markovi Rychlému Ph.D., vedoucímu mé bakalářské práce, za cenné rady a pomoc, když jsem ji nejvíce potřeboval.

Obsah

Kapitola 1

Úvod

Trendem poslední doby je neustálé zvětšování obrazovek mobilních zařízení i jejich výkonu. Dostali jsme se již do takové fáze, že je tyto zařízení možné využívat obdobně jako klasické počítače a tak je jejich využití pro synchronizaci souborů na snadě. Vyvíjená aplikace poskytuje systém, který může každý uživatel využít pro svůj účel a svým způsobem. Nejčastěji je *Git* využíván programátory pro verzování souborů vyvíjených programů. Rozšíření *Git LFS* a *Git Annex* slouží pro správu velkých souborů v *Git* repozitářích. Užitím *Git LFS* se dosáhne efektivity využití paměťového prostoru zařízení. *Git Annex* zase usnadní například ukládání videí nebo i jiných velkých souborů na externí úložiště, pro pozdější synchronizaci mezi různými zařízeními různých systémů.

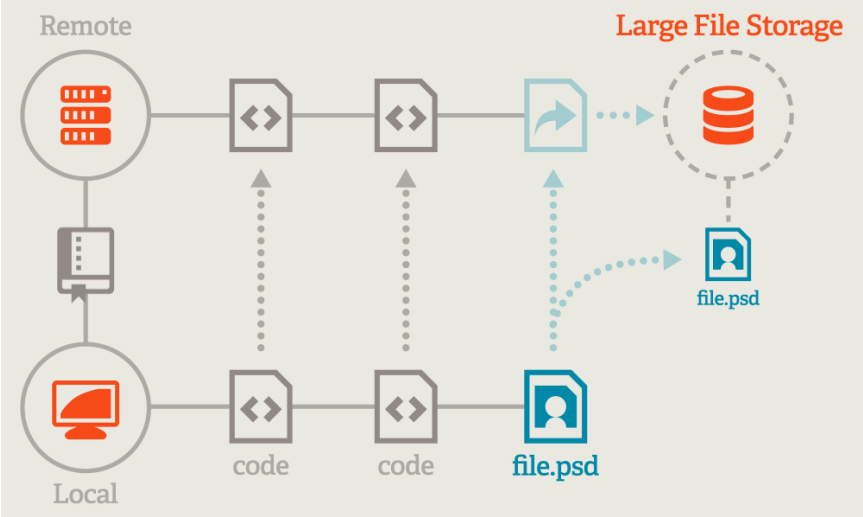
Cílem práce je navrhnout, implementovat a otestovat aplikaci určenou pro operační systém *Android*. Tato aplikace bude uživateli zprostředkovávat *Git* formou přívětivého grafického rozhraní. Dále bude implementovat rozšíření *Git LFS* a *Git Annex* pro práci s velkými soubory. Aplikace je určena zejména vývojářům a jiným pokročilým uživatelům. Je tedy navržena jako maximálně transparentní, za účelem efektivního řešení problémů vznikajících při používání *Git*.

1.1 Git

Git slouží zejména programátorům k verzování jejich práce i jejího sdílení s ostatními členy týmu. Jeho využití je široké a to zejména při využití rozšíření *Git LFS* [4] nebo *Git Annex* [9], která se zaměřují na práci s velkými soubory.

1.2 Git LFS

Git Large File Storage (*Git LFS*) nahrazuje velké soubory v repozitářích ukazateli. Samotné soubory jsou pak uloženy na vzdáleném serveru. Tento systém tedy slouží k efektivnímu uložení velkých souborů v *Git*. Jedná se například o video záznamy, zvukové stopy, datasety a jiné velké binární soubory.



Obrázek 1.1: Architektura *Git LFS* [4]

1.3 Git Annex

Git Annex [9] slouží k indexaci, synchronizaci a sdílení souborů mezi více úložišti, nezávisle na komerční službě nebo centrálním serveru [3]. V repozitáři je uložen symbolický odkaz na klíč, který je hash daného souboru. Samotný soubor je pak uložen v adresáři *.git/annex/*. Při změně souboru se mění jen jeho hash a aktualizuje symbolický odkaz. Tímto způsobem je zajištěno šetření místa, jelikož samotný soubor je v repozitáři uložen maximálně jednou. Odkazy na takto sledované soubory budou uloženy v repozitáři a uživatel tak v danou chvíli nemusí mít na paměti, kde jsou právě fyzicky uložené. Jelikož *Git Annex* používá jednoduchý formát *Git* repozitáře, je navíc garantováno, že tyto soubory budou v budoucnu dostupné i bez jeho použití.

Kapitola 2

Specifikace řešení

Hlavním cílem aplikace je nabídnout uživateli řešení pro verzování a synchronizaci velkých souborů jejich *Git* repozitářů na zařízeních systému *Android*.

2.1 Funkce aplikace

Aplikace bude mít dvě základní funkce. Jsou jimi správa repozitářů a provádění příkazů *Gitu*. Uživatel bude moci spravovat *Git* repozitáře následujícím způsobem. K přidání nového repozitáře bude mít tři možnosti. Může přidat adresář s daným repozitářem z místních souborů zařízení, ve zvolené složce inicializovat nový nebo klonovat vzdálený. Při otevření repozitáře nad ním bude provádět základní příkazy *Gitu* a také některé vybrané funkce již zmíněných rozšíření.

2.2 Cílová skupina

Cílovou skupinou jsou především programátoři nebo i jiní technicky zdatní uživatelé. Ti aplikaci využijí nejčastěji pro prohlížení jejich repozitářů, ale mohou je také jakkoliv měnit a pracovat na nich třeba i z veřejné dopravy. *Git Annex* využijí například pro přehlednou správu souborů uložených na více fyzických úložištích.

2.3 Průzkum existujících řešení

Během průzkumu již existujících aplikací jsem se zaměřil jak na aplikace operačního systému *Android*, tak na osobní počítače (dále jen PC) operačních systémů Linux a Windows.

2.3.1 Android

Pro operační systémy *Android* je trh s řešeními *Gitu* velice omezený. Existuje zde několik málo aplikací s podporou pouze pro čtení repozitáře ale i takové, které zvládají i ostatní základní příkazy *Gitu*. Jejich popis a mé postřehy z nich se dočtete na následujících řádcích.

MGit [1]

Za zmínku z nich stojí zejména MGit. Bohužel neposkytuje podporu pro *Git LFS* ani *Git Annex*. K implementaci příkazů *Gitu* využívá knihovnu *JGit*. *JGit* sice v aktuální verzi podporuje *Git LFS*, ale aplikace tuto podporu nemá. K hlavním přednostem aplikace *MGit* patří otevřený kód a velice intuitivní ovládání.

Úvodní obrazovka aplikace je seznam repozitářů. Po kliknutí na některý z nich se zobrazí obrazovka s jeho detaily. Nalezneme zde správce souborů, log a status repozitáře. Na této obrazovce se také nachází základní ovládací prvek *Gitu*. Jedná se o boční panel, který se vysunuje z pravé strany obrazovky. V něm jsou obsaženy všechny poskytované příkazy *Gitu*. Tedy jeho užívání je při porozumění obecného užívání *Gitu* jednoduché. Tato aplikace má integrovaný správce souborů i textový editor. Ovšem tento editor není dokonalý. Špatně se v něm posouvá kurzor a navíc nemaže konce řádků. Práce s ním je tedy spíše na obtíž. Naštěstí zde autoři přidali i možnost zvolení vlastního editoru z nainstalovaných aplikací. Správce souborů je také velice jednoduchý a pro plnou správu souborů spíše nedostačující.

Pocket Git [2]

Dále existuje například aplikace *Pocket Git*. Ta je placená a její kód není veřejně přístupný. Využívá integrovaného správce souborů, ale jejich editaci již nechává plně na jiných aplikacích. *Pocket Git* má na první pohled přehlednější uživatelské rozhraní. Jednotlivé příkazy *Gitu* rozděluje do různých kategorií a vedle souborů přidává ikonku o jeho stavu. Nicméně *Add* a *Commit* jsou natolik integrované do správce souborů, že jejich správné užití není vůbec intuitivní. Navíc při práci s touto aplikací často narazíte na nejednoznačná chybová hlášení, která neobsahují bližší popis chyby.

Termux [3]

Pro vývojáře upřednostňující příkazový řádek je možnost instalace aplikace *Termux* a nainstalování *Gitu* do prostředí jeho terminálu. Tam je i možné doinstalovat rozšíření *Git LFS* a *Git Annex*. *Git LFS* lze nainstalovat přímo jako balíček. *Git Annex* je možné stáhnout z jeho oficiálních webových stránek [4] a dle návodu [5]uvést do provozu. Obě tato rozšíření lze ovládat z příkazové řádky, přičemž *Git Annex* i formou uživatelského rozhraní. To je implementováno v prohlížeči. Tato webová aplikace je přehledná i pro mobilní zařízení a umožňuje synchronizaci souborů mezi repozitáři různých zařízení.

2.3.2 PC

Na Linux i Windows existuje mnoho aplikací, které práci s repozitáři zvládají velice dobře. Nicméně prostředí *Androidu* je od toho na PC natolik rozdílné, že prostor pro inspiraci je značně omezený.

[1] https://play.google.com/store/apps/details?id=com.manichord.mgit
[2] https://play.google.com/store/apps/details?id=com.aor.pocketgit&hl=en
[3] https://play.google.com/store/apps/details?id=com.termux&hl=en
[4] https://git-annex.branchable.com/
[5] https://git-annex.branchable.com/Android/

GitKraken [6]

Dobré zkušenosti mám například s aplikací *GitKraken*. Ta zobrazuje repozitář přehledně ve stromové struktuře. V ní lze přímo najetím myši na uzel *Commitu*, provádět změny. Příkazy *Gitu* má přehledně zobrazené v horním panelu. Navíc jsou zde dobře řešeny konflikty v souborech. Na jedné straně obrazovky vidíte jednu verzi a na druhé straně druhou. Ve spodní části obrazovky se generuje verze nová. Tu vytvářítе postupným procházením obou současných verzí a vybíráním vyhovující varianty. *GitKraken* umí pracovat i s *Git LFS*. K ovládání takto sledovaných souborů používá zvláštní vysouvací nabídku s funkcemi *Git LFS*. Ta se v případě práce s repozitářem podporující toto rozšíření zobrazí vedle základních příkazů. Výběr souborů, které takto sleduje, lze měnit v nastavení repozitáře nebo při přidávání souborů do *stage*.

Ungit [7]

Na první pohled dobrým dojmem působí i aplikace *Ungit*. Ta vás při každé akci naviguje krok po kroku a usnadňuje tak používání *Gitu* pro méně zkušené uživatele. Jedná se o webovou aplikaci založenou na *node.js*. Pro její instalaci je třeba příkazová řádka, pro spuštění pak webový prohlížeč. Její hlavní výhoda je tedy nezávislost na platformě. Její ovládání je rychlé, jelikož aplikace zjednodušuje určité procedury *Gitu*. Například sama nabízí `Commit` bez nutnosti přidávat soubory do `Stage`. Nicméně aplikace tím velice zapouzdřuje příkazy *Gitu*. Na základní obrazovce kromě stromu změn repozitáře není další ovládací prvek a aplikace se tak v konečném důsledku jeví až příliš uzavřeně.

2.3.3 Zhodnocení průzkumu

Z testování aplikací vyplynulo, že nejjednodušší způsob práce s *Gitem* je tehdy, když aplikace transparentně zobrazuje příkazy *Gitu* a jejich použití nechá na uživateli. Předejde se tím chybám, jejichž hlášení nejsou vždy dostačující k vyřešení problému. Pokud je funkce dobře zpracována, není třeba vést uživatele krok po kroku. Ovládání se tak urychlí a je stále přehledné.

Testované mobilní aplikace často využívají vlastní textový editor a správce souborů. V obou případech tyto aplikace integrují velice jednoduché verze a jejich použitelnost je tak značně omezená.

Dalším bodem jsou chybová hlášení. Těm by měla aplikace pokud možno předcházet. Pokud chybě již není vyhnutí, alespoň by měla mít přesný popis a nebo i návrh jejího řešení.

Poslední bod se týká uživatelského rozhraní. Aplikace *MGit* při klonování repozitáře skrývá určité položky při jejich nadbytečnosti. To je sice užitečný prvek, nicméně při skrytí položky dojde k posunutí těch následujících na její místo a to působí velice rušivě.

[6] https://www.gitkraken.com/
[7] https://github.com/FredrikNoren/ungit

Kapitola 3

Vývoj aplikací pro systém Android

Android je open-source platforma vyvinutá společností *Google*. Její první oficiální verze se dostala na svět 23. Října 2008 a od té doby značně vyspěla. Je založena na systému Linux. Většina fyzických zařízení, které ji podporují staví na architektuře *arm*. *Android* totiž není mířen přímo na konkrétní zařízení tak jako například *iOS* od firmy *Apple*. To přináší mnohé kompromisy, které musí postupovat jak její vývojáři, tak samotní programátoři aplikací. Zařízení se liší svým hardwarem i softwarem. Mají různé velikostí pamětí i displejů.

Při vývoji *Android* aplikací je tedy nutné brát ohled na nejnovější trendy a sledovat procentuální zastoupení kritických parametrů tak, aby výsledná aplikace splňovala zadané požadavky na většině cílových zařízení. To má za následek roztříštěnost aplikací podle mnoha kritérií tak, aby byly plně funkční na co možná největším počtu zařízení. Naštěstí je při vývoji na této platformě k dispozici mnoho nástrojů, se kterými je možné se s těmito problémy vypořádat.

Tato kapitola se zabývá teoretickými základy tohoto systému, které je užitečné mít pro úspěšný vývoj aplikací na paměti. Při získávání přehledu o principu programování *Android* poslouží zejména oficiální online dokumentace [1] a návody [2]. Především z těchto návodů čerpá následující text, pojednávající o základech aplikací. Také je možné najít různou kvalitní tištěnou literaturu. Například Pro účely programování této aplikace se například osvědčila kniha *Vývoj aplikací pro Android*[10].

3.1 Základy aplikace

Aplikace pro *Android* mohou být psány v Kotlinu, Javě, nebo C++. Nástroje *Android SDK* kompilují kód spolu s ostatními potřebnými daty do *APK* souboru. Prakticky se jedná o *zip* archiv, který *Android* používá pro instalaci aplikací.

Každá aplikace pracuje ve svém vlastním uzavřeném prostoru. *Android* implementuje princip nejmenších pověření - *principle of least privilege*. Ten zaručuje, že každá aplikace má práva k přístupu jen ke zdrojům, které potřebuje. Další práva lze aplikaci přiřadit pouze s explicitním souhlasem uživatele.

[1] https://developer.android.com/docs
[2] https://developer.android.com/guide/

3.2 Android ABI

Různá zařízení mají osazeny různý hardware a tedy i procesory. Různé procesory používají různé instrukční sady. Každá kombinace procesoru a instrukční sady má vlastní aplikační binární rozhraní - *Application Binary Interface (ABI)*. *ABI* [3] zahrnuje následující informace:

- Instrukční sadu.
- *Endian* načítání a ukládání paměti. *Android* je vždy *little-endian*.
- Konvenci sdílení dat mezi aplikacemi a systémem.
- Formát spustitelných binárních souborů. *Android* vždy používá *ELF* [4].
- Formu implementace *C++* kódu [5].

3.3 Komponenty aplikace

Komponenty aplikace jsou základním stavebním blokem každé *Android* aplikace. Tyto komponenty jsou vstupním bodem aplikace pro uživatele i pro samotný systém.

3.3.1 Typy komponent

Existují čtyři typy komponent:

- **Activities** - Aktivity
- **Services** - Služby
- **Broadcast receivers** - přijímače vysílání
- **Content providers** - Poskytovatele obsahu

Activities

Aktivita je vstupní bod uživatele každé aplikace a reprezentuje jednu obrazovku aplikace. Příkladem takové aktivity je seznam přijatých *SMS* zpráv. Po kliknutí na danou zprávu pro její otevření, dojde k vyvolání další aktivity, která zobrazuje obsah této zprávy. Jiná aktivita může zajišťovat obrazovku pro odpověď na tuto zprávu, další její přeposlání jinému příjemci. Aktivita zajišťuje následující interakce mezi systémem a aplikací:

- Sledování aktivity uživatele k tomu, aby systém udržoval aktuální proces spuštěný.
- Aktivita má přehled o předchozích přerušených aktivitách, ke kterým by se uživatel mohl vrátit.
- Při zrušení aktuálního procesu aktivity pomáhá aplikaci k vrácení se do předchozí aktivity.
- Poskytuje systému způsob k implementaci přechodů mezi aktivitami

[3] https://developer.android.com/ndk/guides/abis
[4] https://linuxhint.com/understanding_elf_file_format/
[5] http://itanium-cxx-abi.github.io/cxx-abi/

Services

Services nebo také *služby* poskytují způsob pro udržení aplikace běžící na pozadí. Například se jedná o přehrávání hudby, stahování souborů a podobné akce. Tyto dvě akce reprezentují dva různé způsoby využití služeb:

- Přehrávání hudby je akce, kterou uživatel v reálném čase vnímá a systém tak musí její proces prioritně udržovat v chodu.
- Stahování souborů nebo jinému zpracování dat na pozadí uživatel nevěnuje veškerou svoji pozornost a proto má systém větší volnost při správě tohoto procesu.

Pro svoji flexibilitu se *services* staly velice užitečným stavebním blokem různých funkcí systému. Jsou jimi implementovány živá pozadí, upozornění na různé akce, spořiče obrazovky a mnoho dalších funkcí.

Broadcast receivers

Broadcast receiver je komponenta, která umožňuje systému nebo jiné aplikaci doručit událost cílové aplikaci mimo běžné uživatelské rozhraní. Tyto události může systém aplikaci doručit i pokud zrovna aplikace neběží. Příklad takové události může být upozornění kalendáře na naplánovanou událost. Aplikace kalendáře nemusí běžet, systém přesto tuto událost doručí *broadcast receiveru* aplikace, která toto upozornění zobrazí. *Broadcast receiver* nezobrazuje uživatelské rozhraní, ale může poskytovat upozornění v notifikacích, které uživateli vznik dané události oznámí.

Content Providers

Content providers, v překladu poskytovatelé obsahu, spravují data aplikace. Tyto data mohou být uloženy v souborovém systému, v *SQLite* databázi, na internetu nebo na kterémkoliv jiném trvalém úložišti, ke kterému má aplikace přístup. *Content providers* poskytují rozhraní k přístupu k těmto datům. Data jsou identifikována svým *URI - Uniform Resource Identifier*. Systém po obdržení daného *URI*, rozhodne na základě práv aplikace o přiřazení daného zdroje dat. *URI* tedy slouží obdobně jako absolutní cesta v souborovém systému, jen její užití je univerzální.

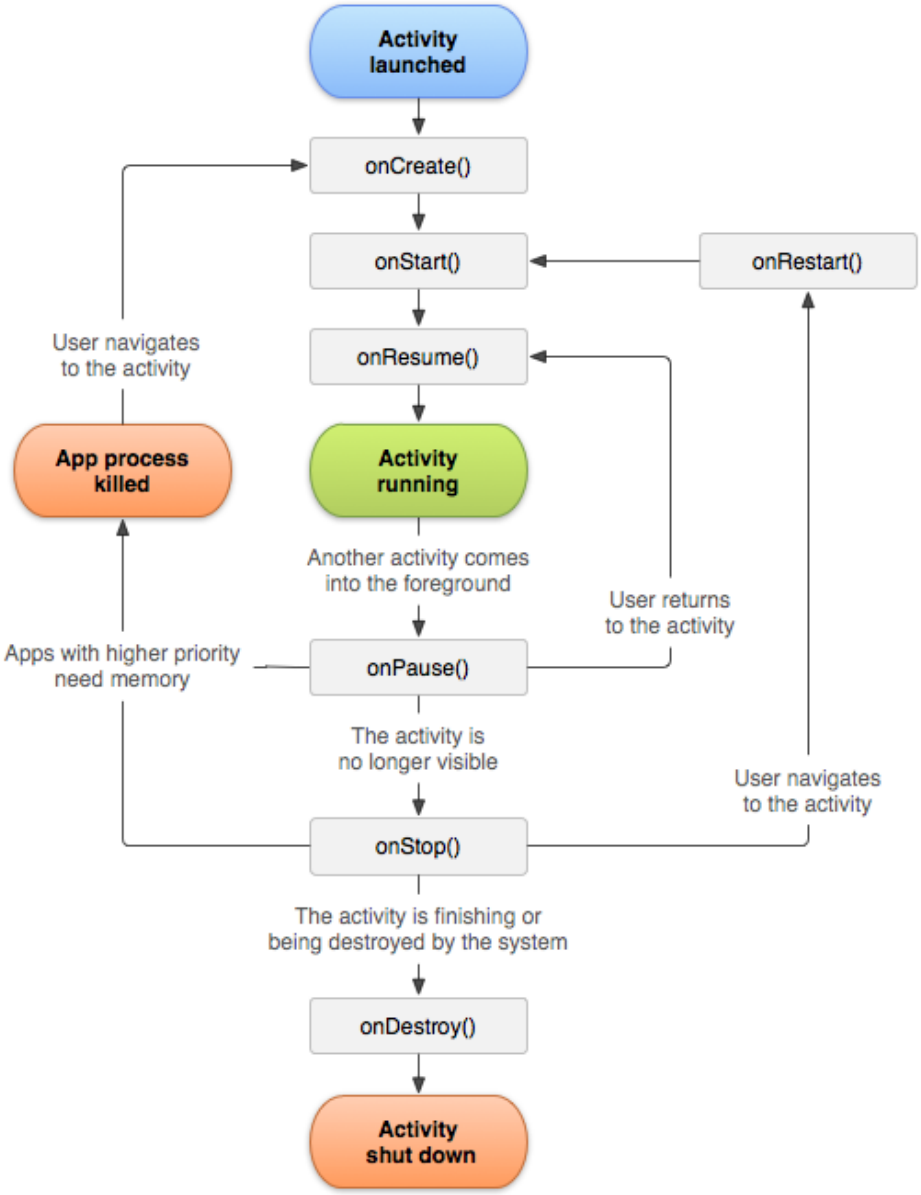
3.3.2 Aktivace komponent

Activity, *service* a *broadcast receiver* jsou aktivovány mechanismem zvaným *intent* [6]. *Intenty* během chodu aplikace propojují jednotlivé komponenty tak, aby mohly vzájemně spolupracovat. Fungují tedy jako most, přes který si mezi sebou komponenty vyměňují informace.

[6] https://developer.android.com/reference/android/content/Intent

3.4 Životní cyklus aktivity

Protože uživatel má plnou volnost přecházet mezi aplikacemi i mezi aktivitami, je často potřeba pro různé účely detekovat změny těchto stavů. K tomu aktivita poskytuje řadu metod zpětného volání tzv. *callback*. Ty jsou volány v případě, že systém aktivitu vytváří - **OnCreate()**, přerušuje - **onPause()**), zastavuje - **onStop()**, obnovuje - **onRestart()** nebo ruší - **onDestroy()**.



Obrázek 3.1: Životní cyklus aktivity [7]

3.5 Architektura aplikace

Mobilní prostředí aplikací je velice odlišné od toho na PC, což má vliv i na architekturu aplikací. Zejména je to dáno tím, že uživatel často využívá kooperace více aplikací. Například pro sdílení snímků fotoaparátu na sociálních sítích, daná aplikace vyvolá *intent* pro spuštění kamery. Ten spustí jinou aplikaci, poskytující rozhraní pro pořízení snímku. Tato aplikace po jeho pořízení, předá snímek původní aplikaci a ta ho dále zpracovává. Tento způsob práce je pro *Android* typický a aplikace s ním musí umět pracovat.

Je důležité mít na paměti, že komponenty mohou být spuštěny v různém pořadí a operační systém může jejich běh v jakoukoliv dobu ukončit. Proto by data a stav aplikace neměli být uchovávány v rámci těchto komponent.

Tato kapitola čerpá informace z oficiálních návodů pro architekturu *Android* [7]. Jedná se o doporučenou architekturu pro vývoj mobilních aplikací a aplikace, které se týká tato práce je na ní postavena. Pro implementaci této architektury se využívá knihovny *Android Jetpack* [8].

3.5.1 Základní architektonické principy

Existují dva základní architektonické principy. Jejich použití řeší problém, kde uchovávat data a stav aplikace tak, aby byly odděleny od uživatelského rozhraní. Aplikace se tím stává konzistentní a dobře testovatelná.

Oddělení zodpovědnosti

Základem této architektury je, aby aktivita nebo fragment implementoval pouze logiku týkající se uživatelského rozhraní. To umožní vyhnout se mnoha problémům, které jsou způsobené životním cyklem aplikace.

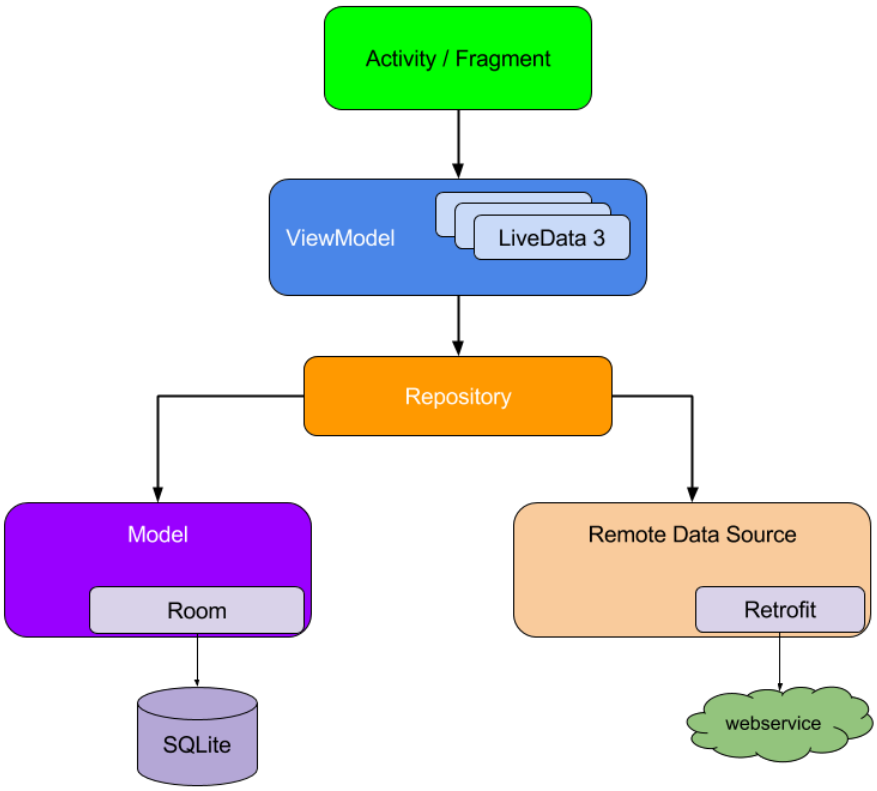
Uživatelské rozhraní řídí model

Dalším důležitým bodem je řídit uživatelské rozhraní z perzistentního modelu. *Model* je komponenta, která má za úkol spravovat data aplikace. Je nezávislá na objektech uživatelského rozhraní a komponent aplikace tak, aby nebyla ovlivněna jejím životním cyklem.

[7] https://developer.android.com/jetpack/docs/guide
[8] https://developer.android.com/jetpack

3.5.2 Doporučená architektura

Následující diagram zobrazuje závislosti jednotlivých komponent doporučené architektury. Jejich dodržení je základem pro splnění zmíněných architektonických principů. Tuto architekturu lze také popsat jako *Model, View, ViewModel* - zkráceně *MVVM*. Vyjma části *Remote Data Source* ji implementuje i aplikace této práce.



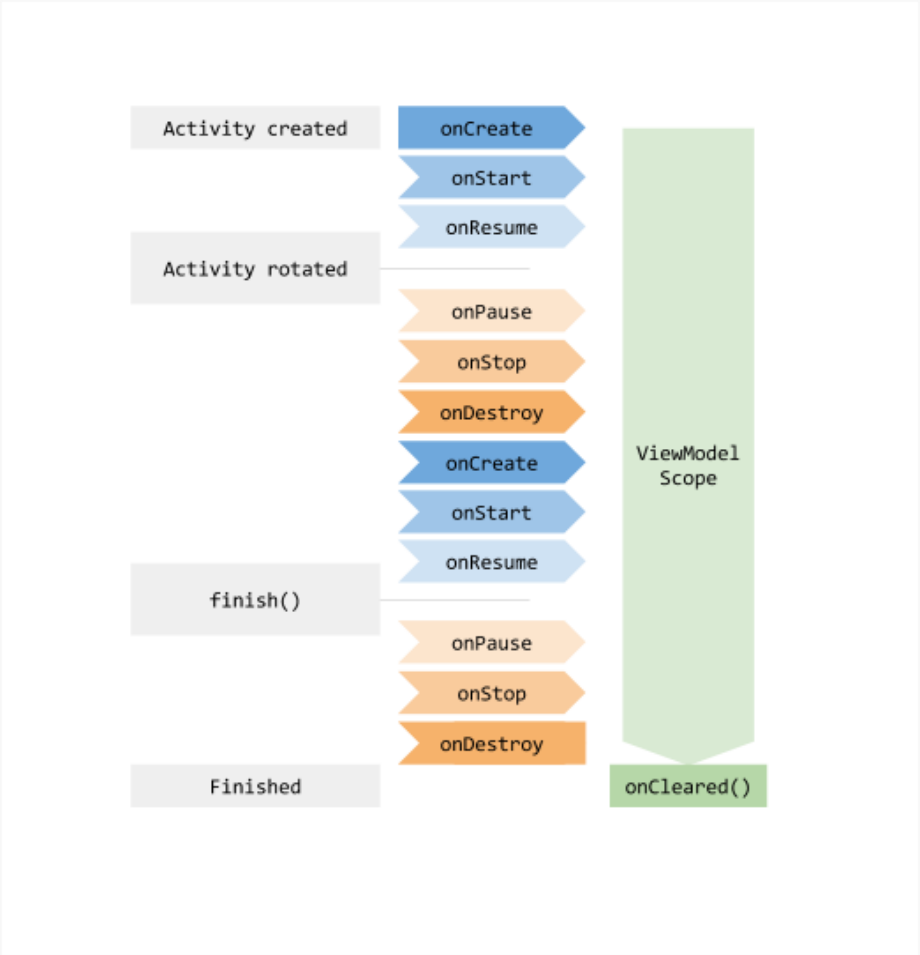
Obrázek 3.2: Doporučená architektura [5]

1. Třída **Activity / Fragment** implementuje logiku uživatelského rozhraní. V terminologii *MVVM* se jedná o *View*.
2. **ViewModel** [9] - třída, umožňující ukládat a spravovat data aplikace nezávisle na jejím životním cyklu.
3. **LiveData** [6] - třída, obsahující data. Je uzpůsobena ke komunikaci mezi *View* a *ViewModelem*.
4. **Repository** - třída, poskytující operace nad daty.
5. **Model** - třída, definující data databáze.
6. **Room** [10] - knihovna poskytující abstraktní vrstvu nad *SQLite* databází.

[9] https://developer.android.com/reference/androidx/lifecycle/ViewModel
[10] https://developer.android.com/topic/libraries/architecture/room

3.5.3 ViewModel

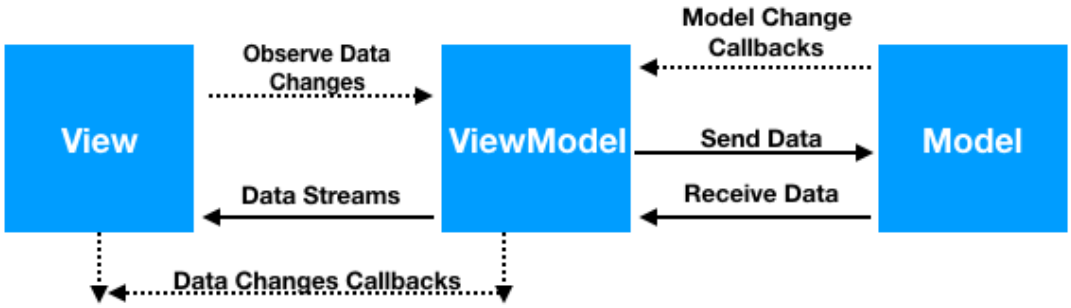
Stěžejní částí této architektury je *ViewModel*. Do něj se přesouvá logika aplikace, data i operace nad nimi. *ViewModel* je inicializován v rámci aktivity nebo fragmentu. Jak již bylo naznačeno v předchozí kapitole 2, data uložená v jeho rámci přetrvávají změny stavu spojené s životním cyklem aktivity / fragmentu. Jak dokládá i 3.3, jeho stav zůstává v paměti dokud neskončí daná aktivita nebo není odpojen příslušný fragment.



Obrázek 3.3: Životní cyklus ViewModelu[8]

3.5.4 Architektonický vzor MVVM

Pro správné užití tohoto architektonického vzoru je nutné dodržovat následující principy správné komunikace. Pro komunikaci směrem z *View* do *ViewModelu* slouží *Observer* [6]. Ten sleduje data *LiveData* a při jejich změně vykoná předem definované operace.



Obrázek 3.4: Vzor MVVM [2]

3.5.5 Příklad inicializace LiveData

Následující kód názorně předvádí inicializaci *LiveData*. Jedná se o *wrapper*, který může obsahovat jakákoliv data. Může tedy i implementovat kolekce, například seznam.

Z důvodu zachování její hodnoty v rámci životního cyklu aktivity se inicializace provádí ve *ViewModelu*.

```
public class NameViewModel extends ViewModel {

    // Create a LiveData with a String
    private MutableLiveData<String> currentName();

    public MutableLiveData<String> getCurrentName() {
        if (currentName == null) {
            currentName = new MutableLiveData<String>();
        }
        return currentName;
    }
    // Rest of the ViewModel...
}
```

Výpis 3.1: Inicializace *LiveData* [6].

3.5.6 Příklad získání hodnoty LiveData

Hodnota *LiveData* potřebná pro aktualizaci uživatelského rozhraní se získává pomocí *Observeru*. K odběru změn se přihlásíme z Aktivity / Fragmentu, kde danou hodnotu potřebujeme.

```
public class NameActivity extends AppCompatActivity {

    private NameViewModel model;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Other code to setup the activity...

        // Get the ViewModel.
        model = new ViewModelProvider(this).get(NameViewModel.class);

        // Create the observer which updates the UI.
        final Observer<String> nameObserver = new Observer<String>() {
            @Override
            public void onChanged(@Nullable final String newName) {
                // Update the UI, in this case, a TextView.
                nameTextView.setText(newName);
            }
        };

        // Observe the LiveData, passing in this activity as the
            LifecycleOwner and the observer.
        model.getCurrentName().observe(this, nameObserver);
    }
}
```

Výpis 3.2: Získání hodnoty LiveData [6]

Kapitola 4

Návrh aplikace

Po zhodnocení průzkumu i vlastních zkušeností, byly na aplikaci stanoveny následující požadavky. Aplikace bude:

1. transparentně poskytovat příkazy *Gitu*.
2. uživatele přehledně informovat o tom co právě dělá, co očekává a jaký je výstup.
3. využívat externí správce souborů.
4. mít co nejmenší počet za sebou následujících aktivit a tedy i přechodů mezi nimi.

4.1 Funkce aplikace

Git je velice komplexní systém a proto hrozí, že jeho plná implementace by na zařízeních *Android* byla nepřehledná. Vybrány byly tedy nejdůležitější funkce, které jsou nutné pro základní správu repozitářů.

4.1.1 Správa repozitářů

Po spuštění aplikace uživatele uvítá obrazovka se seznamem sledovaných repozitářů. Repozitář je možné do něj přidat několika způsoby. Prvním je přidání již existujícího repozitáře specifikováním jeho cesty v úložišti zařízení. Druhým je klonování nebo inicializace repozitáře z prostředí aplikace. Tento seznam repozitářů je synchronizován s úložištěm zařízení. Příkazy *Gitu* bude moci uživatel provádět po otevření daného repozitáře.

4.1.2 Příkazy Gitu

Jelikož žádné aplikace pro *Android* kromě *Termux* neimplementují rozšíření *Git Annex* a nenalezl jsem knihovnu, která by toto dokázala, bylo rozhodnuto pro příkazy *Gitu* i rozšíření využít zkompilované binární soubory. Všechny tyto příkazy budou dostupné při otevření repozitáře v bočním výsuvném panelu aplikace. Pro transparentní zobrazení stavu repozitáře, budou tyto funkce zobrazovat i svůj klasický textový výstup.

Git

Pro umožnění základního užití nástroje *Git* budou uživateli dostupné následující příkazy:

- **`add`** - přidání souborů do *stage*
- **`commit`** - vytvoření nového *commitu* - potvrzení změn a vytvoření nového uzlu *Gitu*
- **`push`** - nahrání obsahu nových *commitů* na vzdálený server
- **`pull`** - stažení obsahu nových *commitů* ze vzdáleného serveru
- **`status`** - výpis stavu souborů repozitáře
- **`log`** - detailní výpis *commitů* repozitáře
- **`reset --hard`** - obnova změněných souborů, které ještě nejsou součástí aktuálního uzlu
- **`remote add origin`** - přidání *URL* nového vzdáleného serveru
- **`remote set-url origin`** - editace *URL* aktuálního vzdáleného serveru
- **`branch`** - výpis větví
- **`checkout <branch>`** - přepnutí z aktuální větve na větev `<branch>`
- **`checkout --track <branch>`** - stažení obsahu vzdálené větve a přepnutí z aktuální větve na tuto novou větev

Git LFS

Pro rozšíření *Git LFS* pak budou dostupné tyto příkazy:

- **`track <pattern>`** - přidání souborů, které odpovídají regulárnímu výrazu *pattern* sledovaných souborů
- **`untrack <pattern>`** - ukončení sledování souborů daných výrazem *pattern*
- **`track`** - výpis sledovaných výrazů
- **`ls-files`** - výpis sledovaných souborů
- **`status`** - výpis stavu repozitáře s *Git LFS*
- **`env`** - výpis prostředí *Git LFS*, pro hledání řešení chyb repozitáře

4.2 Použité technologie a nástroje

Před samotným programováním aplikace bylo třeba udělat průzkum nástrojů, které se při vývoji na zařízení *Android* používají. Tyto nástroje byly vybrány s důrazem na efektivitu vývoje i náročnost jejich použití. Nejdůležitějším z nich je *Android Studio* [1]. Pro verzování byl použit nástroj *Git*, prostřednictvím aplikace *GitKraken* [2]. Kód aplikace byl synchronizován se vzdáleným repozitářem na serveru *GitHub* [3]. Pro dynamické generování instalačních souborů aplikace byly využity *GitHub Actions* [4]. Pro vytváření binárních souborů *Gitu* a *Git LFS* byl použit *Docker* [5]. Obraz *Dockeru* pro jejich kompilace poskytuje aplikace *Termux packages* [6].

4.3 Architektura aplikace

Aplikace je psána v jazyce *Java*. Dále využívá návrhového vzoru *Model–view–viewmodel* (dále jen *MVVM*). K implementaci této architektury aplikace využívá knihovny *Android Jetpack* [7].

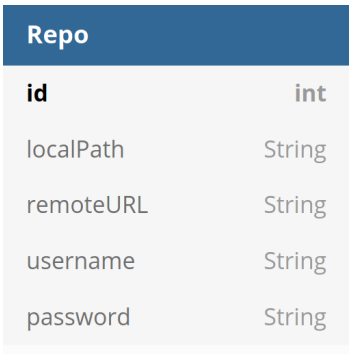
4.3.1 Kotlin vs. Java

Od 7. května 2019 se Kotlin stal preferovaným jazykem vývoje pro *Android*. Nicméně při přípravě k programování aplikace bylo odhaleno, že naprostá většina zdrojů na internetu pro řešení problémů pro tuto platformu je psána v Javě. *Android Studio* sice umožňuje zkonvertovat kód do Kotlinu, ale ani to se neobejde bez další práce. Užití Kotlinu má tu výhodu, že dovoluje programátorovi vynechat určité části kódu, které jsou nutné pro běh aplikace, ale přímo neřeší daný problém. V angličtině se pro ně vžil výraz *boilerplate code*. Ovšem tento kód je přesto třeba vygenerovat, ale o to se již stará Kotlin. To je také jeden z důvodů, proč kompilace Kotlin trvá déle. Pokročilým *Android* vývojářům jistě přijde rychlejší práce vhod, ale začínající programátor této platformy více ocení transparentnost Javy.

[1] https://developer.android.com/studio
[2] https://www.gitkraken.com/
[3] https://github.com/
[4] https://github.com/features/actions
[5] https://www.docker.com/
[6] https://github.com/termux/termux-packages
[7] https://developer.android.com/jetpack

4.3.2 Databáze

Databáze je využívána pro získání přehledu o repozitářích *Gitu*, které uživatel aplikací sleduje. Každý takový repozitář je reprezentován entitou databáze `Repo`. Tato tabulka obsahuje absolutní cestu ke složce repozitáře, URL vzdáleného serveru a uživatelské jméno a heslo pro přístup k tomuto serveru. Tyto položky je třeba při provádění příkazů *Gitu* aktualizovat tak, aby stav entity v okamžitém časem odpovídal stavu repozitáře.



Obrázek 4.1: Entita repozitáře

4.3.3 Návrhový vzor

Model–view–viewmodel je v době psaní této práce doporučovaným návrhovým vzorem *Android* aplikací. K volbě tohoto návrhového vzoru dopomohlo také využití knihovny *Room*. Tato knihovna totiž spoléhá na využití *MVVM* vzoru alespoň pro účely funkčnosti databáze. Je tomu tak proto, že data, která závisí na databázi se ukládají do proměnné datového typu *LiveData*. Hodnotu této proměnné lze sledovat a na jejím základě řídit běh aplikace. Aby byla hodnota této proměnné perzistentní při běhu aplikace, uchovává se její hodnota ve *ViewModelu*. Hlavní takovou proměnnou je v této aplikaci seznam všech *Git* repozitářů, `mAllRepos`, který se nachází ve třídě `RepoRepository`.

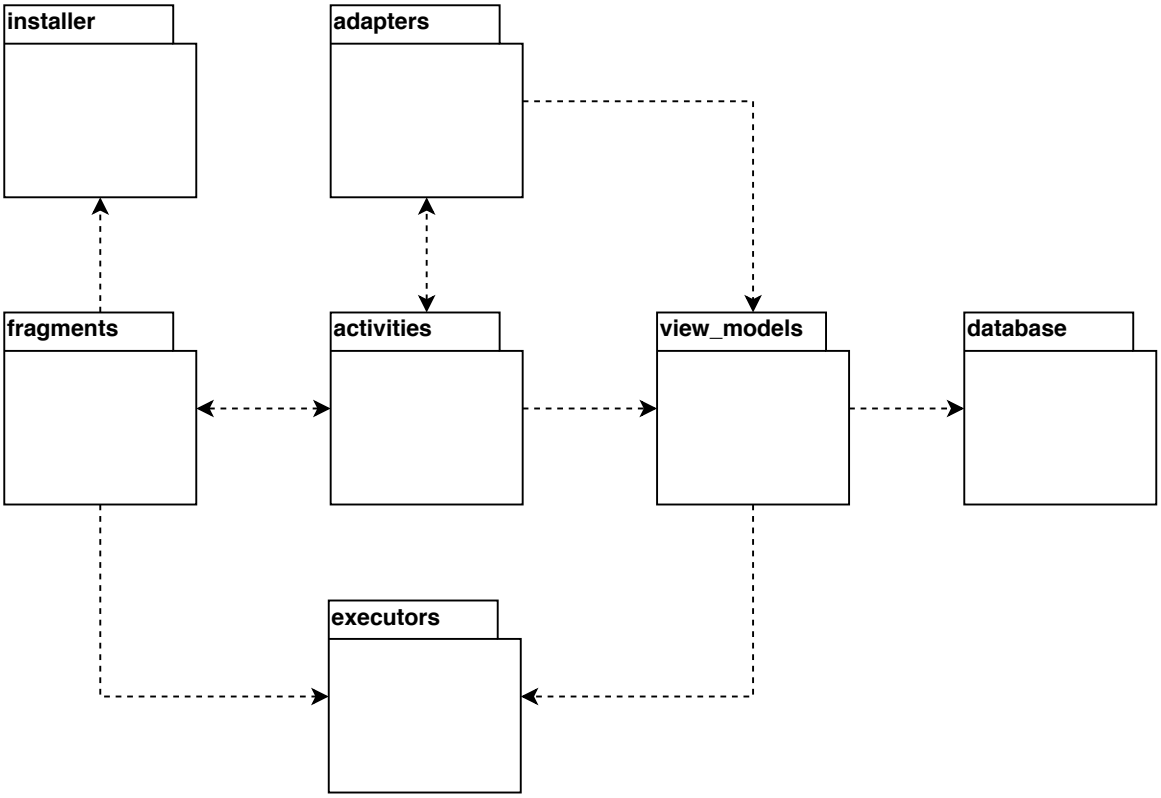
4.3.4 Obrazovky aplikace

Aplikace bude rozdělena podle obrazovek do aktivit. Tyto aktivity dále mohou obsahovat různé fragmenty. Ke každé aktivitě, která poskytuje určitou obrazovku je připojen její *ViewModel*. Ten jí poskytuje data a funkcionalitu. Vzhled obrazovky je dán jejím *layoutem*.

4.3.5 Dělení aplikace do balíčků

Podle zaměření tříd je aplikace dělena do třech základních balíčků. Jsou jimi `java`, `cpp`, a `res`. V balíčku `java` je specifikováno chování aplikace. Hlavním obsahem balíčku `cpp` jsou archivy s binárními soubory *Gitu* a program `bootstrap.c` pro jejich instalaci. Posledním balíčkem je `res`. Ten obsahuje všechny *layouty*, ikony a další grafické i textové prvky, které aplikace využívá pro grafické rozhraní.

Následující popis funkčnosti a závislostí balíčků se bude týkat balíčku `java`. Záměrně byl z diagramu vynechán balíček `utilities`. Jeho třídy lze použít kdekoliv v aplikaci a pro budoucí vývoj aplikace jeho zahrnutí nemá opodstatnění.



Obrázek 4.2: Diagram závislostí balíčků

activities

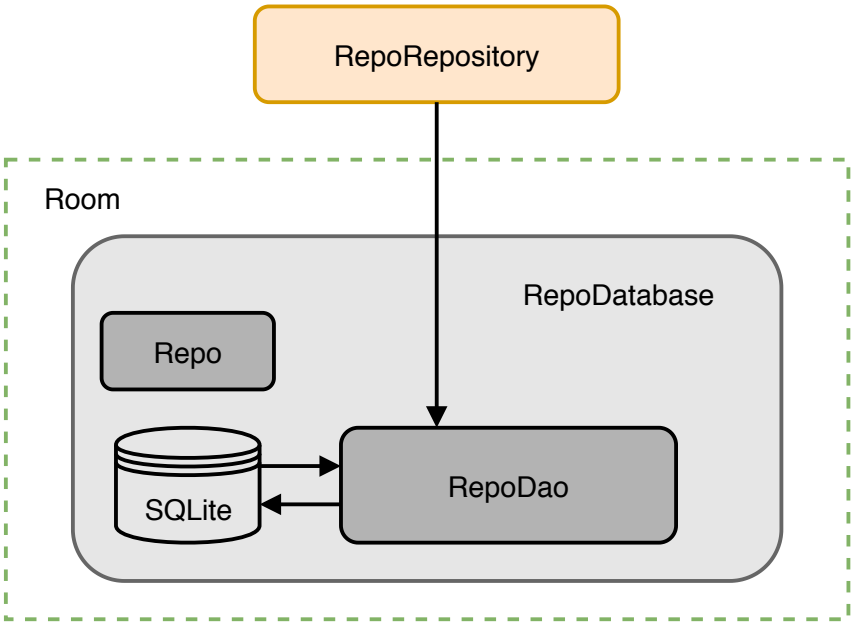
Nejdůležitější součást balíčku `Java` je balíček `activities`. Ten aplikaci dělí na obrazovky.O každou z nich se stará jedna třída. V případě, že aplikace potřebuje určitou obrazovku, je zavolána příslušná aktivita s jejím chováním. Všechny aktivity aplikace rozšiřují základní aktivitu `BasicAbstractActivity`. Ta implementuje společné prvky rozhraní aktivit. Například získávání oprávnění, zobrazení různých oznámení a dialogů.

adapters

Tento balíček obsahuje třídy, které slouží k zobrazení položek stejného typu. Tato aplikace je využívá k zobrazení seznamu repozitářů základní obrazovky a příkazů *Gitu* v bočním výsuvném panelu. Adaptér `RepoTasksAdapter.java` pro zobrazení tohoto výsuvného panelu byl převzat z implementace aplikace *MGit* třídy `RepoOperationsAdapter.java` [8] a následně upraven pro potřeby aplikace.

database

Tento balíček obsahuje balíček `model`, ve kterém se nachází třída `Repo`. Ta implementuje tabulku databáze, uchovávající všechny potřebné informace o repozitáři. Instance databáze se uchovává ve třídě `RepoDatabase`. K přístupu k ní se využívá třída `RepoDao`. Tato třída obsahuje metody volající *SQLite* dotazy databáze. Aplikaci je databáze zprostředkována třídou `RepoRepository`, která odpovídá `Repository` modelu *MVVM*. Databáze se ukládá do bezpečného vnitřního prostoru balíčku aplikace. Pro získání abstraktní vrstvy nad databází aplikace využívá knihovny *Room Persistence Library* [9].



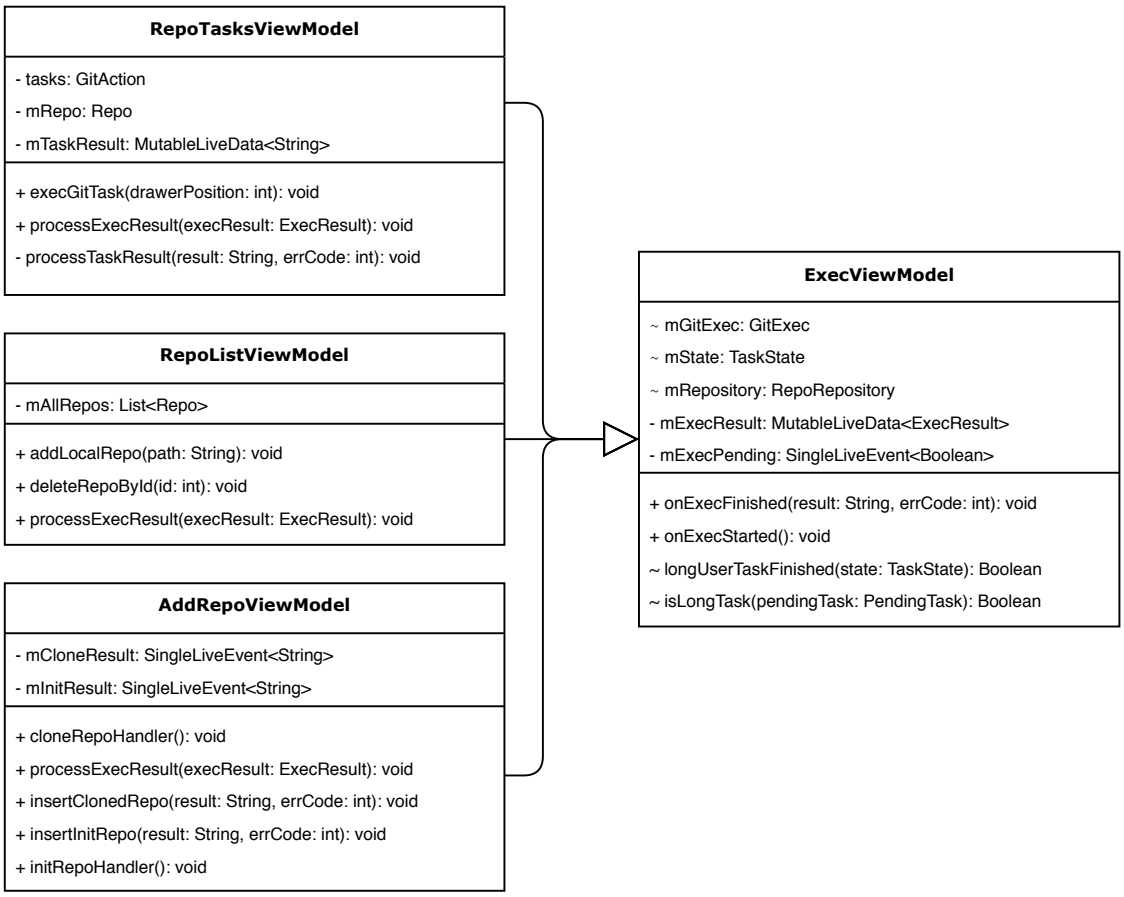
Obrázek 4.3: Struktura databáze [1]

[8] https://github.com/maks/MGit/blob/master/app/src/main/java/me/sheimi/sgit/adapters/RepoOperationsAdapter.java

[9] https://developer.android.com/topic/libraries/architecture/room

view_models

Třídy obsahující perzistentní data a implementující logiku nad nimi. Tento balíček odpovídá části *ViewModel MVVM* a poskytuje aplikaci metody zajišťující její funkčnost. Komunikace mezi třídami *ViewModelů* a aktivitami je zajištěna pomocí *databindingu*, *observerů* a veřejných metod, které tyto třídy poskytují.



Obrázek 4.4: Diagram závislostí tříd *ViewModelů*. Pro přehlednost byl zahrnut jen výběr stěžejních atributů a metod těchto tříd.

fragments

Fragmenty jsou třídy které dynamicky rozšiřují nebo mění obsah aktivity. Aplikace je používá k instalaci - `InstallFragment`, nastavení - `SettingsFragment` a zobrazování dialogů pro získání vstupu od uživatele. Každá aktivita může obsahovat několik fragmentů, které samostatně řeší určitou část aktivity. Kód třídy `InstallFragment` byl inspirován článkem z *Android Research Blog* [10].

install

O instalaci binárních souborů se stará třída `InstallTask` [11] v balíčku `install`. Ta při prvním spuštění aplikace pomocí `AsyncTask` [12] zkopíruje potřebné soubory ze složky `cpp` do interní paměti zařízení.

executors

Základní příkazy *Gitu* i jeho rozšíření jsou implementovány v balíčku `executors`. Tyto jednotlivé příkazy implementují metody třídy `GitExec`. Tyto metody volají metodu `run` třídy `BinaryExecutor`.

utilities

Jedná se o třídy, metody a proměnné, které je možné použít kdekoliv v aplikaci. Součástí jsou také třídy `TaskState` pro definování stavu zpracovávaného příkazu *Gitu* a `UriHelper` [13], sloužící pro získání *URI* k přístupu k paměti zařízení.

[10] https://androidresearch.wordpress.com/2013/05/10/dealing-with-asynctask-and-screen-orientation/

[11] https://github.com/termux/termux-app/blob/master/app/src/main/java/com/termux/app/TermuxInstaller.java

[12] https://developer.android.com/reference/android/os/AsyncTask

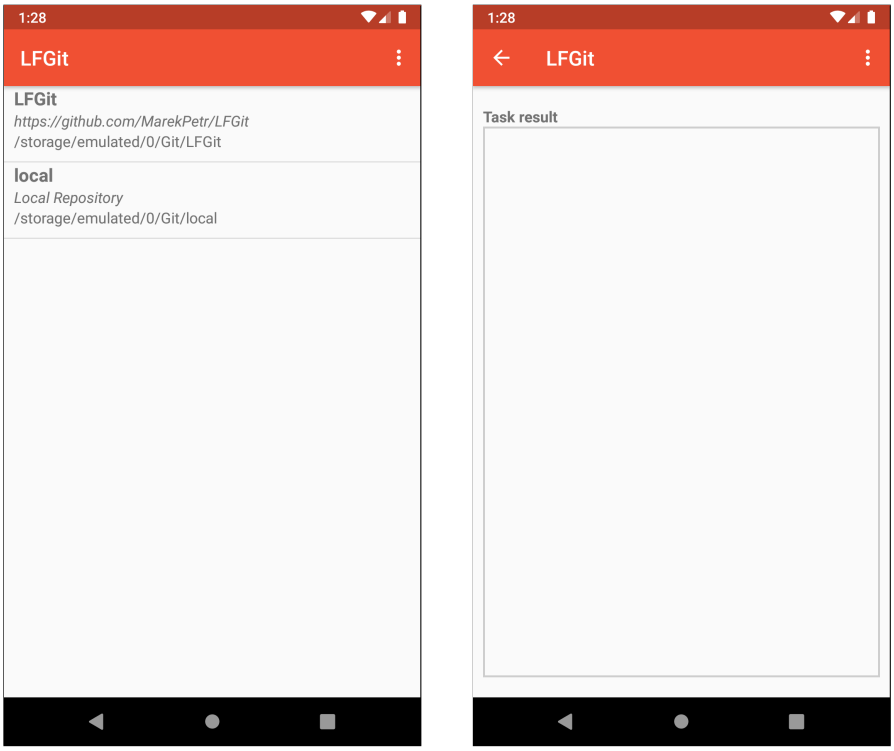
[13] https://gist.github.com/asifmujteba/d89ba9074bc941de1eaa#file-asfurihelper

4.4 Grafické uživatelské rozhraní

Významnou součástí řešení mobilní aplikace je i její uživatelské rozhraní. To bylo navrženo s důrazem na užití *Material Designu* [14]. Uživatelé *Android* jsou na něj zvyklí z většiny populárních aplikací. Orientace v něm je tedy pro ně bezproblémová.

Grafické rozhraní se nejvíce inspiruje aplikací *MGit* a přidává prvky vzniklé z požadavků na aplikaci. Především se jedná o přidání příkazů rozšíření a změnu rozhraní pro práci s repozitářem. Uživateli bude po volání příkazů *Gitu* sdělen přesný textový výstup, který mu poslouží pro další práci s *Gitem*.

Nejprve byly na papír navrženy velice jednoduché wireframy. Ty slouží pro ujasnění obsahu nejdůležitějších obrazovek. Postupně byly testovány a přepracovávány tak, aby poskytly přívětivé ovládání aplikace. Poté byly tyto výsledné obrazovky naprogramovány přímo v prostředí *Android studia* a užity pro první prototypy aplikace. Ukázka takto získaných obrazovek je vidět na obrázku 4.5.



(a) Seznam repozitářů (b) Otevřený repozitář

Obrázek 4.5: Základní obrazovky

Snímek 4.5a zobrazuje úvodní obrazovku aplikace. Nachází se na ní seznam repozitářů. U každého z nich jsou uvedeny jeho detaily. Jedná se o název repozitáře, jeho vzdálené umístění na serveru a místní cestu v zařízení. Pro přidání repozitářů a nastavení aplikace slouží menu v pravém horním rohu.

Po otevření repozitáře přejde aplikace na druhou obrazovku 4.5b. Tam uživatel vykoná operace nad repozitářem. Ty jsou dostupné v pravém výsuvném panelu. *Task result* pole slouží pro výpis výsledků jednotlivých příkazů *Gitu*.

[14] https://material.io/

4.5 Manipulace se soubory

Správce souborů je možné implementovat různými způsoby, s různým stupněm jeho komplexnosti. Pro otevírání a editování souborů, včetně symbolických odkazů uživatel užije externí aplikace. Je mu tak ponechána volnost při volbě tohoto správce a předejde se hledání kompromisů pro jeho implementaci. Navíc aplikace získá větší prostor pro ostatní funkce a uživatelské rozhraní se zjednoduší. Nevýhodou může být chybějící přehledný výběr souborů pro funkce, které pracují s jednotlivými soubory. Ale i s tím lze v aplikaci pracovat využitím *Storage Access Frameworku* [15].

4.6 Možné způsoby integrace binárních souborů

Jak již bylo zmíněno, aplikace pro příkazy *Gitu* využívá binárních souborů. Ty je samozřejmě nejprve nutné do prostoru aplikace nějakým způsobem přenést. Existují zde dvě možnosti. První je využití nativní knihovny o jejíž přenos a spouštění se postará systém *Android*. Druhá možnost je tyto operace provádět v rámci aplikace po instalaci balíčku.

4.6.1 Nativní knihovny

Z implementačního pohledu jednodušší je užití binární knihovny s příponou *.so*. Tuto knihovnu je jen třeba v rámci struktury aplikace umístit do správného adresáře. Systém si s její instalací poradí během samotné instalace aplikace. Tato metoda je dobře aplikovatelná v případě, že jsou k dispozici staticky linkované binární soubory se strojovým výstupem. Z nich je pak snadné za použití *Android NDK* [16] a *JNI* [17] vytvořit funkce, které lze používat přímo v kódu a získávat tak z těchto knihoven jejich výstup. Staticky linkované binární soubory v sobě obsahují všechny potřebné závislosti a jejich použití je tak možné samostatně. Lze jim tedy jednoduše přiřadit potřebnou příponu a budou zcela funkční. Získat tyto soubory je možné například křížovou kompilací daného programu. Staticky linkovaný *Git* je možné zkompilovat využitím již existujících nástrojů [18]. Problém s tímto způsobem tkví v tom, že takto získaný binární soubor nelze jednoduše modifikovat. Je nutné ho pokaždé znovu zkompilovat, což je časově velice náročné. Navíc tyto binární soubory musí obsahovat veškeré knihovny, které pro svůj běh potřebují. Tedy při použití více těchto binárních souborů dochází k jejich redundanci a nabývání na celkové velikosti.

[15] https://developer.android.com/guide/topics/providers/document-provider
[16] https://developer.android.com/ndk
[17] https://developer.android.com/training/articles/perf-jni
[18] https://github.com/EXALAB/git-static

4.6.2 Vlastní binární soubory

Druhá možnost je využít binárních souborů dynamicky linkovaných a jejich spouštění implementovat v rámci aplikace. Tyto binární soubory nemají jejich závislosti obsažené přímo v nich samých, ale hledají je v daných umístěních. Tím dochází k úspoře místa. Také jsou jednoduše rozšiřitelné. Tento způsob řešení není pro zařízení *Android* zcela běžný a přináší tak další řadu problémů. Předně je nutné mít je zkompilované pro pevně danou cestu a správně nastavovat systémové proměnné. Dále tyto soubory nelze spouštět přes rozhraní *JNI*. Pro spouštění se využívá metod tříd, které vytváří vlastní proces na základě dodaných parametrů. Těmi jsou například metoda `exec` třídy `Runtime` [19] nebo metoda `command` třídy `ProcessBuilder` [20]. Toto spouštění i následné čtení výsledků je nutné provádět na samostatném vlákně a implementace tak není zcela triviální.

4.7 Návrh integrace binárních souborů

Pro tuto aplikaci bylo použito dynamicky linkovaných binárních souborů s vlastní instalací i spouštěním. Lépe se s těmito soubory pracuje a aplikace tak bude lépe do budoucna rozšiřitelná. Navíc je velké množství dynamicky linkovaných programů již připraveno pro kompilaci prostřednictvím *Termux-packages* [21]. Pro samotné spouštění bylo využito třídy `ProcessBuilder`. Je snadno použitelná a umožňuje intuitivní nastavování různých parametrů běhu procesu.

4.8 Aplikační binární rozhraní - ABI

Většina fyzických zařízení používá architekturu *arm* [22]. Pro ladění aplikací se používá *Android* emulátorů a ty naopak pro nejlepší výkon používají architekturu *x86*. Pro vydání aplikace na *Google Play* je nutné, aby aplikace podporovala 32-bitové i 64-bitové verze dané architektury. Proto aplikace podporuje obě architektury pro obě verze. Jelikož by takto vytvořený *APK* balíček byl příliš velký, aplikace používá pro distribuci formát *Android App Bundle* [23].

[19] https://developer.android.com/reference/java/lang/Runtime
[20] https://developer.android.com/reference/java/lang/ProcessBuilder
[21] https://github.com/termux/termux-packages/
[22] https://handstandsam.com/2016/01/28/determining-supported-processor-types-abis-for-an-android-device/
[23] https://developer.android.com/guide/app-bundle

Kapitola 5

Implementace

Po návrhu řešení aplikace následuje implementační část. V předchozí kapitole byly popsány základní části aplikace, využité nástroje a architektura aplikace. Tato kapitola pojednává o vývoji aplikace a problémech, které bylo třeba řešit.

5.1 Získání spustitelných binárních souborů

Následující text rozebírá postup získání binárních souborů pro účely příkazů *Gitu*, způsob jejich instalace a spouštění.

5.1.1 Kompilace binárních souborů

Jak již bylo zmíněno při návrhu v kapitole 4.2, binární soubory jsou zkompilovány využitím repozitáře *Termux-packages* [1]. Ten pro tento účel poskytuje obraz *Dockeru*. Pro jeho použití pro jinou aplikaci je nutné upravit skript `scripts/build/termux_step_setup_variables.sh` tak, aby cesta ke spustitelným souborům odpovídala cílové aplikaci. Při kompilaci pro tuto aplikaci byla nastavena cesta `TERMUX_PREFIX` na `/data/data/com.lfgit/files/usr`. Takto byly zkompilovány binární soubory pro *Git* i *Git LFS*. *Git Annex* touto cestou bohužel získat nelze. Jeho kompilace je velice problémová a přes veškeré úsilí se nakonec nepodařila. Další informace o provedeném postupu se nachází v kapitole 5.3.4.

5.1.2 Instalace binárních souborů

Pro instalaci binárních souborů bylo využito třídy `TermuxInstaller` aplikace Termux [2]. Ta řeší podobný problém při instalaci linuxového prostředí. Využití části metody `setupIfNeeded` tak vyřešilo problémy s instalací symbolických odkazů do aplikace. Běžné kopírování souborů, jehož metody jsou popsány například zde [3] totiž kopírují samotné soubory, na které tyto symbolické odkazy ukazují a tím dochází k jejich redundanci a nabývání velikosti instalace. Tato část byla implementována třídou `InstallTask` v balíčku `install`.

Tento postup instalace vyžaduje užití *Android NDK* [4] pro získání zdroje dat ze zkomprimovaného souboru ve formátu *ZIP*. Ty jsou využity jak pro snížení velikosti, tak pro snadnou implementaci načtení a přenosu souborů.

[1] https://github.com/termux/termux-packages/

[2] https://github.com/termux/termux-app/blob/master/app/src/main/java/com/termux/app/TermuxInstaller.java

[3] https://www.baeldung.com/java-copy-file

[4] https://developer.android.com/ndk/

Po kompilaci byly smazány některé nepotřebné soubory zvětšující velikost instalace. Dále bylo třeba vygenerovat seznam symbolických odkazů pro všechny architektury. Ten byl vygenerován příkazem `find . -type l -ls > SYMLINKS.txt`. Poté byl tento seznam upraven tak, aby odpovídal formátu `symlink` → `file`. Třída `installTask` byla upravena tak, aby s tímto formátem pracovala. Soubor `SYMLINKS.txt` byl dále připojen ke složce s binárními soubory dané architektury. Celá tato složka byla zkomprimována a archiv přesunut do složky `cpp`. Z těchto archivů se poté během instalace aplikace, za pomocí *Android NDK*, kopírují soubory do zařízení.

5.1.3 Spouštění binárních souborů

Jak již bylo zmíněno v kapitole 4.6.2, aplikace využívá pro spouštění spustitelných souborů `Process Builder`. Ten je implementován metodou `run` třídy `BinaryExecutor`. `Process Builder` se sice postará o vytvoření nového procesu, ale pokud je po jeho ukončení očekáván jeho výstup, nemůže se samozřejmě běh aplikace během čekání na něj blokovat. Proto se tyto příkazy spouští na samostatném vlákně a výsledek je předáván prostřednictvím příslušného *callbacku*, tedy zpětného volání. Toto zpětné volání je implementováno rozhraním `ExecListener` třídy `executors`. Toto rozhraní poté implementuje třída, která očekává výsledek daného volání.

5.2 Aplikace

Po implementaci spouštění binárních souborů přichází na řadu implementace samotné aplikace. Základní popis implementace stěžejních částí je rozdělen podle jednotlivých obrazovek aplikace.

5.2.1 Seznam repozitářů

Uvítací obrazovka je implementována aktivitou `RepoListActivity`. Ta při prvním spuštění nainstaluje potřebné binární soubory a dále zobrazí prázdný seznam sledovaných repozitářů. Tento seznam zobrazuje aktuální stav databáze repozitářů. Repozitář uživatel do seznamu přidá prostřednictvím menu. Vybraná akce spustí daný *Intent* a přesune uživatele na další obrazovku. Mezi tyto akce patří i přidání repozitáře. Přidané repozitáře jsou také synchronizovány s úložištěm. V případě smazání z úložiště dojde při dalším načtení tohoto seznamu k jeho smazání ze seznamu a tedy i databáze. Obnovení seznamu lze provést i okamžitě gestem táhnutí shora dolů. V případě smazání repozitáře v době jeho otevření, bude uživatel na toto upozorněn při provádění příkazů *Gitu* a navrácen zpět do seznamu repozitářů.

5.2.2 Přidání repozitáře

Repozitář lze do seznamu přidat třemi způsoby. Pokud již existuje v paměti zařízení, lze ho přidat tlačítkem menu `Add repository`. Tato akce spustí *Intent* `ACTION_OPEN_DOCUMENT_TREE` [5] pro vybrání složky s repozitářem. Dále je možné repozitář inicializovat a klonovat. Obrazovka s klonování a inicializací je implementována aktivitou `AddRepoActivity`. Také je podporováno mělké klonování pro zadaný počet *commitů*, tedy hloubku.

[5] https://developer.android.com/reference/android/content/Intent#ACTION_OPEN_DOCUMENT_TREE

5.2.3 Provedení příkazů *Gitu*

Po kliknutí na repozitář dojde k jeho otevření. V pravém bočním panelu se nachází seznam všech podporovaných příkazů. Ten je implementován třídou `RepoTasksAdapter`. Při kliknutí na jeho položku (příkaz) dojde k zavolání metody `execGitTask RepoTasksViewModelu`. Ta z pole příkazů implementovaného rozhraním `GitAction` vybere podle polohy v panelu ten správný a vykoná ho. V případě, že daný příkaz potřebuje další parametry, jsou před jeho vykonáním tyto parametry vyžádány od uživatele.

5.3 Problémy objevené při implementaci

Implementace provázelo velké množství překážek. Ty se začaly objevovat již při procesu získání binárních souborů a jejich řešení nebylo vždy snadné. Následující text se bude věnovat nejzajímavějším problémům, které nejsou při vývoji zcela běžné.

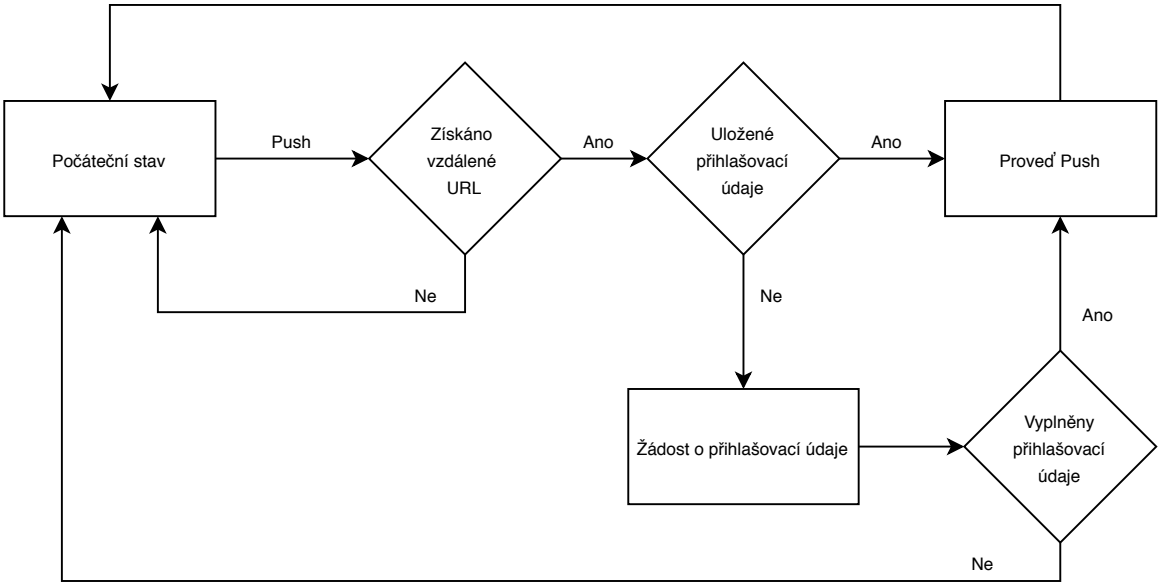
5.3.1 Příkazy Gitu

Logika příkazů *Gitu* je implementována ve *View Modelech* daných aktivit. Jelikož všechny volané funkce vrací hodnotu zpětným voláním, bylo nutné pro získání všech vstupů pro vykonání daného příkazu definovat stavy jeho zpracování. K tomu slouží třída `TaskState`. Obsahuje atribut `mInnerState` určující aktuální stav rozpracovaného příkazu a atribut `mPendingTask` definující zpracovávaný příkaz. Existují zde dva speciální stavy atributu `mInnerState` `FOR_APP` a `FOR_USER`. Tyto stavy určují, jestli bude výsledek následujícího příkazu zobrazen uživateli, nebo bude využit jako vstupní argument pro jiný účel aplikace.

Push

Nejsložitější příkaz k implementaci je příkaz *push*. Ten pro přístup do vzdáleného repozitáře potřebuje jak jeho *URL*, tak přihlašovací údaje uživatele. Ty jsou od uživatele žádány pro každý nově přidaný repozitář pouze při prvním vykonávání tohoto příkazu. Po jejich zadání jsou uloženy v databázi v tabulce daného repozitáře. Databáze je uložena ve vnitřním prostoru aplikace, kam má přístup jen ona samotná. Uložení těchto citlivých údajů je tedy relativně bezpečné. Před provedením příkazu jsou znaky uživatelského jména a hesla kódovány do *MIME* formátu *application/x-www-form-urlencoded* [6].
Samotné provedení pak probíhá prostřednictvím *http* nebo *https* protokolu a to příkazem `git push http(s)://username:password@domain`. Tento způsob aplikace používá z důvodu nemožnosti zadávat vstupní data binárním souborům během jejich běhu. *ProcessBuilder* sice umožňuje před spuštěním programu zadat vstupní data, ale není možné v rámci běhu reagovat na čtení vstupů z příkazové řádky tak jako při využití terminálového rozhraní.

[6] https://developer.android.com/reference/java/net/URLEncoder



Obrázek 5.1: Zjednodušený diagram provedení příkazu push.

5.3.2 Správce souborů

Implementace užití externího správce souborů se ukázala jako více problematická, než se při návrhu předpokládalo. Původní návrh této funkcionality spočíval v předání *URI* externí aplikaci správce souborů, kterou uživatel používá jako výchozí. Toho nebylo možné dosáhnout z důvodu použití rozdílných typů *URI* existujících aplikací správce souborů. Pro jeho zahrnutí do aplikace by tak bylo nutné implementovat vlastní správce souborů. Toho se aplikace snažila z mnoha důvodů již od návrhu vyhnout. Zejména proto, že různí uživatelé budou mít na tento správce různé požadavky a pro implementaci tak komplexního správce souborů, který by uživatele přesvědčil o jeho používání namísto jiné aplikace, nebyl prostor. Správce souborů tedy nakonec implementován nebyl a aplikace slouží čistě jako *Git* klient. Uživatel tak bude přecházet z jedné aplikace do druhé, obdobně jako při práci na PC. To pro pokročilého uživatele systému *Android* a *Git*, na kterého tato aplikace cílí, nepředstavuje překážku. Toto rozhodnutí je podpořeno i faktem, že samotné užití programu *Git* neřeší žádný komplexní problém uživatele. Ten proto pravděpodobně bude používat řadu dalších aplikací, které v kooperaci splní jeho záměr.

5.3.3 Git LFS

S implementací tohoto rozšíření *Gitu* se pojí dva problémy. První se týká samotné integrace do prostředí systému *Android*, druhý pak mechanismu *Git Hooks* [7].

Hooks, neboli háček, *Gitu* umožňuje před nebo po vykonání jeho příkazu spustit z definovaného adresáře daný skript. *Git LFS* tohoto mechanismu využívá pro synchronizaci repozitáře.

Integrace do aplikace

Integraci tohoto rozšíření také provázely problémy, ale všechny byly řešitelné. Neobvyklý problém nastal při prvních pokusech o přidání sledovaného souboru do *stage*. *Git LFS* při něm zobrazoval chybou hlášku, že není možné vytvořit proces pro spuštění jeho programu `filter-process`. Pro zjištění možné příčiny byl příkaz `git add .` spuštěn prostřednictvím ladícího programu *strace* [8]. Pro jeho užití bylo nutné výstup přesměrovávat do souboru. `StringBuilder`, který je využit pro zpracování výstupu z binárních programů již tak rozsáhlý výstup nezvládal zpracovávat a vracel jen jeho část. Ovšem i po vyřešení této chyby *strace* konkrétní chybu neodhalil.

Pro pomoc při řešení byla vytvořena tzv. *issue* v oficiálním repozitáři *Git LFS* [9]. S pomocí jednoho z hlavních vývojářů tohoto programu byla příčina tohoto problému odhalena. Využitím přepínače `strace '-f'` bylo zjištěno, že *Git LFS* nenalezl interpret *shell*. Problém byl vyřešen přidáním symbolického odkazu na `/system/shell` do složky `bin`, kde byl tento interpret očekáván.

Git LFS hooks

Integrací problémy s *Git LFS* stále nekončily. Při vykonávání kteréhokoliv příkazu, který spouštěl tzv. *hook*, *Git* vracel chybu pro nedostatečná práva pro otevření těchto souborů.

Bylo to dáno tím, že příkaz pro instalaci *Git LFS* do repozitáře, vytvářel *hooky* uvnitř tohoto repozitáře, ale nepřiřadil jim patřičná práva. Řešení byla možná dvě. Buď po proběhnutí každé instalace nastavit práva těmto souborům ručně, nebo tyto *hooky* vytvářet ve vnitřním prostoru aplikace, kde jsou již potřebná práva nastavena.

Na první pohled by se mohlo zdát, že uživatelsky přívětivější je první řešení. Uživatel *hooky* vytvoří jednou při instalaci *Git LFS* a při klonování repozitáře na jiném zařízení, již budou nainstalovány. Bohužel situace taková není. *Hooky* jsou při výchozím nastavení umístěny lokálně ve složce `.git` repozitáře a při klonování zkopírovány nejsou [10].

Z tohoto důvodu byla vybrána druhá varianta. *Hooky* se vytváří automaticky ve vnitřním prostoru aplikace při její instalaci. Uživatel je navíc nemusí instalovat manuálně v každém repozitáři zvlášť a to zjednoduší jeho práci.

[7] https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks
[8] https://strace.io/
[9] https://github.com/git-lfs/git-lfs
[10] https://www.atlassian.com/git/tutorials/git-hooks

5.3.4 Git annex

Získávání spustitelných souborů pro *Git Annex* se ukázalo jako velice problematické. Následující text je shrnutím mnoha provedených pokusů, z nichž některé byly velice blízko k získání těchto souborů.

Oficiální distribuce pro Termux

Nejprve byly staženy instalační soubory *Git Annex* z jeho oficiální webové stránky [11]. Tyto soubory byly nainstalovány do vnitřního prostoru aplikace s příslušnými právy pro spuštění. Při prvních pokusech byly zjištěny problémy, které se týkaly chybějících programů interpretu *shell*. Pro jejich vyřešení byl stažen program *Busybox* [12] pro architekturu testovacího zařízení. Ten problém nevyřešil, jelikož z neznámého důvodu nebylo možné ho spustit. Proto byly potřebné skripty přepsány do podoby, kterou *shell* zařízení byl schopný zpracovat . Později se zjistilo, že používaná verze *Busyboxu* byla vadná a problém vyřešila jiná verze, ale tento program již nebyl potřeba. Přes přepsání všech inkriminovaných příkazů skriptů *Git Annexu* se nedařilo tento program spustit. Užitím programu *strace* bylo zjištěno, že bezpečnostní zabezpečení systému *Android* - tzv. *Seccomp* filtrování zabraňuje spuštění určitého kódu spojeného se spouštěním *Git Annexu*. Tento problém již řeší verze *Git Annexu* určená pro *Termux*. Po jejím prozkoumání bylo zjištěno, že řešení leží v nástroji *Proot*, který toto filtrování umí obejít. Bohužel ani po mnoha pokusech spouštění *Git Annexu* tímto nástrojem nebylo dosaženo úspěchu.

Vlastní kompilace Git Annex

Dále následovalo mnoho pokusů o vlastní křížovou kompilaci *Git Annexu*. Jeho kód je psán v jazyce *Haskell* a tato kompilace je tím značně zkomplikována. Byly provedeny pokusy o kompilace pomocí nástroje *Nix* [13], *GHC Android* [14] a dalších. Některé pokusy selhaly již při kompilaci křížového kompilátoru, jiné až při kompilaci *Git Annexu*.

Vlastní kompilace nástroje Proot

Jelikož nebylo skrze křížovou kompilaci dosaženo většího pokroku než užitím verze určené pro *Termux*, další pokusy pokračovaly právě s ní. Na pomoc s problémem s *Proot* byla kontaktována jeho komunita na komunikačním kanále služby *Gitter* [15]. S pomocí těchto vývojářů se podařilo zkompilovat verzi *Prootu*, kterou používá *Termux* pro prostředí balíčku vyvíjené aplikace. To vyřešilo problém s *Seccomp* filtrováním, ale objevil se další. *Git annex* stále nebylo možné spustit. Problém se týkal chybějících programů interpretu. *Termux* pro běh programů vytváří vlastní linuxový systém, kde jsou nainstalovány všechny standardní programy. Toto prostředí mu tato aplikace neposkytuje. Tento problém byl dříve 5.3.4 pro funkci skriptů obejit jejich přepsáním. Pro binární soubory toto bohužel možné není. Implementace vlastního systému uvnitř aplikace je velice problémová a také náročná na úložiště. Tento problém nakonec nebyl vyřešen. Zbývající zdroje byly investovány do vývoje kvalitní aplikace bez podpory *Git Annexu* s možností rozšíření po jejím vydání, v případě nalezení řešení.

[11] https://git-annex.branchable.com/Android/
[12] https://busybox.net/
[13] https://github.com/pololu/nixcrpkgs
[14] https://medium.com/@zw3rk/a-haskell-cross-compiler-for-android-8e297cb74e8a
[15] https://gitter.im

Kapitola 6

Testování

Testování aplikace probíhalo manuálně a to ve třech fázích. První se soustředila na otestování příkazů *Gitu*, druhá na správu sledovaných repozitářů a třetí na celkové uživatelské rozhraní.

6.1 Příkazy Gitu

První fáze testování byla zaměřena na správné provedení příkazů *Gitu*. Při tomto testování byly nad repozitáři spouštěny různé příkazy a jejich výsledek byl konfrontován se stavem repozitáře. Aktuální stav repozitáře se ověřoval využitím aplikace *Termux*, zpětnou kontrolou na serveru *GitHub* i klonováním repozitáře do jiných zařízení, při použití této aplikace. Tím bylo dosaženo zajištění správné funkčnosti těchto příkazů.

Testována byla také schopnost provedení základního vytvoření repozitáře a jeho `push` na nově vzniklý prázdný vzdálený repozitář. Při tomto kroku byl opravován zejména samotný příkaz `push`. Ten byl testován na různé formy *URL*, uživatelského jména i hesla. Přidána byla kontrola na zadání *http(s)* adresy, jelikož i autentizace probíhala touto cestou. Uživatelská jména a hesla byla pro ověření správného *URL* kódování testována na speciální znaky.

Opravy se dotkly i rozšíření *Git LFS*. Během testování bylo využito například příkazů *git-lfs env* nebo *git-lfs status*. Tyto příkazy byly poté pro jejich užitečnost přidány do příkazů aplikace.

Git LFS také zpočátku neposílalo obsah sledovaných souborů na server, ale pouze jejich odkaz. Ten byl při stažení na jiná zařízení neplatný. Tento problém se týkal *Git LFS hooks* a byl již popsán v kapitole 5.3.3.

6.2 Správa repozitářů

Tato fáze testování se zaměřovala na práci se seznamem repozitářů. Testováno bylo validní přidání repozitáře, jeho odebrání i aktualizace jeho informací. Byly odhaleny a odstraněny chyby související se synchronizací odstranění repozitáře během jeho ovládání. Z informací o repozitáři byla testována zejména aktualizace *URL* vzdáleného repozitáře při jeho změně.

6.3 Uživatelské rozhraní

Otestováno bylo i uživatelské rozhraní. Testováno bylo přerušení započatých sekvencí, které předchází různým příkazům tak, aby se tyto prvky chovaly validně. Testování mimo jiné zahrnovalo i kontroly správného stavu zobrazení různých formulářů i po změně rozložení displeje nebo jejich skrytí na pokyn aplikace.

Upravovány byly i velikosti písma a některých jiných grafických prvků. Změnami také prošla obrazovka inicializace a klonování repozitáře. Ta při vodorovném režimu skrývala tlačítko pro spuštění klonování. Proto byla do aplikace přidána speciální verze pro vodorovné rozložení, které toto tlačítko posune do viditelné zóny obrazovky.

6.4 Vydání aplikace

Po otestování byla aplikace uvedena do reálného provozu na *Google Play* pod názvem *LFGit*[1]. Jak již bylo zmíněno v kapitole 4.8, aplikace k distribuci používá balíčku *Android App Bundle* [2]. Zdrojový kód aplikace je dostupný na *GitHub* [3].

[1] https://play.google.com/store/apps/details?id=com.lfgit
[2] https://developer.android.com/guide/app-bundle
[3] https://github.com/MarekPetr/LFGit

Kapitola 7

Závěr

Cílem práce bylo vytvořit mobilní aplikaci pro zařízení systému *Android*, sloužící k ovládání *Git* repozitářů. Tato práce popisuje postup jejího vývoje od návrhu po implementaci a uvedení do reálného provozu.

Aplikace umožňuje uživateli spravovat *Git* repozitáře uložené v paměti zařízení. K tomu slouží úvodní obrazovka aplikace, která zobrazuje sledované repozitáře. Uživatel otevře libovolný repozitář a provádí nad ním příkazy *Gitu* i *Git LFS*. Tyto příkazy jsou dostupné v bočním panelu aplikace. Po provedení příkazu je uživateli zobrazen jeho textový výstup.

Jelikož je *Git* velice komplexní systém, byl hlavní důraz kladen na rozšiřitelnost aplikace a transparentnost dostupných příkazů. Při vývoji bylo prováděno zejména testování funkčnosti těchto příkazů a správy repozitářů.

Oproti již existujícím *Android* aplikací, tato aplikace nabízí především podporu *Git LFS* s možností rozšíření o *Git Annex*, případně i další programy. Mezi již přidané funkcionality patří například mělké klonování nebo zobrazení prostředí *LFS env*. Dále aplikace nabízí transparentní a jednoduché uživatelské rozhraní, které uživateli zobrazuje výstup příkazů shodný s *Git* verzí na PC.

Aplikace oproti mnoha jiným aplikací této platformy neobsahuje správce souborů. Integrovaný správce souborů, který obsahují jiné aplikace, se obecně ukázal jako nedostačující a uživatel tak pro plnou správu souborů užije jiné plnohodnotné aplikace.

7.1 Zhodnocení výsledku práce

Požadavky na práci byly vyjma podpory *Git Annex* splněny. Aplikace má potenciál stát se úspěšnou *Git* aplikací systému *Android* pro práci s velkými soubory.

7.2 Pokračování ve vývoji

Další vývoj aplikace se bude odvíjet více směry. Zejména bude navázáno na postup integrace *Git Annex* s cílem nalezení možného řešení. Mezitím bude probíhat testování v reálném provozu, které odhalí reálný potenciál aplikace. Na základě této zpětné vazby budou přidávány i další příkazy *Gitu* a jiné funkce aplikace. Reálný provoz také ukáže správnost rozhodnutí neimplementace správce souborů, na jehož základě bude rozhodnuto o dalším postupu. Jako další rozšíření se nabízí i podpora autentizace k *Git* serverům pomocí *ssh* klíčů. Současně s provedenými změnami bude také přepracováváno a zdokonalováno uživatelské rozhraní.

Literatura

[1] *Android Room with a View - Java* [online]. [cit. 2020-04-26]. Dostupné z: https://codelabs.developers.google.com/codelabs/android-room-with-a-view/.

[2] Chugh, A. *Android MVVM Design Pattern* [online]. [cit. 2020-04-19]. Dostupné z: https://www.journaldev.com/20292/android-mvvm-design-pattern.

[3] contributors, W. *Git-annex* [online]. Wikipedia, The Free Encyclopedia., květen 2015 [cit. 2020-04-03]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Git-annex&oldid=915249727.

[4] GitHub, Inc. *Git Large File Storage*. [cit. 2019-04-22]. Dostupné z: https://git-lfs.github.com/.

[5] Google LLC. *Guide to app architecture* [online]. [cit. 2020-04-19]. Dostupné z: https://developer.android.com/jetpack/docs/guide.

[6] Google LLC. *LiveData Overview* [online]. [cit. 2020-04-19]. Dostupné z: https://developer.android.com/topic/libraries/architecture/livedata.

[7] Google LLC. *Understand the Activity Lifecycle* [online]. [cit. 2020-04-19]. Dostupné z: https://developer.android.com/guide/components/activities/activity-lifecycle.

[8] Google LLC. *ViewModel Overview* [online]. [cit. 2020-04-19]. Dostupné z: https://developer.android.com/topic/libraries/architecture/viewmodel.

[9] Hess, J. *Git Annex*. [cit. 2019-04-22]. Dostupné z: https://git-annex.branchable.com/.

[10] Lacko Luboslav. *Vývoj aplikací pro Android*. Computer Press (CP Books), 2015. ISBN 978-80-251-4347-6.

Příloha A

Obsah přiloženého paměťového média

- `xmarek66-BP.pdf` - tato písemná zpráva
- `source-code.zip` - zdrojové soubory aplikace