ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Android mobilní aplikace s GPS trekingem
Student:	Bc. Vojtěch Vopařil
Vedoucí:	Ing. Martin Půlpitel
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

Cílem diplomové práce je navrhnout a naprogramovat mobilní aplikaci pro cyklisty pro operační systém Android s využitím dvou programovacích paradigmat – objektově orientovaného a funkcionálně reaktivního, a oba tyto přístupy mezi sebou porovnat.

Funkční požadavky na aplikaci:

- Na mapový podklad dynamicky vykreslujte trasu na základě souřadnic z GPS.
- Jednotlivé úseky trasy vykreslujte v barvě odpovídající rychlosti pohybu v úseku.
- Zobrazujte základní informace o průběhu trasy (ujetá vzdálenost, průměrná rychlost, aktuální rychlost atd.).
- Exportujte trasy do souboru ve formátu GPX.

Nefunkční požadavky:

- Naimplementujte aplikaci ve dvou variantách – s využitím objektově orientovaného přístupu a s využitím funkcionálně reaktivního přístupu.
- Zvolte a pracujte s vhodnými knihovnami, frameworky.
- Zdokumentujte a otestujte obě implementace.
- Porovnejte obě implementace z pohledu vývoje, údržby a nasazení.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 2. ledna 2017

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

Android mobilní aplikace s GPS trekingem

Bc. Vojtěch Vopařil

Vedoucí práce: Ing. Martin Půlpitel

9. ledna 2017

Poděkování

Děkuji Ing. Martinovi Půlpitlovi za vedení mé diplomové práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Berouně dne 9. ledna 2017 .....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2017 Vojtěch Vopařil. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Vopařil, Vojtěch. *Android mobilní aplikace s GPS trekingem.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Diplomová práce se zabývá vývojem mobilní aplikace pro cyklisty pro operační systém Android. Aplikace je vyvinuta ve dvou provedeních se shodnou funkcionalitou, které se odlišují použitým programovacím paradigmatem ve zdrojovém kódu. První provedení je založené čistě na objektově orientovaném paradigmatu, druhé využívá paradigmatu funkcionálně reaktivního. Oba přístupy jsou vzájemně porovnány z hlediska vývoje, testování, nasazení a též údržby a případného budoucího rozšiřování funkcionality aplikace.

Klíčová slova funkcionální programování, reaktivní programování, FRP, programovací paradigma, mobilní aplikace, RXJava, Android, GPS trackování

Abstract

This master's thesis targets the development of a mobile application for cyclists for Android operating system. The application is developed in two implementations with identical functionality that differs with programming paradigm that is used in source code. First implementation is completely based on objected oriented paradigm, second implementation uses functional reactive paradigm. Both approaches are compared from the point of development, testing, deployment and also future possible extension of application functionality.

Keywords functional programming, reactive programming, FRP, programming paradigm, mobile application, RXJava, Android, GPS tracking

Obsah

Úvod **1**
Struktura práce 2

1 Představení platformy Android z pohledu programátora **3**
1.1 Úvod 3
1.2 Technologie a prostředky využívané pro programování aplikací 3
1.3 Schéma Android projektu 4
1.4 Specifické zdrojové komponenty 6
1.5 Perzistence dat 9
1.6 Oprávnění jako prostředek zabezpečení 11
1.7 Ladění a testování aplikace 11

2 Funkcionálně reaktivní programování **13**
2.1 Stručná historie FRP 13
2.2 Koncept FRP 13
2.3 Reaktivní manifest 14
2.4 Návrhový vzor pozorovatel 14
2.5 Základní rozdíly FRP oproti OOP 15
2.6 Komponenty FRP 16
2.7 FRP na platformě Android 18

3 Reálné setkání s FRP **25**
3.1 Úvod 25
3.2 Struktura FRP projektu 25
3.3 Příklady kategorie Stream 25
3.4 Příklady kategorie Subscription 32
3.5 Příklady kategorie Listener 32
3.6 Příklady kategorie Exception 33
3.7 Příklady kategorie RxBus 34
3.8 Shrnutí prvních zkušeností s FRP 35

4 Základní návrh aplikace **39**
4.1 Základní funkcionalita 39
4.2 Schéma aplikace 39

5 Realizace **41**
5.1 Úvod 41
5.2 Struktura zdrojových souborů 41
5.3 Mapové podklady 42
5.4 Vykreslování trasy na základě údajů z GPS 44
5.5 Funkce optimalizace trasy 46
5.6 Export trasy do souboru ve formátu GPX 47
5.7 Načtení trasy z GPX souboru a její sledování 49
5.8 Sekce statistika a práce s databází 50
5.9 Nastavení aplikace 51
5.10 Služba pro zobrazování hromadných Toastů 52
5.11 Prostor pro budoucí vylepšení aplikace 53

6 Specifika OOP a FRP verze **55**
6.1 Specifika z hlediska implementace 55
6.2 Specifika z hlediska testování 59
6.3 Specifika z hlediska ladění 60

7 Shrnutí zkušeností s FRP **61**
7.1 Vývoj, testování a ladění 61
7.2 Nasazení, údržba a budoucí rozvoj 61
7.3 Závěrečné shrnutí 62

Závěr **63**

Literatura **65**

A Požadavky a pokyny pro běh a instalaci aplikace **67**
A.1 Požadavky pro běh aplikace 67
A.2 Pokyny pro instalaci aplikace 67

B Ovládací prvky trackovací obrazovky **69**

C Screenshoty aplikace **71**

D Seznam použitých zkratek **75**

E Obsah přiloženého CD **77**

Seznam obrázků

1.1 Životní cyklus Android aktivity 7

2.1 Reaktivní manifest - vztah vlastnostmí reaktivního systému 15
2.2 Návrhový vzor pozorovatel - UML diagram 15
2.3 UML modely architektur aplikací 16
2.4 Reaktivní podoba hodnot a polí 17

5.1 Diagram základních tříd pro práci s mapami 44
5.2 Optimalizace trasy - ilustrace problematiky 46
5.3 Optimalizace trasy - úhlová situace 48
5.4 Formát GPX - waypoint, route, track 49

Úvod

V rámci této práce se zabývám implementací mobilní aplikace pro cyklisty pro operační systém Android. Téma této práce lze popsat ze dvou hlavních pohledů. Prvním pohledem je implementace mobilní aplikace pro cyklisty dle specifikovaných funkčních a nefunkčních požadavků. Druhým pohledem je porovnání funkcionálně reaktivního programování oproti běžně využívanému objektově orientovanému přístupu při vývoji mobilních aplikací pro platformu Android.

Úkolem mobilní aplikace je především poskytnout GPS trekovací funkci cyklistům. Jednotlivé údaje o poloze budou zaznamenávány a dynamicky vykreslovány v podobě křivky na mapový podklad, přičemž úseky trasy budou vykresleny v barvě odpovídající rychlosti projetí úseku. Zaznamenanou trasu bude možné exportovat do souboru ve formátu GPX.

Porovnání programovacích paradigmat bude založené na zkušenostech při dvojí implementaci aplikace se shodnou funkcionalitou, přičemž první implementace bude využívat čistě objektově orientovaný přístup, druhá bude ve vysoké míře založená na funkcionálně reaktivním programování. U obou přístupů zhodnotím rozdíly, přínosy a zápory při vývoji, testování nasazení a též údržbě a případném budoucím rozšiřování funkcionality aplikace.

Struktura práce

Tato práce je rozdělena celkem do sedmi kapitol:

První tři kapitoly se věnují systému Android a funkcionálně reaktivnímu programování. První kapitola se věnuje představení platformy Android z pohledu programátora, druhá úvodu do funkcionálně reaktivního programování a třetí praktickému vyzkoušení funkcionálně reaktivního programování na příkladech.

Čtvrtá až šestá se pak již týkají realizovaných aplikací. Čtvrtá se věnuje základnímu návrhu aplikace, pátá realizaci aplikace z hlediska popisu realizované funkcionality a implementačních detailů pro shodné části obou verzí aplikace a šestá odlišnostem mezi objektově orientovanou a funkcionálně reaktivní verzí aplikace.

Poslední sedmá kapitola je věnována shrnutí zkušeností s funkcionálně reaktivním programováním.

Kapitola 1

Představení platformy Android z pohledu programátora

1.1 Úvod

Operační systém Android od společnosti Google má dnes za sebou již více než 10 let vývoje, přičemž první zařízení s tímto systémem vstoupilo na trh v říjnu roku 2008. Již první vydaná verze byla na velmi vysoké úrovni a poskytovala širokou podporu pro práci například s webovým obsahem, multimédii či integrovanými senzory v zařízení. Každá další pak přinesla podporu nejen další funkcionality, ale také nových aplikačních rozhraní a frameworků pro usnadnění práce programátorům. Díky pozici tohoto operačního systému na trhu a s tím spojené široké programátorské komunitě je pak možné i nejsoučasnější a prozatím chybějící technologii doplnit pomocí některých z velkého množství dostupných knihoven a pluginů. Android v nejnovějších verzích tak lze považovat za velmi moderní platformu jak z pohledu uživatele, tak z pohledu programátora.

1.2 Technologie a prostředky využívané pro programování aplikací

1.2.1 Programovací jazyk

Aplikace pro systém Android se programují v programovacím jazyce Java. Aktuální vývojové nástroje poskytují dlouhodobě podporu pro verzi 7 jazyka. Volitelně lze využít poměrně nového kompilátoru Jack a linkeru Jill, který u aplikací určených pro nejnovější verze systému Android (od 7.0 výše) podporuje většinu přidaných schopností v Javě 8. Aktuální verze těchto nástrojů, respektive jejich integrace, jsou ovšem stále nevyladěné a problémové, jak jsem měl ostatně možnost i sám vyzkoušet. Dnes je proto pro doplnění poža-

dovaných schopností vhodnější využít podpůrných doplňků a knihoven třetích stran.

1.2.2 Vývojová prostředí

Prvním oficiálním vývojovým prostředím (IDE) pro vývoj aplikací pro Android bylo Eclipse s ADT Pluginem. V roce 2013 vydal Google vlastní prostředí Android Studio, postavené na základu IntelliJ IDEA, které roli oficiálního vývojového prostředí převzalo. Kromě Eclipse a Android Studio/IntelliJ IDEA, lze s pluginem NBAndroid programovat pro Android i v NetBeans, jde však o neoficiální plugin.

1.2.3 Emulátor Androidu

Emulátory Androidu pro desktopové operační systémy jsou součástí základních vývojových nástrojů pro Android, dostupné ke stažení a instalaci z Android SDK Manageru. Google vydává emulátor ke každé vydané verzi systému Android.

1.2.4 Uživatelské rozhraní a grafika

Vývojová prostředí nabízí grafický editor, potažmo náhled, podoby uživatelského rozhraní definovaného v souboru XML. V závislosti na vývojovém prostředí či pluginu a jejich verzích je pak dále poskytována širší podpora pro úpravu uživatelského rozhraní. Speciálním případem je tvorba ikon, které pracují s definovanou průhledností a roztažením. Ty lze vytvářet ze souborů PNG pomocí nástroje Draw 9-patch, který je součástí základního balíku Android nástrojů. Podpora pro generování 9-patch PNG souborů je dnes přítomna minimálně také v novějších verzích Android Studia.

1.3 Schéma Android projektu

Následující popis odpovídá schématu projektu v Android Studiu, v ostatních prostředích se může mírně lišit. Projekt mobilní aplikace se sestává ze čtyř hlavních součástí, a to manifestu, javových zdrojových souborů, ostatních zdrojových souborů a sestavovacích skriptů.

Manifest obsahuje informace o aplikaci, které jsou předávány systému Android před spuštěním aplikace. Obsahuje název balíčku aplikace, seznam oprávnění, která aplikace vyžaduje, název aplikace, odkaz na soubor ikony spouštěče aplikace, seznam takzvaných aktivit, což jsou základní stavební prvky každé aplikace v podobě javových tříd, události, které je aplikace schopná přijímat, informace o službách běžících na pozadí, seznam potřebných knihoven, různá metadata a další. Podrobné informace o manifestu lze najít v oficiální online dokumentaci [1].

Javové zdrojové soubory se dělí dle obecných zvyklostí na zdrojové soubory aplikace, přičemž některé javové třídy na bázi systémových tříd Androidu popíšu v dalších podkapitolách, a zdrojové soubory testů. Odlišností Androidu od běžného javovského projektu je rozdělení testovacích souborů na běžné testy a Android testy. Běžné testy jsou typicky jednotkové testy pro jednotlivé třídy, které je možné spouštět z konzole přes virtuální stroj Javy. Pro běh Android testů je pak třeba sestavit celou aplikaci a spustit ji v emulátoru, typicky jde o testy integrační (testy vzájemné komunikace komponent) a funkční (testy funkčnosti uživatelského rozhraní).

Ostatní zdrojové soubory označované jako `resources` (v projektu podadresář `res`) obsahují soubory různých typů dělených do několika skupin:

drawable Slouží pro uložení bitmapových souborů a speciálních grafických souborů ve formátu XML.

layout Obsahuje XML soubory s informacemi o grafických prvcích a jejich rozložení na obrazovce.

menu Slouží pro definici menu, ale jde o funkcionalitu využívanou především ve starších verzích Androidu a ve výchozím stavu nového projektu tak dnes tuto skupinu nenajdeme.

mipmap Sekvence grafických souborů uložených v různých rozlišeních. Ve většině projektů je zde uložena alespoň sekvence souborů ikony spouštěče aplikace.

raw Obsahuje soubory, které nelze zařadit do jiných skupin

values Slouží k uložení hodnotových prvků jako jsou barvy, rozměry či textové řetězce. Uložen je zde i soubor se styly pro úpravu vzhledu.

xml Rozličné konfigurační soubory uložené ve formátu XML. Jde například o soubor popisující prvky na obrazovce nastavení aplikace.

Některé skupiny mohou dále obsahovat kvalifikátory označené podskupiny, kdy v nich uložené soubory jsou použity místo odpovídajících souborů v hlavní skupině za určitých podmínek, respektive stavů. Jde například o specifickou orientaci a rozlišení displeje či hustotu pixelů, verzi API přístroje a zmínit je důležité i kvalifikátory označující konkrétní světový jazyk u skupiny `values`, které umožňují snadný překlad aplikace. Veškeré napevno uložené textové řetězce zobrazované uživateli by z tohoto důvodu měly být uloženy právě v souborech ve skupině `values`.

V sestavování aplikace pomáhá nástroj Gradle. Některé informace lze místo Android manifestu uvést právě v sestavovacích skriptech, typicky jde třeba o informace o minimální a cílové verzi API či o výpis potřebných knihoven. V případě, že je stejný druh informace obsažen v obou souborech, je platný

ten v sestavovacím skriptu. Sestavovací skripty jsou ve výchozím stavu dva - jeden hlavní a jeden pro vyvíjenou aplikační část, označenou jako „modul“. Případně do projektu zahrnuté lokální knihovny jsou pak dalšími „moduly“ v projektu. Do procesu sestavování aplikace lze pak pomocí nastavení v Gradle skriptu zapojit také připravený nástroj ProGuard pro optimalizaci, redukci a obfuskaci kódu [1].

1.4 Specifické zdrojové komponenty

1.4.1 Android aktivita

Android aktivity, respektive instance tříd dědících od javovské třídy (`Activity`) systému Android, tvoří základní funkční prvky aplikace. Aktivity jsou typicky zaměřené na určitou činnost a zpravidla nabízejí určitou interakci s uživatelem, zobrazení nějaké obrazovky. Každá aktivita musí být uvedena v manifestu. Ve starších verzích Androidu bylo možné aktivity vnitřně členit vnořováním do sebe, dnes jde o zastaralý přístup a pro vnitřní členění aktivity se užívá fragmentů (systémová třída `Fragment`). Android aktivity se spouštějí navzájem pomocí intentů (`Intent`), sloužících k popisu prováděné operace. Součástí intentu mohou být data předávaná mezi aktivitami.

Aktivity mají definovaný životní cyklus, kterým prochází. Schéma tohoto cyklu je dobře vyobrazené na obrázku 1.1. Bližší informace o každém stavu pak poskytne dokumentace [1].

Kromě základní třídy aktivity a rozšířených tříd pro zpětnou kompatibilitu nabízí Android také rozšířené účelové třídy aktivit určené například pro práci s posuvným seznamem či zobrazení a zpracování nastavení aplikace.

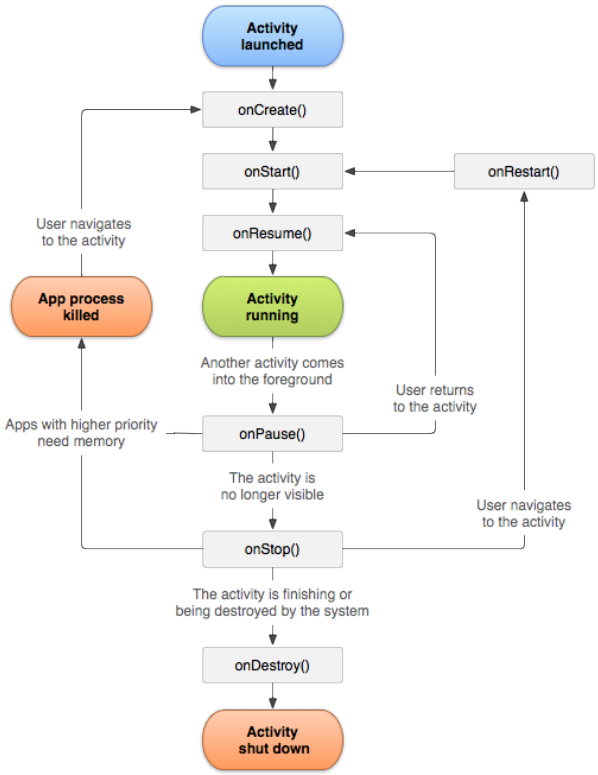
1.4.2 Informační dialog Toast

Toasty jsou jednoduché informační dialogy, ovšem o to častěji používané, které se krátce objeví na obrazovce s určenou informací a poté zmizí. Čas, po který se Toast objeví, není libovolně volitelný. Možnosti jsou jen dvě - krátké zobrazení (dle zdrojových souborů Androidu na webu [2] 2 s) a dlouhé zobrazení (3,5 s). Podstatným omezením je zobrazení vždy pouze jen jednoho Toastu, případné další jsou zařazeny do fronty.

1.4.3 Služby, AsyncTasky a Handlery

Android nabízí mnoho způsobů pro asynchronní spouštění kódu. Využít je samozřejmě možné i ruční vytváření a spravování vláken. Nejčastěji využívanými způsoby jsou však tyto tři - služby, AsyncTasky a Handlery.

[1] Obfuskace slouží k zamezení čitelnosti původního kódu, jsou například odstraněny komentáře a změněna pojmenování.



Obrázek 1.1: Životní cyklus Android aktivity

Služba (`Service`) je část aplikace bez grafického rozhraní a běžící nezávisle na pozadí tak, že může běžet i po ukončení zbylé části aplikace. Základní služba běží na hlavním vláknu, služba založená na třídě `IntentService` je spouštěna v samostatném vlákně.

AsyncTasky jsou přímo určeny pro asynchronní spouštění částí kódu mimo hlavní vlákno. Není třeba vytvářet a starat se o vlákna, stačí jen umístit vykonávaný kód do příslušné implementované metody AsyncTasku. AsyncTasky obsahují rozhraní pro informování hlavního vlákna o aktuálním stavu prováděné činnosti. AsyncTask je na rozdíl od služby svázaný s hlavní částí aplikace, takže se po jejím ukončení ukončí také. Od verze Androidu 3.0 Honeycomb výše je počet paralelně běžících AsyncTasků spouštěných základní metodou `execute(...):AsyncTask<Params, Progress, Result>` omezen na jediný, pro spouštění více paralelně běžících AsyncTasků je proto nutné využít metodu `executeOnExecutor(...):AsyncTask<Params, Progress, Result>`.

Handlery jsou určeny taktéž pro asynchronní spouštění kódu (a též zpracování zpráv), standardně však na rozdíl od AsyncTasků ovšem běží na hlavním vlákně. Typickým využitím je periodické opakování výpočetně nenáročné činnosti, například aktualizace zobrazené časomíry. Vytvořením vlákna třídy

`HandlerThread` a předáním jeho cyklovače (`Looper`) do parametru konstruktoru Handleru pak lze spustit Handler v novém vlákně.

1.4.4 Broadcast receivery a content receivery

Následné komponenty zmíním jen krátce, neboť v realizované aplikace nepředpokládám jejich využití. Pomocí broadcast receiveru lze zachytávát události, které se objeví v systému. Jde například o reakci na přišlou SMS, kterou může aplikace následně zpracovat. Content providery pak slouží k poskytování dat aplikace jiným aplikacím.

1.4.5 Layout

Layouty jsou XML soubory určené k definici grafických prvků (`View`) a jejich rozložení. Layouty se ukládají do skupiny `layout` v `resources`. Lze je vnořovat do sebe přímo v XML i programově. Layouty obsahují jako kořenový XML prvek jeden z následujících typů layoutu:

AbsoluteLayout Pro absolutně pozicované prvky.

FrameLayout Pro rezervaci oblasti k zobrazení typicky jednoho prvku či více se překrývajících prvků.

LinearLayout K zobrazení prvků pod sebou či vedle sebe.

RelativeLayout K zobrazení prvků relativně vůči sobě.

TableLayout Tabulkový layout dělený na sloupce a řádky.

Layouty se pomocí metody `setContentView(layoutResId:int):void` Android aktivity, kde `layoutResId` označuje javovský identifikátor layoutu ve formátu `R.id.%uvedené-id-layoutu-v-XML%`, předávají k zobrazení aktivitou.

Za layouty se dále někdy považují i prvky určené pro hromadné zobrazení podprvků. Mezi ně patří například `ListView` a `GridView`. `ListView` je hromadný prvek seznamu a `Grid View` je hromadný prvek sloužící k zobrazení vnitřních podprvků na dvoudimenzionální posunovatelné mřížce. Zmínit lze i `ScrollView` pro vložení podlayoutu k posunování. Všechny tři zmíněné prvky se označují jako takzvané kontejnery (`containers`).

1.4.6 View

View jsou jednotlivé grafické prvky zobrazené v rámci layoutu. View je široké množství, proto zde zmíním jen ty zřejmě nejčastěji používané:

EditText Textové pole. Může nabývat různých typů, od obyčejného pole přes vyhrazená pole pro datum a pole pro hesla, po víceřádková pole a pole s automatickým dokončováním obsahu.

TextView	Textový popisek. Může být taktéž víceřádkový.
Button	Běžné (formulářové) tlačítko. V novějších verzích SDK, dnes i se zpětnou kompatibilitou, bylo zavedeno alternativní plovoucí tlačítko `FloatingActionButton` určené především pro vyvolání nezávislých akcí, menu apod.
ImageView	Pro zobrazení obrázku.
RadioButton	Přepínací prvek pro výběr jedné hodnoty z více možností.
CheckBox	Zatrhávací prvek.
ToggleButton	Přepínací prvek typicky pro aktivaci, respektive deaktivaci nějaké funkce. Od API verze 14 lze alternativně použít prvek `Switch`.

Každé View musí mít určenu svoji šířku a výšku, ať již absolutně nebo relativně. Pomocí uvedeného identifikátoru, který musí být v rámci všech layoutů zobrazených jednou Android aktivitou unikátní, lze k prvku přistupovat z javového kódu. K prvkům se z Android aktivity přistupuje přes metodu `findViewById(id:int):View`, kde parametr je podobně jako u layoutu ve formátu `R.id.%uvedené-id-View-v-XML%`.

Vzhled prvků View lze upravit styly. Zatímco počet různých prvků je širší než v případě jazyka HTML pro webové stránky, možnosti úpravy vzhledu nejsou zdaleka tak široké, jako dnes známe z oblasti kaskádových stylů. Některé atributy vzhledu je přitom navíc nutné nastavit přímo danému prvku a nikoli ve stylu. Na druhou stranu bych ale neoznačil možnosti stylů za výrazněji limitující. S vývojem platformy Android se navíc i tyto možnosti stále vylepšují a rozšiřují.

Zařízení, na nichž Android běží, mají velmi rozmanité velikosti a rozlišení displejů, i proto se na Androidu silně dbá na responzivní design, který bude uživatelem mimo jiné snadno čitelný, respektive ovladatelný. Součástí tohoto přístupu je i preference uvádění velikostí prvků v jednotkách `dp` a `sp`. Obě jednotky jsou závislé na hustotě displeje, přičemž pokud `px` jsou pixely obrazovky a `dpi` hustota displeje v bodech na palec, platí vzorec `px = dp * dpi / 160`. Jednotka `sp` je dále škálovaná jednotka `dp` v závislosti na uživatelském nastavení velikosti písma v systému, užívá se proto právě zejména pro určení velikosti písma.

1.5 Perzistence dat

1.5.1 Úvod

Android nabízí několik možností pro trvalé uložení dat. Aplikace má možnost ukládat data do souborů do úložiště označeného jako interní, do souborů

do úložiště označeného jako externí, do databáze a pro ukládání nastavení je k dispozici rozhraní sdíleného nastavení. Data databáze a sdíleného nastavení jsou ukládána do interního úložiště.

1.5.2 Vnitřní a vnější úložiště

Záměrně jsem v úvodu napsal, že úložiště jsou označena jako interní a externí, konkrétní fyzické umístění souborů totiž záleží na zařízení. Zatímco dříve tato označení většinou odpovídala a interní úložiště byla vnitřní paměť zařízení a externí úložiště označovalo paměťovou kartu, v dnešní době, v souvislosti s růstem kapacity vnitřní paměti zařízení, značí mnohdy obě úložiště právě paměť vnitřní. Od Androidu 6.0 Marshmallow výše je pak podporována funkce „Adoptable Storage“, která umožňuje přeměnit paměťovou kartu v úložiště pevně svázané se zařízením a využít ho jako úložiště interní.

Základní rozdíly mezi interním a externím úložištěm jsou výchozí oprávnění, dostupnosti úložiště a svázanosti s aplikací (viz kapitola „Saving Files“ v dokumentaci Androidu [1]). V interním úložišti má každá aplikace svůj podadresář a ve výchozím nastavení k němu může přistupovat pouze ona. Ani uživatel nemá k datům mimo aplikaci přístup. Z externího úložiště může číst každý a zapisovat každá aplikace, která má příslušné oprávnění, přičemž aplikace může zapisovat do tohoto úložiště na libovolné místo. Interní úložiště je aplikaci dostupné vždy, externí může být nedostupné. Při odinstalaci aplikace se soubory uložené v interní paměti mažou spolu s aplikací, soubory v externím úložišti zůstávají nedotčeny.

Při vývoji aplikace je třeba počítat s tím, že zejména starší zařízení disponují velmi často interním úložištěm o malé kapacitě, a velké soubory tak ukládat raději do externího úložiště.

1.5.3 Databáze

Pomocí doplňujících knihoven lze v aplikaci pracovat s různými databázemi, nativní databází platformy Android je však databáze SQLite. Jde o jednoduchou a snadno použitelnou SQL databázi, zvolenou právě s ohledem na použití v mobilním zařízení. Pro práci s touto databází nabízí Android několik podpůrných systémových tříd, které přístup k ní z aplikace usnadňují. Jedná se především o třídy `SQLiteOpenHelper` a `SQLiteDatabase`, z nichž se první stará o spojení s databází a poskytuje rozhraní pro aktualizaci schématu databáze (databázových tabulek) a druhá poskytuje rozhraní pro vykonání SQL dotazů a příkazů.

1.5.4 Sdílená nastavení

Sdílená nastavení slouží k uložení nastavení a dalších perzistentně ukládaných hodnot používaných v rámci aplikace. Slovo „sdílená“ zde znamená, že jsou

tyto hodnoty přístupné odkudkoli v aplikaci. Kromě ukládání nastavení a hodnot jednotlivě poskytuje Android rozšiřující aktivitu `PreferenceActivity`, pomocí které lze vytvořit obrazovku pro hromadné nastavení. Prvky (grafické) pro tuto aktivitu se neuvádí v běžném layoutu, ale ve speciálním XML souboru s kořenovým prvkem `PreferenceScreen`, přičemž se dále v XML nepoužívají běžná View, ale opět prvky vytvořené přímo pro obrazovku nastavení.

1.6 Oprávnění jako prostředek zabezpečení

Android používá systém oprávnění zejména ke zvýšení zabezpečení uživatelských dat a informací, které se týkají soukromí uživatele. Například v této aplikaci budu využívat údajů z GPS senzoru a budu proto potřebovat minimálně oprávnění pro přístup k informaci o poloze zařízení, neboť se tato informace dotýká soukromí uživatele a je k ní tak bez oprávnění přístup zamezen.

Oprávnění aplikaci přiděluje uživatel. Na Androidu do verze 5.x je nutné poskytnout veškerá požadovaná oprávnění při instalaci aplikace. V aplikacích, které jsou vyvíjeny oproti SDK ve verzi 23 a vyšší, a tehdy, pokud jsou instalovány na zařízení s verzí Androidu 6.0 a vyšší, jsou oprávnění aplikaci přidělována, za běhu aplikace tehdy, kdy je aplikace vyžaduje, respektive přesněji, kdy je programátor nechá v aplikaci vyžádat.

Jak již bylo zmíněno dříve, vyžadovaná oprávnění se uvádí v manifestu, což bylo vše, co musel programátor donedávna učinit. Dnes, kdy se oprávnění udělují za běhu, to má programátor o to složitější. Již přidělená oprávnění navíc může uživatel v systémovém nastavení aplikaci opět odebírat, s čímž je třeba počítat. Před provedením akce je proto třeba ověřit, zdali aplikace oprávnění má, a pokud ne, tak si ho vyžádat. Jde o asynchronní proces, takže zachytávání informace o případném novém přiděleném oprávnění probíhá v jiné metodě. Konkrétně jde o volání `ActivityCompat.checkSelfPermission(...)` pro zjištění stavu oprávnění, `ActivityCompat.requestPermissions(...)` pro vyžádání oprávnění a metodu `onRequestPermissionsResult(...):void` (daná Android aktivita musí rozšiřovat třídu `AppCompatActivity`) pro zpracování výsledků žádosti.

1.7 Ladění a testování aplikace

1.7.1 Ladění aplikace

Možnosti ladění aplikace jsou závislé na konkrétním vývojovém prostředí. Co se týče Android Studia, jsou jeho schopnosti ladění aplikace, a též profilování aplikace či podpora rozlišení vývojových a finálních sestavení, dle mého názoru na porovnatelné úrovni s jinými známými pokročilými vývojovými prostředími a bylo by zbytečné je zde podrobněji popisovat.

Za zmínku ovšem stojí nástroj „Android Device Monitor“ z balíku Android nástrojů, který umožňuje sledovat stav emulátoru či zařízení. Sledovat je možné co se týče zvolené aplikace například alokace na heapě či spuštěné thready, dále, pokud to zařízení povolí, obsah interní paměti, průběh síly signálu WiFi, využití CPU, využití RAM a další.

Při vývoji pak občas přijde vhod také teminálový nástroj „adb“, kterým je možné do zařízení instalovat aplikace, oběma směry přenášet data, pracovat se souborovým systémem a mnohé další.

1.7.2 Testování aplikace

Testování aplikace jsem se již krátce dotkl v části popisující schéma projektu. Pro připomenutí, zmiňoval jsem, že testy se na Androidu dělí na testy nezávislé na platformě Android a testy, které běží pod systémem Android.

Zmínil jsem též, že první kategorie jsou typicky jednotkové testy. Některé třídy lze takto snadno otestovat, ovšem mnohé jsou závislé na některé ze systémových tříd Androidu. V takovém případě je třeba objekty systémových tříd mockovat. [2] Objekty třídy `Context` (což jsou například i aktivity) a některé další třídy (od verze 7.1 Nougat jejich počet ubyl) je možné mockovat přímo API nabízeným Androidem, u ostatních tříd je nutné využít dodaného frameworku. Obecně doporučován je framework Mockito, avšak mnohdy jeho schopnosti nestačí a je potřeba využít pokročilejší PowerMockito. Výchozím testovacím frameworkem pro jednotkové testy je v Android Studiu JUnit, ale lze pracovat i s jiným.

Druhou kategorií testů jsou testy integrační a funkční. Pro psaní těchto testů je k dispozici framework Espresso, pomocí kterého lze popsat průchod aplikací a ověřovat její stav. Tyto testy není nutné psát v Android Studiu celé ručně, ale lze využít funkce nahrání testu, kdy programátor může průchod aplikací naklikat a v kódu testu následně doplnit jen požadovanou verifikaci stavů.

Velkým pomocníkem při testování aplikace je pak také emulátor, který umožňuje simulovat různé stavy senzorů a další stavy zařízení. Mimo jiné zde lze simulovat třeba GPS polohu zařízení, což jistě často upotřebím během vývoje realizované aplikace.

[2] Mockování znamená nahrazení objektu nějaké třídy zástupným objektem, který se jako objekt stejné třídy tváří, ale ve skutečnosti jím není.

Kapitola 2

Funkcionálně reaktivní programování

2.1 Stručná historie FRP

Koncept funkcionálně reaktivního programování je mnohem novější než objektově orientované programování, které plně zavedl Smalltalk v sedmdesátých letech. Nejde ovšem rozhodně o novinku, poprvé se tento koncept (paradigma) objevil již před 20 lety[3]. Až v posledních letech se však stává populárním a šířeji využívaným. Začaly vznikat knihovny a frameworky, které umožňují, respektive usnadňují, používat toto paradigma v různých programovacích jazycích a na různých platformách. A mezi tyto jazyky dnes patří i Java a mezi platformy Android.

2.2 Koncept FRP

Velmi zjednodušeně řečeno je funkcionálně reaktivní programování událostmi řízeným programováním nad asynchronními toky dat v aplikaci za využití prvků funkcionálního programování, což jsou posloupnosti jednoduchých operací (transformací) založených na lambda kalkulu. Ze slova reaktivní pak také vychází požadavek na rychlou odezvu na požadavky a reakce na změny v aplikaci. Přesněji definovaných pohledů na FRP existuje více, jedním z nich je například reaktivní manifest, který uvádím následně.

Událostmi řízené programování se dělí na dva základní typy, „push-based“ a „pull-based“, a oba se pokládají za součást FRP. Přístup „push-based“ vyšle vzniklou událost k cíli či cílům okamžitě. V případě „pull-based“ přístupu naopak zdroj vzniklé události čeká na její vyžádání.

2.3 Reaktivní manifest

Dalším pohledem na funkcionálně reaktivní programování je reaktivní manifest [4], který byl zveřejněn skupinou programátorů v roce 2014 a který definuje čtyři základní body, které by měla plně reaktivní aplikace splňovat. Těmito body jsou responzivita, odolnost, pružnost a řízení událostmi. Volně přeloženo a zkráceno jsou definovány následovně:

responzivita
: Pokud je to možné, systém reaguje včasně. Responzivita je základním pilířem použitelnosti a funkčnosti. Responzivní systémy cílí na konzistentní a rychlou odezvu. Toto chování usnadňuje zpracování chyb, zvyšuje spokojenost uživatelů a podporuje je v dalším užívání aplikace.

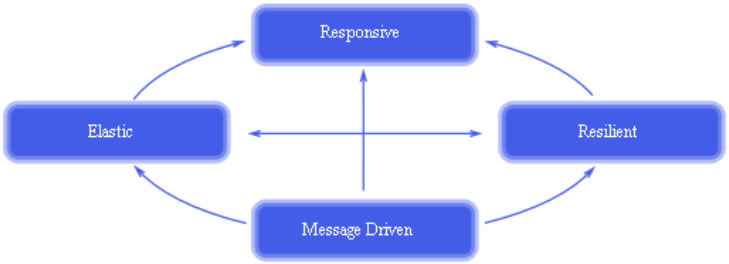
odolnost
: Systém je odolný vůči chybám a v případě chyby zůstává responzivní. Tato vlastnost platí pro veškeré systémy a aplikace, nejen ty kritické. Pro zajištění odolnosti se využívá replikace, izolace a delegace. Izolace zajistí, že chyba nezpůsobí pád celého systému a bude možná obnova do plně funkčního stavu. O obnovu komponent se starají jiné komponenty (delegace) a vysoká dostupnost je zajištěna replikací.

pružnost
: Systém zůstává responzivní nezávisle na zátěži. Reaktivní systémy mohou měnit rychlost vstupů snižováním či navyšováním prostředků dané komponentně dostupných, čímž je zajištěno, že v systému či aplikaci neexistuje centrální úzký bod.

řízení událostmi
: Reaktivní systémy spoléhají na asynchronní výměnu zpráv mezi komponentami, které zaručí oddělení komponent a tím nízkou provázanost systému. Toto oddělení znamená též předávání informací o chybách formou zpráv. Komunikace je neblokující, což znamená, že příjemci zpráv využívají prostředky pouze tehdy, kdy je potřebují, tedy na ně kladou nižší nároky.

2.4 Návrhový vzor pozorovatel

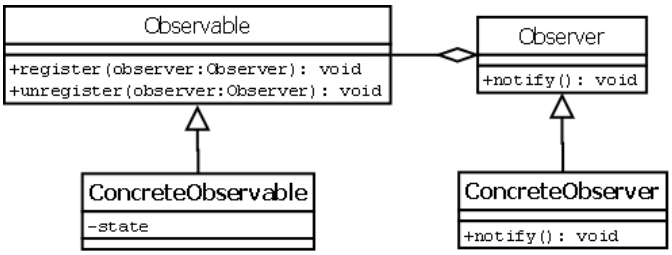
Jak bylo zmíněno, (funkcionálně) reaktivní programování je řízené událostmi. Pro implementaci tohoto přístupu se využívá návrhového vzoru „pozorovatel“ (anglicky observer), kde na jedné straně vystupuje objekt pozorovaný (observable či subject) a na druhé objekty pozorující (observer), které na pozorovaném objektu závisí. Pro vytvoření či naopak zrušení závislosti obsahuje pozorovaný veřejnými metodami, pomocí kterých se k němu pozorovatel registruje, respektive od něj odregistruje. V případě, že se změní stav pozorovaného



Obrázek 2.1: Vztah vlastností reaktivního systému dle manifestu [4].

objektu, informuje o tomto stavu všechny pozorující objekty, které následně vykonají na základě přijaté zprávy požadovanou činnost. UML diagram tohoto vzoru je zobrazen na obrázku 2.2.

Co se týče vztahu k reaktivnímu systému či aplikaci pokud jde o podmínku na asynchronnost, ta platí tehdy, reagují-li pozorovatelé na přišlou zprávu asynchronně.



Obrázek 2.2: Návrhový vzor pozorovatel - UML diagram

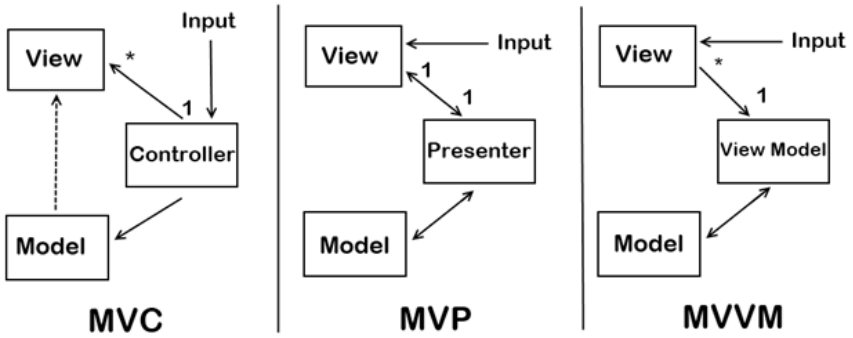
2.5 Základní rozdíly FRP oproti OOP

Rozdíl reaktivního programování oproti objektově orientovanému programování vychází již z návrhového vzoru pozorovatel. Zatímco v případě obj. or. programování se často využívá proměnného sdíleného stavu, reaktivní programování využívá zasílání zpráv mezi pozorovanými a pozorovateli.

Funkcionální programování je silně založeno na tzv. „čistých“ funkcích (pure function), které na stejný vstup vždy vrací stejné hodnoty a nevyužívají a neovlivňují v průběhu svého vykonávání své okolí. Právě tyto čisté funkce lze přesně popsat lambda kalkulem. Zatímco tedy obj. or. programování je koncept týkající se objektů a vazeb mezi nimi, koncept funkcionálního programování se zabývá především operacemi. S tím souvisí i podmínky, za kterých je daný koncept vhodnější. U obj. or. programování je poměrně snadné přidávat nové typy, respektive podtypy, objektů a u funkcionálního naopak přidávat novou funkcionalitu založenou na stávající sadě typů objektů.

Transformační funkce funkcionálního programování jsou obvykle velmi jednoduché a k požadovanému výsledku vede série zřetězených funkcí. U objektově orientovaného programování toto často neplatí a též zpravidla dochází k hojnému využívání pomocných proměnných. U funkcionálního programování se tyto nevyužívají vůbec.

Architektura Model-View-ViewModel Funkcionálně reaktivní programování mění také architekturu aplikací s uživatelským rozhraním. Zatímco obj. or. aplikace užívají typicky archiktektury Model-View-Controller či v případě Androidu zřejmě častěji Model-View-Presenter, FRP přichází s architekturou Model-View-ViewModel. Zatímco u MPV a MVC jsou hodnoty View aktualizovány z kontroleru, respektive modelu, u MVVM jsou hodnoty zveřejňovány ve ViewModelu prostřednictvím pozorovaného, nezávislého na konkrétním View. Jednotlivá View pak plní roli pozorovatelů, kteří se k pozorovaným ViewModelu registrují. UML modely těchto architektur jsou zobrazeny na obrázku 2.3.



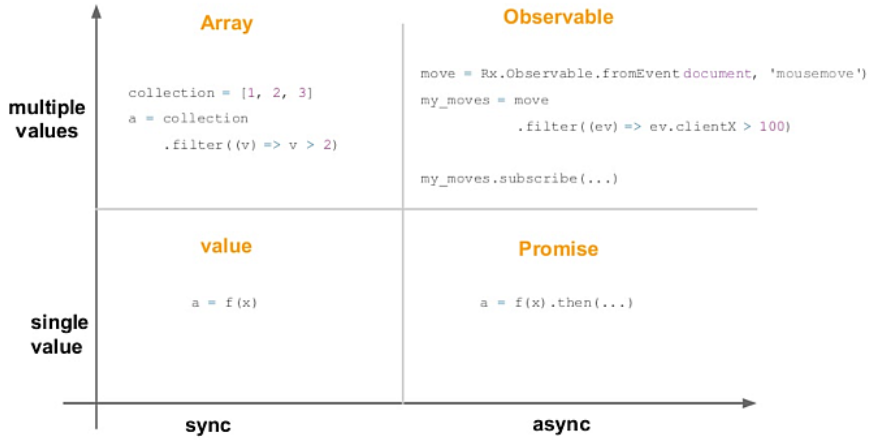
Obrázek 2.3: UML modely architektur aplikací – zleva Model-View-Controller, Model-View-Presenter a Model-View-ViewModel; Obrázek pochází z [5].

2.6 Komponenty FRP

2.6.1 Komponenty reaktivního programování

Stavebními prvky reaktivního programování jsou již známá dvojice pozorovatel a pozorovaný, reálně se však u reaktivního programování setkáváme s větším počtem více specializovaných prvků. Kromě základních „observable“ se lze například setkat s termínem „subject“. V některých implementacích může jít o běžného pozorovaného, ale například RxJava, které se věnuji dále v textu, označuje za „subject“ objekt, který plní roli jak pozorovaného, tak pozorovatele. Jelikož běžný „observer“ není limitován počtem přijatých zpráv, existuje v některých implementacích specializovaný podtyp „single observer“ a od-

povídající pozorovaný „single“, u kterých dochází ke zpracování vždy pouze jedné zprávy. Někdy se lze pro tento podtyp setkat, namísto označení „single“, s označením „promise“.



Obrázek 2.4: Reaktivní podoba hodnot a polí (zde využita reaktivní knihovna Rx.js a JavaScript; Ze slidu přednášky [6].

Pozorovaní se dále dělí na tzv. „horké“ („hot“) a „chladné“ („cold“). „Horký“ pozorovaný začíná vysílat zprávy ihned, „chladný“ čeká, než ho začne nějaký pozorovatel pozorovat. Druhý přístup zajišťuje, že pozorovatel obdrží veškeré vytvořené zprávy.

2.6.2 Komponenty funkcionálního programování

Aby mohly funkce provádět transformace, potřebují data. Tato vstupně-výstupní data jsou označována jako „proudy“ („streamy“). U reaktivních systémů pak vystupuje proud pro každou funkci jako skupina pozorovaných a transformační funkce jako pozorovatel. Zatímco klasičtí „observable“ bývají push-based, proudy jsou pull-based a nemůže tak dojít k výpadku některé zprávy. Zpracování u (plně) reaktivních systémů probíhá též plně asynchronně, nečeká se tedy, než se každá daná transformace provede pro všechna vstupní data.

Transformačních funkcí je široké množství, ale knihovnami jsou podporovány nejméně následující nejčastěji užívané transformační funkce:

map — Transformuje přijatou hodnotu na jinou hodnotu.

flatMap — Transformuje přijatou hodnotu na více hodnot.

filter — Propustí jen hodnoty splňující zadanou podmínku.

reduce — Spojí zadaným způsobem hodnoty do jedné, například zřetězením nebo součtem hodnot.

Další funkce dostupné ve funkcionálně reaktivní knihovně RxJava, které se budu věnovat, uvedu v podsekci 2.7.2.

2.7 FRP na platformě Android

2.7.1 Stručný přehled

Streamy a lambda výrazy jsou součástí Javy 8, ovšem o její problematické podpoře na Androidu jsem psal již v úvodu. Vývoj software jde, ozvlášť ve velmi dynamickém mobilním segmentu, rychle dopředu, takže se situace pravděpodobně velmi brzy zlepší, dnes ale podle mého názoru udělá programátor lépe, využije-li knihoven a doplňků třetích stran.

Podporu lambda výrazů v javovém kódu přidává Gradle plugin Retrolambda. U podpory streamů jsou možnosti dvě, jednak přímo tomuto účelu vyhrazená knihovna StreamSupport, a pak také knihovna RxJava, která nabízí širokou podporu funkcionálně reaktivního programování.

Knihovna RxJava byla v druhé polovině roku 2016 vydána ve své druhé verzi, která obsahuje výčet schopností ještě širší. RxJava je knihovna zaměřená obecně na Javu, proto se při vývoji na Androidu přidává ještě knihovna RxBinding, která propojuje RxJavu a Android API. Pro automatické ukončení registrací k pozorovaným na základě životního cyklu Android aktivity či Fragmentu pak slouží knihovna RxLifecycle. Tato knihovna však s sebou nese určité úskalí, které popíšu v sekci 2.7.7.

Jako poslední zmíním knihovnu RxBus. Tato doplňková knihovna přidává podporu pro registraci pozorovatele k pozorovanému pomocí anotací, což je výhodné zejména v případě komplexnějších aplikací s větším počtem pozorovaných, pozorovatelů a jejich vazeb, kde se tímto usnadňuje použití a do jisté míry zvyšuje přehlednost.

2.7.2 RxJava 2

2.7.2.1 Úvod

Knihovna RxJava [7] je implementací funkcionálně reaktivního rozhraní ReactiveX v jazyce Java. Mimo Javy existují implementace pro C#, C++, JavaScript, Python a mnohé další. Zkušenosti s reaktivním vývojem v jednom jazyce tak může programátor téměř okamžitě plně upotřebit i v jiném jazyce, rozdíly jsou malé.

V případě RxJavy budu popisovat již jen její druhou verzi. Ta je sice nedávno vydaná a širší dokumentace ještě potřebuje doplnit, na druhou stranu ovšem při praktickém využití působí bezproblémově a stabilně a chybějící části vysvětlující dokumentace se týkají především nově přidaných možností. Za vývojem RxJavy stojí známá společnost Netflix Inc., což samo o sobě

zvyšuje důvěru v kvalitu provedení a rychlé odstraňování případných chyb. Osobně tak nevidím důvod, proč setrvávat u verze první [3].

Dokumentaci k RxJavě lze nalézt na oficiálních stránkách rozhraní ReactiveX (`http://reactivex.io`), kde jsou základní prvky a transformační funkce (operátory) vysvětleny společně a následně pro jednotlivé implementace uvedeny příklady a pro operátory dále případné další podobně zaměřené operátory a též příklady k nim. Právě části dokumentace vyhrazené implementaci v podobě RxJavy 2 vyžadují k dnešnímu dni doplnění. Čerpat informace lze samozřejmě i z Javadocu, který je dostupný i online [8].

2.7.2.2 Prvky užívané v RxJavě

RxJava zavádí šest základních typů pozorovaných – běžný `Observable`, pozorovaný nabízející regulaci rychlosti emitace `Flowable`, pro jedinou zprávu určený `Single`, dále kombinace s funkcí pozorovatele `Subject`, pozorovaný emitující pouze informaci o ukončení či chybě `Completable` a kombinace `Single` a `Completable` v podobě `Maybe`.

Speciálním typem pozorovaného je `ConnectableObservable`, který nevytváří mělké kopie objektů pro jednotlivé pozorovatele, ale emitované hodnoty sdílí. Hodnoty dále nezačínají být emitovány ihned, ale až po zavolání metody `connect`.

Typ pozorovaného `Subject` se dále dělí na další čtyři podtypy v závislosti na tom, kdy a které hodnoty pozorovatel obdrží. U `AsyncSubject` obdrží pozorovatel pouze poslední hodnotu a to až po oznámení ukončení emitace hodnot. U `BehaviorSubject` obdrží poslední hodnotu před registrací a dále všechny následující. V případě `PublishSubject` jsou obdrženy pouze hodnoty emitované po registraci a u posledního `ReplaySubject` jsou nově registrovanému pozorovateli zpětně emitovány všechny již dříve vyslané hodnoty a též všechny následující.

Pro pozorované `Single`, `Completable` a `Maybe` existují vlastní podtypy observerů `SingleObserver`, `CompletableObserver`, respektive `MaybeObserver`. Pozorovaný `Flowable` využívá klasický `Observer`.

K pozorovaným se pozorovatelé registrují pomocí metody `subscribe(...)`. Existují zde dva základní druhy pozorovatelů, resp. dvě základní generické třídy. Jedním pozorovatelem je `Observer`, který musí implementovat metody `onSubscribe(...)`, `onNext(...)`, `onComplete()` a `onError(...)`. Druhým pozorovatelem je `Consumer` (v RxJavě verze 1 se používalo označení `Action`) s jedinou povinně implementovanou metodou `accept(...)`. Pro možnost zpracování chyb i v případě využití pozorovatele `Consumer`, byla vytvořena přetížená metoda `subscribe(...)` se dvěma parametry typu `Consumer`, z nichž první zpracovává emitované hodnoty a druhý slouží ke zpracování chyb.

[3] Je pravdou, že jsem v praxi využil jen malý zlomek nabízené funkcionality, takže plnou stabilitu zdaleka nemohu potvrdit, šlo ovšem zřejmě zejména o funkcionalitu nejčastější.

Typy `Observer` a `Consumer` se liší také způsoben ukončení registrace. V případě typu `Observer` si tento při registraci pozorovaného přes metodu `onSubscribe(disposable:Disposable` v podobě speciálního objektu typu `Disposable` zaznamená. Po ukončení emise hodnot zavolá pozorovaný metodu `onComplete()` (ta je v případě streamů zavolána automaticky) observeru a ten se postará o odregistraci zavoláním metody `dispose():void` na objekt `Disposable`. U pozorovaného typu `Consumer` je postup odlišný. Při jeho registraci se objekt typu `Disposable` stává návratovou hodnotou. O zavolání metody `dispose():void` na tento objekt se tedy musí postarat pozorovaný.

Pozorovatele je možné nechat pracovat na zvoleném typu vláken volbou plánovače vláken (třída `Scheduler`) pomocí metody `subscribeOn(scheduler-:Scheduler)` a alternativně `observeOn(scheduler:Scheduler)`. První uvedení metoda definuje, na kterém vlákně se má začít, přičemž pozice v řetězci transformačních funkcí není podstatná. Druhá metoda ovlivňuje volbu vláken pro transformační funkce v řetězci následující a může být až v počtu odpovídajícím počtu transformačních funkcí, před každou z nich.

2.7.2.3 Základní operátory v RxJavě

V RxJavě jsou samozřejmě přítomny již uvedené čtyři transformační funkce považované za ty nejzákladnější. Jejich celkový počet je ale řádově vyšší. Uvedu krátce některé z nich, strukturovaně dle jejich určení.

Operátory tvořící pozorované

from... Existuje v mnoha specializovaných podobách, např. `fromArray` pro tvorbu `Observable` z pole hodnot, `fromCallable` z výsledku vyhodnocení zadané metody či `fromIterable` z iterovatelného prvku.

just Tvoří společného pozorovaného pro jeden až devět zadaných objektů.

range Generuje pozorovaného emitujícího posloupnost celých a po sobě přímo následujících čísel – `m` prvků od zadaného `n`.

interval Pro postupnou emitaci posloupnosti celých od nuly a s daným časovým odstupem jednotlivých hodnot.

timer Emituje jednu hodnotu po zadaném časovém intervalu.

Transformační operátory (mimo `map` a `flatMap` jde o:)

groupBy Rozděluje emitované hodnoty do kategorií, které tvoří noví pozorovaní.

buffer Seskupuje emitované hodnoty na základě zadaného časového intervalu. RxJava nabízí také mimo jiné přetíženou metodu `buffer`, která seskupuje hodnoty do skupin o zadaném počtu.

Filtrující operátory (mimo `filter` jde o:)

debounce Tento operátor posílá k dalšímu zpracování pouze poslední obdrženou hodnotu v určeném časovém intervalu.

skip Prvních `n` hodnot zadaných v parametru operátoru je vyjmuto z dalšího zpracování. Pro vyjmutí naopak posledních `n` hodnot slouží operátor `skipLast`

Vykonání akce

doOn:. Jde například o operátory `doOnNext` `doOnCompleted` či `doOnError` se zadaným pozorovatelem v parametru. Podobně jako u observerů dojde k jejich zavolání při emisi další hodnoty, respektive ukončení emise, respektive vzniku chybové události. Stejná podobnost platí i u operátorů `doOnSubscribe` a `doOnDispose`, které reagují na registraci, respektive ukončení registrace pozorovatele.

Zpracování chyb

„catch“ Tento operátor se v RxJavě nachází v implementacích pojmenovaných `onErrorReturn`, `onErrorResumeNext` a `onExceptionResumeNext`. `onErrorReturn` emituje v případě chyby speciální hodnotu a další emitaci ukončí. Metody `onErrorResumeNext` a `onExceptionResumeNext` v případě chyby, respektive obdržené výjimky, začnou emitovat hodnoty záložního pozorovaného.

retry V případě chyby zkouší pozorovatele znovu registrovat. V RxJavě existují tři přetížené metody – první bez parametru nemá omezení na opakování, druhá má jako parametr maximální počet opakování a třetí bere jako parametr funkci, která rozhodne o pokračování či nepokračování návratem logické hodnoty. Čtvrtou variantu je operátor `retryWhen`, který bere jako parametr opět funkci a o opakování rozhodne, navrátí-li se emitovaná hodnota nebo chyba.

Agregační operátory (mimo `reduce` jde o:)

sum, ... Navrátí jako pozorovaného součet všech emitovaných hodnot poté, co pozorovaný oznámí ukončení emitace. Podobně jsou zde operátory `min`, `max` či `count`, jejichž účel je zřejmý.

concat Tento operátor agreguje hodnoty jen co se týče emitace pozorovaných. Tyto hodnoty pak agreguje do jednoho dále emitovaného proudu. Agregovat je též možné dva a více proudů přímo zadaných pozorovaných.

Speciální operátory

delay Zpozdí všechny emitované hodnoty o zadaný časový interval

2.7.3 RxJava a paralelní zpracování

Byť zpracování událostí pozorovateli může probíhat na základě zvolení plánovače vláken metodou `subscribeOn(...)` paralelně, samotné zpracování událostí jedním pozorovatelem paralelní není. Ačkoli existuje doplňková knihovna `RxJavaParallel`, jedná se o knihovnu v experimentálním stadiu, určenou pro RxJavu verze 1 a již dlouhodobě nerozvíjenou. Existují však řešení, jak paralelizace dosáhnout. Řešení popisuje ve svém článku [9] Thomas Nield. Využívá operátoru flatMap způsobem, který transformuje přijaté hodnoty na pozorované a k nim jednotlivě registruje pozorovatele. Pro lepší ilustraci přidávám část kódu uvedenou ve zmíněném článku:

```
Observable<Integer> vals = Observable.range(1,10);

vals.flatMap(val -> Observable.just(val)
            .subscribeOn(Schedulers.computation())
            .map(i -> intenseCalculation(i))
).subscribe(val -> System.out.println(val));
```

Výpis hodnot je prováděn sekvenčně, ale v pořadí, v jakém bylo dokončeno jejich paralelní zpracování metodou `intenseCalculation(...)`.

2.7.4 RxBinding 2

Knihovna RxBinding [10] nabízí sadu tříd pro umožnění registrace pozorovatelů k událostem grafických prvků Androidu. Najdeme zde tak například třídy `RxView` či `RxTextView` s metodami pro vytvoření pozorovaných z jednotlivých událostí daného View, například metodu události kliknutí na prvek `clicks(view:View):Observable<Object>`, která ve spojení s již známou metodou `subscribe(...)`, tedy registrací pozorovatele, tvoří ekvivalent metody `addOnClickListener(...)` View pro při registraci posluchače. Druhá verze je uzpůsobená pro použití s RxJavou 2.

2.7.5 Retrolambda

Retrolambda se stará o převod lambda výrazů do formátu podporovaného Javou 7. Vyžaduje pouze dva úkony, uvedení v seznamu závislostí v Gradle,

v tomto případě v konfiguračním souboru celého projektu a nikoli modulu, jako je tomu u knihoven. Druhým úkonem je uvedení kompatibity s Javou 8 v podsekci `compileOptions` sekce `android` konfiguračního souboru modulu. Tímto úkonem vypneme zobrazování chybových oznámení vývojového prostředí o nepodpoře v kódu uvedených lambda výrazů. Lambda výrazy lze pak užívat stejným způsobem, jako v Javě 8.

Retrolambda existuje ve více provedeních. Pro projekt Android Studia je zřejmě nejvhodnější využít provedení ve formě Gradle pluginu [11].

2.7.6 StreamSupport

Tato knihovna [12] je vyhrazena čistě pro podporu streamů. Na rozdíl od RxJavy se snaží být přímým náhradníkem podpory streamů v Javě 8 a zachovává tak třeba i stejné názvosloví. Podstatný rozdíl, který jsem nalezl, je přímá podpora paralelního zpracování, pro kterou v případě této knihovny stačít užít operátoru `parallel` a není tak třeba poměrně složitě konvertovat hodnoty na pozorované.

2.7.7 RxLifecycle

Knihovna RxLifecycle [13] slouží pro automatické ukončení registrací pozorovatelů k určeným pozorovaným při ukončení Android aktivity či Fragmentu. Požadavek na navázání k životnímu cyklu jedné ze zmíněných Android komponent se pozorovateli předává před registrací pozorovatelů. Podmínkou použití je však připravenost této Android komponenty k tomuto úkonu. Toho lze docílit například založením komponenty na některé z předpřipravených tříd, například `RxActivity` či `RxFragment`, odpovídajících svým ekvivalentům bez předpony „Rx“. Mezi těmito třídami se ovšem nenachází například rozšířená třída pro `ListActivity` či `MapFragment`. Druhou možností je použití pomocné knihovny s názvem Navi určenou k vytváření interakcí s komponentami Androidu. Třetí možností je pak implementace vlastního poskytovatele požadovaných informací o stavu Android komponenty.

Vzhledem k jisté složitosti použití této knihovny je tak u jednodušších aplikací zřejmě vhodnější spravovat si registrace ručně přes metody životního cyklu Androidu a za použití `CompositeDisposable`.

2.7.8 RxBus

Knihovna RxBus [14] slouží, jak již bylo uvedeno, k usnadnění a zpřehlednění registrace pozorovatelů k pozorovaným za pomoci anotací. Pozorovatel i pozorovaný mají každý svoji anotovanou metodu, pozorovaný označenou `@Produce` a pozorovatel `@Subscribe`. Nutné je rozlišit, který pozorovaný emituje pro kterého pozorovatele, pro což se využívá atributu `tags`. Důležitá je podpora volby vlákna, respektive plánovače vláken, tedy asynchronního zpracování. Tato volba se provádí přes atribut `thread`.

Jak vyplývá z názvu knihovny, její implementace je založena na sběrnici, konkrétně reaktivní sběrnici událostí. K této sběrnici je třeba instanci dané třídy nejprve registrovat (volání `RxBus.get().register(object:Object)-:void`) a po ukončení činnosti odregistrovat (`RxBus.get().unregister(object:Object):void`).

Kód třídy využívající RxBus tak bude vypadat nějak takto:

```
...
    protected void onStart() {
        ...
        RxBus.get().register(this);
        ...
    }
...
    protected void onStop() {
        ...
        RxBus.get().unregister(this);
        ...
    }
...

    @Produce(
            thread = EventThread.COMPUTATION,
            tags = {
                    @Tag("SubscriptionA")
            }
    )
    public String produce() { ... }
...
    @Subscribe(
            thread = EventThread.IO,
            tags = {
                    @Tag("SubscriptionA")
            }
    )
    public void consume(String item) { ... }
...
```

Kapitola 3

Reálné setkání s FRP

3.1 Úvod

Předchozí kapitola popisuje funkcionálně reaktivní programování a dostupné knihovny na základě veřejně dostupné dokumentace a článků věnujících se tomuto tématu. Před začátkem používání funkcionálně reaktivního programování v cílové aplikaci jsem však považoval za vhodné, vyzkoušet si ho na jednoduchých příkladech předem. A právě těmto příkladům se budu věnovat v této kapitole. Příklady jsou součástí Android projektu (aby mohla být vyzkoušena i funkcionalita knihoven navázaných na Android), který je též umístěn na přiloženém CD.

3.2 Struktura FRP projektu

Projekt „FRP Examples“ obsahuje sady příkladů rozčleněné do několika kategorií. Každá sada příkladů má pak vyhrazenu vlastní Android aktivitu, mající na starosti interakci s uživatelem. Spuštění příkladů provádí uživatel stisknutím tlačítka v aplikaci, kdy dané tlačítko je vyhrazeneno určitému paradigmatu, knihovně či variantě.

Podstatné je zmínit balíček `examples`, který obsahuje dle příslušené kategorie do podbalíčků rozdělené třídy s metodami zajišťujícími zpracování jednotlivých příkladů. Výsledky vyhodnocení příkladů se zobrazí pod nadpisy jednotlivých příkladů. U první sady příkladů se výsledky zobrazují i s orientačním časem, který vyhodnocení zabralo.

3.3 Příklady kategorie Stream

Nyní se již budu věnovat konkrétním příkladům, přičemž začnu s příklady věnujícími se streamům.

3.3.1 Sada příkladů Stream 01

Tato sada se věnuje zpracování pole, respektive seznamu čísel typu objektového `Long` i primitivního `long` pomocí cyklu, proudů knihovny SupportStream a proudů RxJavy. Úkol pro demonstraci je každou hodnotu zdvojnásobit a všechny dvojnásobky původních hodnot sečíst.

Stream 01 – Příklad 01 První příklad pracuje se seznamem (`ArrayList`) hodnot `Long`. Vyhodnocení probíhá vždy jednovláknově na hlavním vlákně.

Varianta cyklu, na základě zadání úkolu, prochází postupně seznam hodnot a průběžně přičítá k celkovému výsledku dvojnásobek aktuální procházené hodnoty. Jsem přesvědčen, že ukázku tohoto jednoduchého kódu zde není potřeba uvádět.

Varianta streamů SupportStream volá statickou metodu `stream(...)` s objektem seznamu jako parametrem na třídu `SupportStream`. Řetězec volání pak dále pokračuje operátorem `map`, vracejícím dvojnásobek přijaté hodnoty, operátorem `reduce` pro provedení součtu a nakonec operátorem `get`, který se stará o převedení přepravního objektu hodnoty (třída `Optional`) na objekt `Long`. Kód pro tuto variantu příkladu tak vypadá následovně:

```
...
... UnaryOperator<Long> unOpDouble = a->2*a;
... BinaryOperator<Long> biOpSum = (a,b)->a+b;
...
public void example01() {
    Long sum = StreamSupport
                        .stream(vLongList)
                        .map(unOpDouble)
                        .reduce(biOpSum)
                        .get();
    ...
}
...
```

Kód příkladu RxJava vypadá velmi podobně, změnily se jen názvy tříd a operátorů. Třídu `StreamSupport` nahradila třída `Observable`, namísto metody `stream(...)` je tu metoda `fromIterable(...)` a operátor `get` se změnil na `blockingGet`. Změnily se i parametry operátorů. Jak je vidět na předchozí ukázce, transformace jsou v případě SupportStream definovány ve třídách `UnaryOperator`, respektive `BinaryOperator` pracujícími s objekty stejné třídy. U RxJavy se ke stejnému účelu užívá tříd `Function` a `BiFunction`, přičemž vstupní a výstupní objekty mohou být různých typů. Výsledný kód je tedy následující:

```
...
... Function<Long,Long> fDouble = a->2*a;
... BiFunction<Long,Long,Long> biFsum = (a,b)->a+b;
...
public void example01() {
    Long sum = Observable
                        .fromIterable(vLongList)
                        .map(fDouble)
                        .reduce(biFsum)
                        .blockingGet();
    ...
}
...
```

Všechny tři varianty dávají očekávaný výsledek. Orientační srovnání rychlostí všech variant uvedu později.

Stream 01 – Příklad 02 Druhý příklad se liší zdrojem dat, kterým není seznam, ale pole.

V kódu cyklu se kromě proměnné zdroje oproti prvnímu příkladu nemuselo měnit vůbec nic.

Třída `StreamSupport` v metodě `stream` pole jako parametr nepřijímá. K dispozici je ovšem třída `J8Arrays` se stejně nazvanou metodou, která z pole objektů stream vytvořit dokáže. Po této změně již není v kódu třeba nic měnit.

U RxJavy se nemění třída, ale naopak metoda. Místo metody nazvané `fromIterable` použijeme `fromArray` a potřebná úprava je hotová.

Stream 01 – Příklad 03 Příklad třetí je shodný jako druhý, jen výpočet neprobíhá na hlavním, ale jednom samostatném vlákně.

Varianta cyklu využívá objektu systémové třídy `Handler`, který spouští v novém vlákně třídy `ThreadHandler` dané `Runnable` obsahující kód výpočtu a následně volání metody `runOnUiThread()` android Aktivity pro výpis výsledku do příslušného View.

Knihovna StreamSupport možnost spouštění na jednom separátním vlákně nemá, takže se příkladu neúčastní.

RxJava využívá již známou metodu `subscribeOn(...)` a jako plánovač vláken byl zvolen `Schedulers.computation()`.

Stream 01 – Příklad 04 Zdrojem je opět pole dat, ale výpočet tentokrát probíhá paralelně.

Varianta cyklu pro složitost řešení v tomto příkladu implementována nebyla.

U knihovny StreamSupport stačí užít operátoru `parallel`.

Paralelnímu zpracování dat jednoho pozorovaného v RxJavě za pomoci operátoru `flatMap` jsem se již v samostatné podsekci věnoval. Reálný pokus o takové zpracování však na testovaném objemu dat nedopadl úspěšně, neboť byl příliš výpočetně náročný. Využil jsem proto operátoru `buffer`, který spojí zadaný počet emitovaných hodnot do jedné. Z těchto spojených hodnot jsem následně pomocí operátoru `flatMap` vytvořil nové pozorované, jejichž úkolem bylo na výpočetních vláknech zdojnásobit jim přiřazené hodnoty a sečíst je dohromady. Sečtení těchto mezivýsledků pak proběhlo sekvenčně. Pro tento příklad jsem zvolil za velikost bufferu desetinu velikosti pole, pole tedy bylo rozděleno do deseti vláken. Kód vypadá následovně:

```
...
public void example04() {
    Long sum = Observable
        .fromArray(vLongArray)
        .buffer(vLongArray.length/10)
        .flatMap(val -> Observable
                .fromIterable(val)
                .subscribeOn(Schedulers.computation())
                .map(i -> 2*i)
                .reduce(biFsum)
                .toObservable())
        .reduce(biFsum)
        .blockingGet();
    ...
}
...
```

Stream 01 – Příklad 05 Příklad pátý slouží pouze ke zjištění rychlosti zpracování příkladu 04 v RxJavě na větším počtu vláken. V tomto příkladě je počet vláken stanoven na jeden tisíc.

Stream 01 – Příklad 06 Příklad 06 se vrací na hlavní výpočetní vlákno. Zaměřením tohoto příkladu jsou primitivní typy, zde konkrétně `long`.

V cyklu se stejně jako v příkladu 02 mění jen název proměnné. U streamu StreamSupport je opět nutno použít jinou třídu, `LongStreams`, a tentokrát i jinou metodu, `of(...)`.

V případě RxJavy jsem došel ke zjištění, že zpracování tohoto příkladu není možné. RxJava dokáže pracovat pouze s poli objektů, z pole primitivních typů lze vytvořit pozorovaného emitujícího pole pouze jako hodnotu s celým polem.

3.3.2 Srovnání rychlosti zpracování příkladů Stream 01

Pro sadu Stream 01 zbývá uvést orientační srovnání rychlosti zpracování příkladů. Měření jsem provedl vždy minimálně pětkrát a v tabulce je uvedena nejnižší naměřená hodnota. Měření bylo provedeno přístroji se čtyřjádrovým CPU Qualcomm Snapdragon 801 o frekvenci 2,5 GHz s operačním systémem Android verze 5.1.1. Hodnoty jsou uvedeny v milisekundách.

	01	02	03	04	05	06
Cyklus	271	265	304	-	-	4
StreamSupport	285	290	-	114	-	13
RxJava	324	305	343	163	447	-

Chyba jednotlivých měření se pohybuje v řádu desítek ms, po opakovaných měřeních však lze na první pohled označit OOP přístup a streamy StreamSupport za výkonově vyrovnané. RxJava je na tom výkonově o něco hůř, ale nijak výrazně.

Nízké hodnoty u šestého příkladu jsou téměř jistě dány optimalizacemi kompilátoru, díky nimž bylo možno pro zpracování pole primitivního typu `long` využít vektorových instrukcí CPU.

Jak je vidět, rozdíly mezi cyklem a streamem StreamSupport jsou na úrovni chyby měření, RxJava pak zaostává až o několik desítek procent.

Zajímavé je srovnání mezi příklady 03, 04, 05 a 06, tedy zpracováním na hlavním vlákně, na výpočetním vlákně a dvakrát paralelně. Režie pro výpočet na výpočetním vlákně si vyžádala o několik desítek ms navíc. Paralelní zpracování streamy StreamSupport pak trvalo méně než polovinu doby zpracování sekvenčního na hlavním vlákně. V případě RxJavy a příkladu 05 se zpracováním na 10 vláknech bylo zrychlení nižší, ale též se podařilo dosáhnout zrychlení téměř o polovinu. Vysoká paralelizace ovšem výkonu nesvědčí, 1000 vláken způsobilo, že potřebná doba zpracování naopak o zhruba polovinu vzrostla.

3.3.3 Sada příkladů Stream 02

Druhá sada příkladů se věnuje jen knihovnám StreamSupport a RxJava a není měřen ani čas. Cílem této sady je vyzkoušet si další využití streamů a zjistit případné odlišnosti mezi chováním těchto dvou knihoven.

Stream 02 – Příklad 01 Příklad první vyhledává v poli řetězců řetězce o zadané délce a ty spojuje oddělené čárkou do jednoho řetězce. K tomuto účelu slouží v obou knihovnách operátor `filter` a známý `reduce`.

Častý problém při spojování řetězců pomocí cyklu nastává za požadavku na vložení spojovacího řetězce jen mezi dvě spojované hodnoty. První nebo poslední hodnota pak musí být mimo cyklus, případně řetězec dodatečně zkrácen. V případě proudů je ovšem dle očekávání spojovací prvek vkládán pouze mezi spojované prvky, a tak nutnost dalších úprav odpadá. Ukázka je součástí prvního příkladu, ve výsledku část před prvním středníkem.

Důležité je ověřit, co se stane, pokud žádná hodnota podmínce ve filtru nevyhovuje. Takové situaci se věnuje druhá část příkladu. Zatímco knihovna RxJava v této situaci navrátí hodnotu null, v případě knihovny StreamSupport se vyvolá výjimka.

Obě knihovny ovšem nabízí možnost stav chybějícího prvku odchytit a vložit prvek náhradní. U StreamSupport k tomuto účelu slouží operátor `orElse` s náhradním prvkem jako parametrem, přičemž tento operátor nahrazuje operátor `get`. U knihovny RxJava se před operátor `blockingGet` vkládá operátor `defaultIfEmpty`, opět s náhradním prvkem jako parametrem. Výsledek vyhodnocení třetí části se ve výsledku nachází za druhým středníkem. Kód je pak tedy následující:

StreamSupport

```
...
... J8Arrays
        .stream(vStringArray)
        .filter(s -> s.length()==8)
        .reduce((a, b) -> a + ", " + b)
        .orElse("no value");
...
```

RxJava

```
...
... Observable
        .fromArray(vStringArray)
        .filter(s -> s.length()==8)
        .reduce((a, b) -> a + ", " + b)
        .defaultIfEmpty("no value")
        .blockingGet();
...
```

Stream 02 – Příklad 02 Příklad druhý dál ověřuje chování za stavu chybějící hodnoty, tentokrát přímo v poli. Místo jednoho z řetězců v poli jsem vložil hodnotu `null` a zkusil tato data nechat zpracovat ve streamu. Streamy obou knihoven v tomto případě vyhazují výjimku. Zkusil jsem dále jako první operátor po vytvoření streamu vložit operátor `filter` filtrující řetězce

odpovídající hodnotě `null`. V případě RxJavy se opět vyhazuje výjimka, ale StreamSupport již stream dle požadavků zpracoval, výsledek je v aplikaci zobrazen za prvním středníkem. Chování je tedy, dá se říci, opačné od předchozího příkladu, kde výjimku vyhazovala knihovna StreamSupport. Hodnoty `null` knihovna RxJava nepodporuje.

Stream 02 – Příklad 03 Třetí příklad navazuje na příklad druhý a ukazuje, že operace na hodnotách emitovaných do hodnoty `null` u StreamSupport, respektive před touto hodnotu u RxJavy, jsou provedeny. V případě RxJavy je zde zároveň začleněn operátor `defaultIfEmpty`, na němž je demonstrováno, že ani on k vyřešení problému s `null` hodnotami neslouží.

Stream 02 – Příklad 04 Čtvrtý příklad se zabývá zjištěním, že aby byl stream zpracován, nemůže končit libovolným operátorem. Použité příklady opět slouží pouze k demonstraci problematiky, která se reálně projeví spíše výjimečně a až při mnohem složitějších (a smysluplnějších) konstruktech, přesto však považuji za vhodné ji zmínit.

První část příkladu obsahuje pouze operátor `map` s trasformační funkcí složenou z příkazu volání metody pro připojení řetězce emitované hodnoty do společného řetězce. U obou knihoven k vykonání operací streamu nedojde. Druhá část příkladu připojuje operátor `reduce`. Tentokrát dojde k vykonání operací streamu StreamSupport, ale operacestreamu RxJavy vykonány nejsou. Třetí část příkladu připojuje operátor `get`, respektive `blockingGet`. Nyní již dojde k vykonání operací streamu obou knihoven.

Kódy prvních vykonaných streamů jsou tedy následující:

StreamSupport

```
...
J8Arrays
        .stream(vStringArrayShort)
        .map(s -> { ...addString(1, s); return s;})
        .reduce((a, b) -> "");
...
```

RxJava

```
...
Observable
        .fromArray(vStringArrayShort)
        .map(s -> { ...addString(2, s); return s;})
        .reduce((a, b) -> "")
        .blockingGet();
...
```

Stream 02 – Příklad 05 Pátý příklad se zabývá pořadím zpracování emitovaných hodnot při sekvenčním zpracování. Dle očekávání (vzhledem k příkladu 04) výsledky streamů obou knihoven ukazují, že aby byla emitována hodnota další, musí předchozí emitovaná hodnota projít celým řetězcem operací.

3.4 Příklady kategorie Subscription

Tato kategorie příkladů se týká pouze RxJavy a věnuje se vlastnostem registrace pozorovaného k pozorovatelům. Příklady jsou čtyři a jsou si velmi podobné. K pozorovanému jsou vždy registrováni jeden či dva pozorovatelé typu `Consumer` a pozorovatel typu `Observer`. Následně je testováno, zdali jsou zavoláni všichni pozorovatelé a v jakém stavu se po vykonání nachází registrace k pozorovatelům.

Subscription – Příklad 01 První příklad testuje pozorovaného vytvořeného pomocí metody `just`. Ve výsledku můžeme po vykonání příkladu vidět, že došlo k volání všech pozorovatelů a všichni tři pozorovatelé jsou odregistrovaní.

Subscription – Příklad 02 a 03 Další dva příklady mají pozorované vytvořené metodou `fromArray`, respektive `fromIterable`. Chování těchto příkladů je shodné s příkladem prvním.

Subscription – Příklad 04 Čtvrtý příklad využívá místo pozorovaného typu `Observable` pozorovatele `PublishSubject`. Více pozorovatelů opět není problém. O odregistraci je tentokrát dle očekávání nutné se postarat. Po odregistraci pozorovatele metodou `dispose` na při registraci vzniklém objektu `Disposable` již další hodnoty tomuto pozorovateli nejsou zasílány. A vzhledem k pozorovateli typu `PublishSubject` nejsou pozorovateli doručeny ani hodnoty emitované před registrací.

Subscription – Příklad 05 Pátý příklad se zabývá pořadím zpracování emitovaných hodnot při sekvenčním zpracováním. Dle očekávání (vzhledem k příkladu 04) výsledky streamů obou knihoven ukazují, že aby byla emitována hodnota další, musí předchozí emitovaná hodnota projít celým řetězcem operací.

3.5 Příklady kategorie Listener

Kategorie „Listener“ navazuje na předchozí kategorii. V tomto případě testuji za využití knihovny RxBinding 2 chování registrace více (tří) pozorovatelů k pozorovanému typu View systému Android. Příklady tentokrát nejsou číslované ale rozdělené dle způsobu registrace k pozorovanému.

Listener – OOP – Listener První příklad využívá OOP přístup a posluchačů systémového typu `Listener` Androidu. Výsledek ukazuje, že lze mít k jednomu View registrován vždy jen jednoho posluchače, přičemž nově registrovaný nahrazuje předchozího registrovaného.

Listener – Observable Druhý příklad využívá běžné registrace pozorovatele k pozorovanému `Observable`. Na rozdíl od předchozí kategorie příkladů, kde k pozorovanému mohlo být registrováno více pozorovatelů, je v případě pozorovaného vytvořeného knihovnou RxBinding aktivní vždy jen poslední pozorovatel, stejně jako v případě posluchačů OOP.

Listener – Více Observable Třetí příklad testuje vytvoření více pozorovaných k danému View. I tentokrát je chování shodné. Zavolán je tedy pouze poslední registrovaný pozorovatel.

Listener – Sdílený Observable Sdílený pozorovatel `Observable` se chová částečně jako typ `ConnectableObservable`. Zavoláním metody `share` na pozorovaného typu `Observable` nedojde ke kopírování emitovaných hodnot streamu pro jednotlivé pozorovatele. Jak ukazují výsledky tohoto příkladu, nyní již dochází k zavolání všech tří pozorovatelů.

Listener – PublishSubject Posledním testovaným je `PublishSubject`, který zde slouží jako pozorovatel na pozorované View a zároveň jako pozorovaný pro dané tři pozorovatele. Stejně jako v předchozím případě, jsou i tentokrát zavoláni všichni tři pozorovatelé.

3.6 Příklady kategorie Exception

Kategorie Exception se zabývá demonstrací zpracování chyb a výjimek (společná třída `Throwable`, od které výjimky i chyby dědí) na šesti jednoduchých příkladech.

Exception – Příklad 01 První příklad demonstruje odchycení chyby za pomoci metody `subscribe(...)`, která bere jako první argument pozorovatele typu `Consumer` pro běžné zpracování a jako druhý argument opět pozorovatele typu `Consumer`, tentokrát pro zpracování výjimek a chyb. Jako vstupní data proudu slouží pole s dvojicí textových řetězců následovanou hodnotou `null` a opět textovým řetězcem. Při vykonání příkladu jsou první dvě hodnoty zpracovány, hodnota `null` vyvolá zavolání pozorovatele pro zpracování chyby a zpracování proudu je ukončeno bez zpracování posledního řetězce.

Exception – Příklad 02 Druhý příklad je podobný a nahrazuje pozorovatele typu `Consumer` pozorovatelem typu `Observer`. Zpracování chyb má na starosti automaticky metoda `onError(e:Throwable)` observeru. Výsledek vykonání je stejný jako v předchozím případě.

Exception – Příklad 03 Příklad třetí ukazuje pokus o zotavení z chyby za pomoci operátoru `retry`. Pokud při opakovaném pokusu o zpracování emitované hodnoty k chybě či výjimce nedojde, je zpracován celý proud jako by žádná chyba nenastala. V opačném skončí celé zpracování proudu chybou, kterou je nutné odchytit mimo proud.

Exception – Příklad 04 Ošetření chybového stavu v proudu je ve čtvrtém příkladu řešeno operátorem `onErrorReturnItem`. V případě chyby či výjimky je zpracování proudu ukončeno a navrácen objekt či hodnota.

Exception – Příklad 05 Všechna předchozí řešení mají jedno společné, po odchycení chyby již nedojde ke zpracování zbývajících hodnot ve streamu. RxJava bohužel neumožňuje chybu způsobující hodnotu přeskočit. Nezbývá tak, než důkladně analyzovat potenciálně problémové hodnoty a zajistit jejich včasné odchytávání či transformaci. V některých případech je pak možné použít i složené lambda transformace, kde základní transformaci uzavřeme do bloku try-catch a v případě chyby vrátíme například zvolenou výchozí hodnotu, jak je ukázáno právě v tomto pátém příkladu.

3.7 Příklady kategorie RxBus

Poslední kategorie příkladů se zabývá knihovnou RxBus. Příkladům přidružená Android aktivita obsahuje celkem sedm tlačítek. První vytváří instanci třídy mající na starosti příklady RxBus. Druhé a třetí tlačítko pak vytváří instanci producentů (pozorovaných), respektive konzumentů (pozorovatelů), čtvrté a páté se starají o jejich registraci k RxBus a nakonec šesté a sedmé o jejich odregistraci. Všechny anotované metody jsou spouštěny na výpočetním vlákně.

RxBus - Příklad RxBus1 První příklad se týká metody `produce1()` producenta a metody `consume1(...)` konzumenta, propojených anotacemi s tagem „RxBus1“. Po vytvoření a registraci producenta a konzumenta dojde k emitaci hodnoty metodou `produce1()`, kterou následně odchytí metoda `consume1(...)` a společně s UNIX time časem vypíše.

RxBus - Příklad RxBus2 Následující příklad testuje jednoho producenta a dva konzumenty. Konkrétně jde o metodu `produce2(...)` konzumenta a me-

tody `consume2(...)` a `pconsume2(...)` konzumenta, tag je „RxBus2“). Jak je vidět pod popiskem příkladu, zavoláni jsou oba konzumenti.

RxBus - Příklad RxBus3 Třetí příklad využívá též metody konzumenta `pconsume2(...)` a testuje možnost použití více tagů. Producentem je metoda `produce3()`. Předchozí příklad proběhl úspěšně a zavolán byl i konzument uvedený druhým tagem pro tento příklad („RxBus3“).

RxBus - Příklad RxBus4 Čtvrtý příklad navazuje na předchozí dva příklady a zkoumá možnost řetězení použitím anotace konzumenta i producenta. Takové využití ovšem není možné, spojení definované tagem `RxBus4` je ignorováno.

RxBus - Příklad RxBus5 Pátý příklad využívá taktéž metody konzumenta `pconsume2(...)`, která mimo jiné vytváří instanci třídy `RxBusObject`. Tato třída obsahuje producenta `oproduce5()` využívajícího tag `RxBus5`. Tentokrát k zaslání zprávy dojde a metoda `consume5(...)` konzumenta tak vypíše výsledek.

Během zpracování pátého příkladu (velmi pravděpodobně) dojde k chybě, kterou oznámí zobrazený Toast. Ta je dána tím, že je metoda `pconsume2(...)` zavolána dvakrát a zpracování probíhá paralelně. RxBus se totiž neregistruje k objektu, ale ke třídě, a ta může být registrována pouze jednou. Anotované metody jsou pak zavolány výhradně po registraci třídy k RxBus. Toto chování do značné míry omezuje možnost použití této knihovny. Dalším zjištěným omezením je nemožnost detekce, zdali je třída RxBus registrována jinak, než pomocí výjimek. Metoda navracející informaci o stavu registrace v sestavené knihovně není. A posledním zjištěným omezením je nemožnost registrace k RxBus z výpočetního vlákna, pro registraci je tedy nutné přepnout se na vlákno hlavní.

3.8 Shrnutí prvních zkušeností s FRP

Získané zkušenosti s užitím funkcionálně reaktivního programování nejprve popíšu strukturovaně v odstavcích částečně odpovídajících jednotlivým kategoriím a následně provedu celkové shrnutí.

Streamy V kombinaci s lambda výrazy se streamy ukázaly jako velmi silný nástroj pro hromadné zpracování dat. Použití streamů je intuitivní a urychluje zpracování dat. Jak knihovna StreamSupport, tak knihovna RxJava, nabízí široké množství různých operátorů. Objektově orientované programování má nevýhodu v objemnějším kódu, funkcionálně reaktivní programování je naopak velmi minimalistické. Na druhou stranu ovšem v případě početnějších řetězců

operátorů trpí menší přehledností. To se však dá poměrně snadno změnit řazením operátorů pod sebe a zejména pro ony početnější řetězce operátorů dopisováním vysvětlujících komentářů za jednotlivé transformace dat. Knihovna StreamSupport se hodí pro zpracování malých objemů dat a nenáročných transformačních operací na hlavním vlákně a naopak též pro vysoce paralelizované zpracování větších objemů dat s náročnějšími transformacemi. Knihovnu RxJava je pak vhodné použít postačí-li nám či potřebujeme-li sekvenční zpracování na výpočením či jiném vlákně. Menším úskalím knihovny RxJava je nutnost zamezení přítomnosti `null` hodnot ve vstupních datech. U knihovny StreamSupport lze tuto hodnotu v proudu snadno filtrovat. Knihovna RxJava se ukázala jako o něco pomalejší, než knihovna StreamSupport i objektově orientovaný přístup, ale nejde o zásadní rozdíl, a tak ve většině případů nasazení pravděpodobně nebude vadit.

Reaktivní programování s RxJava Stejně jako v případě streamů je knihovna RxJava dobře použitelná i v reaktivním programování za použití pozorovatelů. Je třeba si dát pozor na odlišnost chování registrace k pozorovateli vytvořenému z Android View pomocí knihovny RxView, který pro běžného pozorovaného registruje vždy jen jednoho pozorovatele.

Za negativum RxJavy považuji způsob ošetření výjimek, které se musí hlídat v každém potenciálně problémovém operátoru. RxJava sice nabízí různé operátory pro ošetření chybových stavů, nenabízí ovšem možnost chybovou hodnotu přeskočit. V případě, že se chyba vyskytne či se hodnotu opakovaným pokusem nepodaří úspěšně zpracovat, tak dojde vždy k ukončení emitace dalších hodnot pozorovatelem.

Knihovna RxBus Tato doplňující knihovna ve mně zanechala velmi rozpačité dojmy. Omezující zjištění popsaná u pátého příkladu považuji za podstatná. Knihovnu RxBus je tak vhodné využít zejména v aplikacích s jednorázovým spouštěním metod složitějšího asynchronního systému. Jiné použití je možné, ale maže výhody, které má knihovna RxBus přinášet, tedy snadnost použití a přehlednost.

Testování a ladění V příkladech jsem se nevěnoval testování a ladění aplikace při použití FRP. Nyní shrnu prozatímní zkušenost s laděním a dále se testování a ladění budu věnovat v kapitole 6, týkající se již realizace a použití FRP v trackovací aplikaci.

Z implementace těchto příkladů vyplynula co se týče ladění, složitější orientace ve výpisu trasovacích informací chyby či výjimky. Do projektu do balíčku testů (pro snadné spuštění) jsem umístil ještě jeden příklad. Třída v hlavním balíčku slouží jako zdroj hodnot, který zároveň odchytává chyby a výjimky a třída v podbalíčku `consumer` tyto hodnoty zpracovává. Jak lze v příkladu vidět, je trasovací řetězec v případě OOP přímo odpovídající průchodu dat

implementovaným kódem, zatímco v případě FRP je zdrojové a cílové místo proloženo dalšími zhruba deseti průchodovými místy knihovny RxJava.

Již jsem doporučil řazení operátorů streamu pod sebe z důvodu přehlednosti a snadnější dokumentace. Toto doporučení je posíleno i snadnějším laděním, neboť trasovací informace vypisují výskyt chyby na daném řádku a v případě řazení operátorů pod sebe tak lze snadno identifikovat zdroj problému.

Kapitola 4

Základní návrh aplikace

4.1 Základní funkcionalita

Hlavní funkcionalitou bude vykreslování aktuálně projížděné trasy na mapovém podkladu a zobrazování informací o trase. Mapové podklady budou pocházet minimálně ze dvou zdrojů, z nichž alespoň jeden bude načítaný online a druhý bude využívat uložených mapových podkladů v úložišti zařízení, mezi nimiž bude moci uživatel i během trekování přepínat. Po ukončení trekování, na základě pokynu uživatele, bude uživateli nabídnuta možnost exportovat právě projetou trasu do souboru ve formátu GPX. Mapové podklady ukládané do zařízení budou moci být stahovány prostřednictvím aplikace z přednastaveného zdroje. Podklady bude možné z aplikace aktualizovat i mazat.

Kromě trekování nové trasy bude uživateli umožněno načíst GPX soubor a nechat vykreslit na mapu v něm uloženou trasu, podle které se bude uživatel moci na kole vydat. Aplikace bude pro tuto funkcionalitu obsahovat jednoduchý algoritmus detekce progrese ve sledování trasy. Mimo možnosti uložit trasu do GPX souboru budou základní informace o jednotlivých jízdách ukládány do databáze a následně budou moci být uživatelem zobrazeny ve formě souhrnné statistiky.

Aplikace bude uživateli nabízet možnost si chování aplikace částečně upravit a bude připravena pro snadnou lokalizaci do různých světových jazyků. Na základě nastavení budou údaje zobrazovány v metrické anebo imperiální soustavě.

4.2 Schéma aplikace

Prvotním bodem aplikace bude úvodního menu, které bude obsahovat položku vedoucí na hlavní trekovací obrazovku. Následovat bude položka pro výběr trasy pro sledování, a poté položky vedoucí na obrazovku s jednoduchou statistikou najetých jízd, nastavení, včetně správy offline mapových podkladů, a jako poslední položka pro ukončení aplikace.

Trekovací obrazovka bude tvořena informačním panelem na vrchu obrazovky, pod kterou bude umístěna mapa s ovládacími prvky. Vrchní informační panel bude dynamicky zobrazovat informace o aktuálním stavu trasy, konkrétně aktuální rychlost, průměrnou rychlost, ujetou vzdálenost, čas a případně údaje o sledované trase – její název a délku zbývající trasy k projetí. Mapové podklady budou zobrazovány prostřednictvím mapové knihovny. Ovládací prvky umístěné ve vrstvě nad mapou budou sloužit k zahájení, pauze a ukončení trekování, dále zde budou umístěny prvky pro přiblížení a oddálení mapy a také prvek pro přepínání mezi mapovými podklady.

Funkce statistika bude realizována tabulkou se základními statistickými údaji o najetých trasách, jako je počet ujetých tras, celková najetá vzdálenost či celkový strávený čas během jízd. Statistické údaje bude možné tlačítkem nechat vymazat. Sekce nastavení bude dělena do tří částí. V první bude možné upravit chování aplikace, jako je výchozí zobrazovaný mapový podklad, výchozí přiblížení mapy, měrný systém a další.

Kapitola 5

Realizace

5.1 Úvod

V průběhu realizace se mi postupně podařilo implementovat veškerou funkcionalitu uvedenou ve výčtu v návrhu. Taktéž schéma průchodu aplikace odpovídá navrhovanému. Oproti návrhu byly doplněny a rozpracovány další funkce a obrazovky, aby aplikace dosahovala vysoké míry reálné použitelnosti uživateli v praxi.

V této kapitole se budu věnovat převážně obecnému popisu realizovaných funkcí a implementačním detailům shodným pro objektově orientovanou i funkcionálně reaktivní verzi. Popis implementačních rozdílů mezi verzemi budu popisovat především v další kapitole. Funkčně jsou obě verze shodné.

5.2 Struktura zdrojových souborů

Struktura zdrojových souborů vychází ze struktury běžného Android projektu generovaného vývojovým prostředím Android Studio. V popisu struktury se tedy budu věnovat struktuře funkčního zdrojového kódu, tedy adresáři `app/src/main/java`.

V případě funkcionálně reaktivní verze se ve stromové struktuře projektu aplikace objevuje navíc balík `com.jakewharton`, který obsahuje soubory jedné z využitých funkcionálně reaktivních knihoven, která je na rozdíl od zbylých včleněna nesestavená.

Výchozí balík, respektive adresář, kódu realizované aplikace obsahuje třídy pro jednotlivé implementované Android aktivity a soubor `Config.java`, což je soubor obsahující pouze pole se statickými hodnotami konfigurujícími některé vlastnosti aplikace.

Obě verze aplikace obsahují dále následující podbalíky:

adapters — Vlastní adaptéry pro vertikálně posunovatelné seznamy ListView

consumers	Konzumenti (Consumer) pro pozorované (Observable); jen v FRP verzi
database	Třídy pro práci s databází, podbalík dao obsahuje třídy pro uložení respektive načtení jednotlivých typů objektů
enum	Výčtové typy
environments	Důležitý balík tříd sloužících k uchovávání aktuálního stavu aplikace
exceptions	Vlastní třídy výjimek
extenders	Jednoduché rozšiřující třídy pro některé Android třídy
fragments	Třídy fragmentů, konkrétně třídy fragmentů mapových podkladů a dále třídu fragmentu zobrazující podrobnosti o trase
handlers	Třídy, které zajišťují pokročilou funkcionalitu, typicky tu, která pracuje s nějakým vstupem či výstupem
helpers	Taktéž třídy zajišťující pokročilou funkcionalitu, ale typicky tu, která pracuje s vloženými parametry. Většina metod obsažených v těchto třídách je statická.
interfaces	Obsahuje vlastní rozhraní
items	Třídy pro objekty objevující se ve větším množství. V nadpoloviční většině se jedná o čisté data holdery.
services	Třídy pro funkce běžící na pozadí
viewmodel	Třídy modelů obsahujících View přiřazené určité Android aktivitě; jen v FRP verzi, v OOP verzi jsou View modely vnořené třídy Android aktivit

5.3 Mapové podklady

5.3.1 Použité mapové podklady a knihovny pro práci s nimi

Aplikace využívá tří knihoven, respektive zdrojů, pro mapové knihovny. Jedná se o následující mapové podklady a příslušné knihovny pro ně:

- online mapy Google Maps, knihovna pro mapy z balíku knihoven Google Play Services
- online mapy OpenStreetMap, knihovna OSMDroid
- offline mapy Mapsforge, knihovna Mapsforge pro Android

5.3.2 Obecná práce s mapovými knihovnami

Způsob práce s vybranými mapovými knihovnami se mírně liší, ale všechny lze začlenit do aplikace poměrně rychle a snadno.

Ke Google Maps se v nadřazené Android aktivitě přistupuje jako k prvku Fragment, který je založený na třídě `MapFragment`. OSMDroid a Mapsforge používají vlastní View, které lze začlenit přímo do layoutu aktivity, byť z důvodu přehlednosti a strukturovanosti je obvykle lépe užít taktéž Fragment.

Google Maps mapy jsou inicializovány asynchronně přes metodu `getMapAsync(onMapReadyCallback:OnMapReadyCallback):void` se zpětným voláním metody `onMapReady(googleMap : GoogleMap):void`. U OSMDroid a Mapsforge se k mapě přistupuje přes běžné načtení View metodou `findViewById(id:int):View` Android aktivity nebo Fragmentu.

Nejvíce se ale samozřejmě odlišuje knihovna Mapsforge, kde je třeba na začátku inicializovat import a zobrazení mapových podkladů z vybraných souborů. Konkrétně je potřeba nejdříve vytvořit cache pro jednotlivé mapové díly a vytvořit strukturu obsahující záznamy o jednotlivých mapových souborech. Následně je třeba vytvořit novou zobrazovací vrstvu a tuto přidat do mapového View.

Mezi knihovnami se liší například i některá pojmenování prvků či rozhraní API. Například k uložení souřadnic používá Google Maps knihovna třídu `LatLng`, MapsForge třídu `LatLong` a OSMDroid třídu pojmenovanou `GeoPoint`.

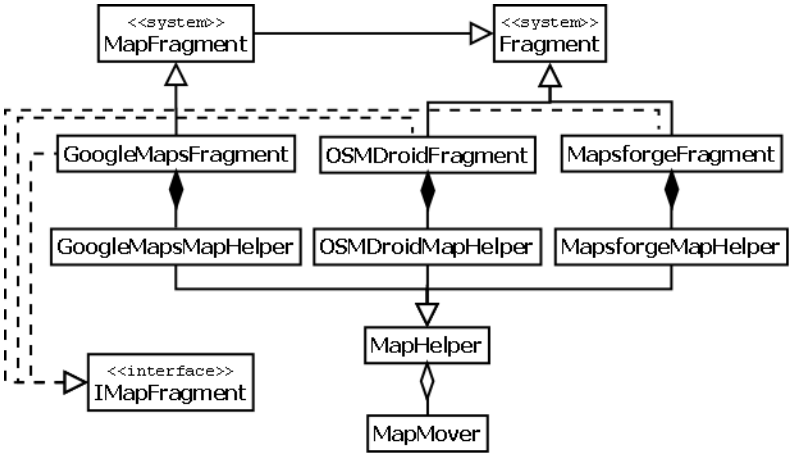
Všechny tři použité knihovny poskytují rozhraní pro přidávání křivek k vyznačení trasy nad zobrazovanou mapu. Omezením je možnost přidávání pouze rovných křivek mezi každými dvěma zvolenými body.

5.3.3 Práce s mapovými knihovnami v realizované aplikaci

Všechny tři použité mapové zdroje jsou používány prostřednictvím Fragmentu. Jelikož se map od Google dědí od třídy `MapFragment`, zatímco u ostatních od třídy `Fragment`, nebylo možné vytvořit společného předka pro shodnou funkcionalitu. Z tohoto důvodu obsahuje každý Fragment map vnořenou rozšiřující třídu třídy `MapHelper`, jejíž některé metody implementuje. Částečnou strukturu tříd zajišťujících práci s mapami zobrazuje diagram 5.1.

5.3.4 Práce s offline mapovými podklady v aplikaci

Mapové podklady Mapsforge načítá a ukládá aplikace z/do adresáře `%externí-úložiště-zařízení%/GPSTracking/Mapsforge`. Oficiálním zdrojem pro Mapsforge mapy je internetová adresa `http://download.mapsforge.org`, avšak aplikace využívá ve výchozí konfiguraci zdroj `http://android.voparil.cz/-gpstracking/mapsforge`. Soubor ve formátu JSON s názvem `available_maps.json` obsahuje seznam dostupných map, které lze stáhnout a pro které je po stažení z tohoto souboru následně vygenerován popisný JSON soubor,



Obrázek 5.1: Zjednodušený diagram základních tříd pro práci s mapami

který je pro použití dané mapy v realizované aplikaci potřebný. Struktura tohoto JSON souboru je popsána v příloze.

Aplikace v sekci věnované správě mapových podkladů zobrazuje seznam nainstalovaných a seznam online dostupných podkladů. Při každém načtení této sekce dojde ke zkopírování zmíněného `available_maps.json` souboru z online zdroje a seznam zobrazených dostupných podkladů je aktualizován. Graficky po vzoru semaforu jsou barevně zvýrazněny podklady nainstalované a aktuální (zeleně), nainstalované, ale zastaralé (žlutě), a nenainstalované (červeně).

5.4 Vykreslování trasy na základě údajů z GPS

5.4.1 Zobrazování trasy na mapě

Ve chvíli, kdy aplikace obdrží novou GPS pozici, dojde k jejímu zaznamenání do objektu třídy `TrackEnvironment` a následně ke zpracování objektem dědícím od třídy `MapHelper`, který vezme dvě poslední souřadnice, zjistí barvu pro tento segment a nechá vykreslit další křivku. Zatímco příprava informací pro vykreslení nové křivky je pro všechny tři mapové zdroje společná, samotné vykreslení křivky zajišťuje metoda implementovaná v potomku třídy `MapHelper`, kterého má každý mapový zdroj vlastní.

Kromě metody pro průběžné přidávání nových segmentů trasy obsahuje třída `MapHelper` metodu, která vykreslí všechny dostupné segmenty. Tato metoda je volána v případě, kdy uživatel změní mapový zdroj a také v případě, kdy uživatel opustí a následně znovu otevře trekovací Android aktivitu.

5.4.2 Dynamický posun mapy

Skokový posun mapy do středu aktuální lokace mi přišel příliš trhaný, proto jsem do aplikace implementoval mechanismus, který posun mapy zjemňuje. O to se stará objekt třídy `MapMover` jako na pozadí běžící proces, který pravidelně každých 20 milisekund kontroluje, zdali se aktuální střed mapy daný posledním posunem shoduje s aktuální zaznamenanou GPS pozicí. Pokud nikoli, provede se 20 posunů mapy po 20 milisekundách o 1/20 rozdílu mezi původním středem mapy a zaznamenanou GPS pozicí. Skokový posun se provádí pouze v případech, kdy vzdálenost mezi středem mapy a GPS lokací je nejvýše 1 m anebo naopak nejméně 1 km.

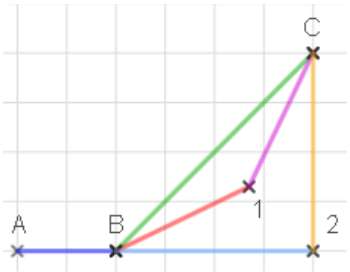
Spouštění a zastavování procesu `MapMover` je obstaráváno prostřednictvím třídy `MapHelper`. Proces `MapMover` běží pouze, je-li některý z mapových zdrojů zobrazen na obrazovce.

5.4.3 Výpočet barvy pro zaznamenaný úsek

Jak je již několikrát zmíněno, křivka zaznamenávané trasy má být barevně proměnlivá na základě rychlosti v daném úseku. Výpočet průměrné rychlosti v úseku je jednoduchý, neboť pro každou zaznamenanou GPS souřadnici je zaznamenáván i čas a výpočet vzdálenosti mezi dvěma souřadnicemi je součástí Android API, konkrétně třídy `Location` a její metody `distanceBetween`. Tato metoda bere za parametry zeměpisné souřadnice dvou lokací a do poslední proměnné určené jako parametr uloží výsledek odpovídající vzdálenosti v metrech. Stačí tedy pomocí této metody získat vzdálenost a tu podělit rozdílem časů.

Nyní je tedy třeba pro danou rychlost určit také odpovídající barvu. Pro barevný přechod byla zvolena stejná škála, jaká se používá běžně například u teploty, tedy pro nízké rychlosti modrá, postupný přechod přes zelenou k žluté, poté přes oranžovou k červené a nakonec tmavnutí až k černé pro nejvyšší rychlost. Implementace tohoto přechodu znamenala rozdělit stanovené rozmezí rychlosti na čtyři úseky, z nichž v první se v barevném prostoru RGB postupně snižuje z maximální hodnoty modrá složka a naopak navyšuje zelená složka, v druhé se navyšuje červená složka, ve třetí se snižuje zelená složka a ve čtvrté se snižuje červená složka.

Ve výchozím nastavení je určeno, že pro každý 1 m/s se barevná hodnota změní nějakým způsobem o 50, což pro jeden zmíněný úsek barevné změny znamená 5,1 m/s a celkově 20,4 m/s, neboli barevné rozlišení vyznačuje rychlost až do hodnoty 73,44 km/h.



Obrázek 5.2: Ilustrace problematiky optimalizace trasy - Představme si, že mobilní zařízení zaznamená na základě údajů z GPS body A, B a následně C. Ve skutečnosti však v případě větších vzdáleností mezi body mohl cyklista jet jinak. Problematické je zejména odbočení do C, kdy cyklista mohl odbočit až v bodě 2 či dále. Navržený algoritmus se snaží odbočení optimalizovat přidáním dalšího bodu, 1.

5.5 Funkce optimalizace trasy

5.5.1 Úvod do problematiky

Pro zařízení či místa se slabším signálem GPS, kde jsou mezi jednotlivými zaznamenanými lokacemi větší vzdálenosti, jsem se pokusil implementovat algoritmus, který kompenzuje změnu směru trasy. Tato optimalizace je ve výchozím stavu zapnutá a lze ji vypnout v nastavení aplikace.

Představme si situaci, kdy nám údaje z GPS ukazují trasu v určitém směru, která následně odbočuje mimo tento směr. Pokud jsou vzdálenosti mezi zaznamenanými lokacemi malé, bude přesnost vykreslené trasy vysoká. Pokud jsou ovšem vzdálenosti větší, nevíme, zdali cyklista ve skutečnosti nezměnil směr až později. Navržený algoritmus má částečně toto kompenzovat. Pro lepší ilustraci problematiky přidávám obrázek 5.2.

5.5.2 Popis algoritmu pro optimalizaci trasy

Implementovaný algoritmus se snaží optimalizovat trasu přes vloženou souřadnici, která leží v prostoru mezi polopřímkou vedenou z předpředchozí (na obrázku 5.2 bod A) přes předchozí lokaci (bod B) a polopřímkou vedenou z lokace předchozí (bod B) a aktuálně zaznamenanou (bod C). Přesná poloha tohoto bodu je určena na základě požadavku, že úhel odbočení u předchozí lokace a u vloženého bodu budou shodné.

Aby optimalizace dávala smysl, nesmí být úhel odbočení k vloženému bodu větší než (180°- úhel odbočení k nové lokaci z GPS) a zároveň musí být větší než polovina této hodnoty. Aby úhel svíraný mezi úsečkou z předchozí do aktuální lokace a úsečkou z vložené lokace do aktuální lokace nebyl příliš velký, držel jsem se spíše nižší zmíněné hranice, konkrétně 60 % vyšší hodnoty. Grafické znázornění je na obrázku 5.3.

5.5.3 Výpočet souřadnice optimalizačního bodu

Nejprve provádím výpočet svíraného úhlu odbočení v trojúhelníku mezi přepředchozí, předchozí a aktuální lokací (na obrázku 5.3 úhel alfa v trojúhelníku ABC) pomocí kosinové věty. Poté stanovím úhel gamma jako násobek hodnoty (180°- alfa), konkrétně 0,6 násobek. Následně vypočítám úhly v trojúhelníku mezi předchozí a novou lokací a vloženým bodem - svíraný úhel u předchozí lokace má hodnotu (180°- alfa - gamma), svíraný úhel u vloženého bodu hodnotu (180°- gamma) a svíraný úhel u nové lokace je 180°mínus dva předešlé úhly.

Pomocí sinové věty aplikované na vzdálenost mezi předchozí a novou lokací a předešlé úhly zjistím vzdálenosti mezi předchozí lokací a vloženým bodem a mezi vloženým bodem a novou lokací.

Poté zjišťuji, zdali úhel odbočení vůči zeměpisným souřadnicím přičítám nebo odčítám. Vypočítám úhel natočení vektoru mezi předpředchozí a předchozí souřadnicí a vektoru mezi předchozí souřadnicí a novou lokací a od druhého úhlu první odečtu. Pokud úhel leží v prvních dvou kvadrantech, budu přičítat, pokud v druhých dvou, budu odčítat. Nyní je potřeba vypočítat konečné souřadnice, které získám vynásobením vzdálenosti mezi předchozí lokací a vloženým bodem a vynásobením kosinem úhlu natočení pro zeměpisnou šířku, respektive sinem úhlu natočení pro zeměpisnou délku a přičtením výsledných hodnot k souřadnicím předchozí lokace.

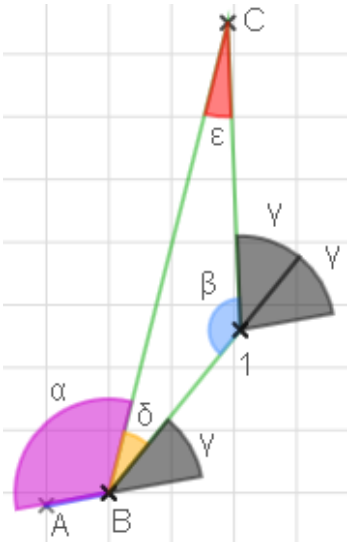
Zbývá určit čas a nadmořskou výšku vloženého bodu, tu vypočítám rozdílem časů a nadmořských výšek mezi předchozí a vloženou lokací a jejich vynásobením poměrem mezi vzdáleností z předchozí lokace do vloženého bodu a celkovou vzdáleností z předchozí lokace do nové lokace přes vložený bod a přičtením těchto hodnot k hodnotám času a nadmořské výšky předchozí lokace.

5.6 Export trasy do souboru ve formátu GPX

5.6.1 Představení formátu GPX

Formát GPX je široce užívané XML schéma určené pro ukládání GPS dat reprezentujících jednotlivé významné body (waypoint(s)), plány cest (route(s)) a záznamy cest (track(s)) 5.4. Dostupné jsou dvě verze schématu, 1.0 a 1.1, ovšem dnes lze i vzhledem ke stáří vyšší verze, která byla vydána již v roce 2004, počítat pouze s touto verzí. Domácí internetovou adresou formátu GPX je `http://www.topografix.com/gpx.asp`, kde lze nalézt příslušné schéma i dokumentaci.

Kořenovým XML elementem GPX souboru je dle schématu element `gpx` s atributy vyjadřujícími odkaz na schéma, verzi schématu a tvůrce datového souboru. Vnořenými elementy jsou elementy pro některé ze zmiňovaných tří druhů záznamu, přičemž pro realizovanou aplikaci užívám pouze elementů pro

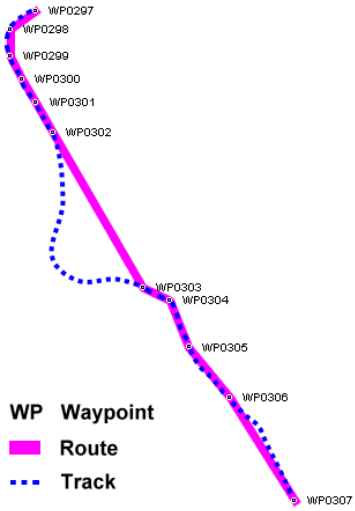


Obrázek 5.3: Úhlová situace optimalizace trasy - Úhel odbočení u předchozí zaznamenané lokace (bod B) a úhel odbočení u vložené optimalizační lokace (bod 1) odpovídají úhlu gamma. Úhel gamma nesmí být větší než (180°- úhel alfa), zároveň ale musí být větší než polovina této hodnoty, jinak by nebylo možné aktuální lokace (bod C) dosáhnout. Úhel gamma je z mého pohledu vhodnější volit spíše blíže k nižší hranici, aby úhel epsilon nebyl příliš velký.

záznam cesty (track). Každý takový element může mít vnořený element `name` s pojmenováním záznamu. Záznam cesty lze následně dělit na segmenty, ale tato možnost není využita. Každý soubor, který generuji tedy obsahuje mimo jiné právě jeden element záznamu cesty a právě jeden podelement segmentu záznamu cesty. Segment cesty již obsahuje jednotlivé záznamy s GPS, přičemž zeměpisná šířka a délka je uvedena prostřednictvím atributu a nadmořská výška a čas prostřednictvím vnořeného elementu.

5.6.2 Export trasy do souboru

Po ukončení trekování je uživatel dialogovým oknem vyzván k udání názvu trasy a potvrzení exportu trasy do souboru. Název trasy bude uveden jak v samotném souboru GPX, tak bude tvořit název souboru před příponou `.gpx`. Pokud daný soubor neexistuje nebo uživatel potvrdí jeho přepsání, započte export trasy. Ten začíná předáním aktuálního objektu třídy `TrackEnvironment` se seznamem zaznamenaných lokací třídě `GPXExport`. Tato třída vytvoří z dodaných dat a na základě GPX XML schématu nejdříve vnitřní programovou reprezentaci XML stromu, poté ji rekurzivně převede do výsledného textového řetězce a ten nakonec předá třídě `FileHandler`, která se postará o uložení do definovaného souboru.



Obrázek 5.4: Formát GPX - Rozdíl mezi bodem (waypoint), plánem cesty (route) a záznamem cesty (track). Obrázek získán z [15]

5.7 Načtení trasy z GPX souboru a její sledování

5.7.1 Úvod

Sledování trasy je rozšířením běžného záznamu nové trasy. Uživatel si může načíst soubor se zaznamenanou trasou ve formátu GPX, která se následně v trekovací Android aktivitě vykreslí na mapový podklad. V oblasti s informacemi o trase pak vidí průběžně se aktualizující informaci o tom, jaká vzdálenost zbývá do konce dané trasy.

5.7.2 Výběr GPX souboru

Výchozím adresářem pro výběr GPX souboru je `%externí-úložiště-zařízení%/GPSTracking/Tracks`. Uživatel se pomocí jednoduchého adresářového průzkumníka může proklikat do jiného adresáře. Za dostupné soubory s trasou jsou považovány pouze ty s příponou `.gpx`, a pouze ty jsou v nabídce uživateli zobrazeny.

5.7.3 Načtení GPX souboru

Zatímco při exportu dat do GPX souboru nevyužívám žádného pomocného rozhraní, používám při importu dat XML parseru `XmlPullParser`, který je součástí Android API. Celý proces importu dat, který probíhá přes třídu `GPXImport`, je velmi jednoduchý. Nejdříve vytvořím prázdnou kolekci pro seznam lokací, následně si u třídy `FileHandler` vyžádám obsah souboru, inicializuji XML parser a nakonec v cyklu nechám detekovat nalezené XML elementy,

a ty požadované zpracuji. Jelikož nadmořská výška a čas jsou ve vlastním XML elementu, jsou u otevíracího tagu příslušného bodu trasy zaznamenány hodnoty pro zeměpisnou šířku a délku, v dalším kroku doplněny hodnoty nadmořské výšky a času a nakonec u zavíracího tagu jsou tyto hodnoty přidány jako nová lokace do seznamu lokací. Ze souboru se postupně načítají veškeré záznamy lokací tras nezávisle na trase a na segmentu trasy. Po zpracování celého souboru je kolekce lokací navrácena Android aktivitě pro načítání trasy, a ta ji následně předá v parametru nově spouštěné Android aktivitě pro trekování.

5.7.4 Vykreslení sledované trasy

Vykreslení sledované trasy probíhá stejným způsobem jako vykreslení zaznamenávané trasy, akorát je celá trasa vykreslená v jednotné šedé barvě. Při postupu v trase se postupně úseky překreslují ze šedé barvy na šedozelenou.

5.7.5 Sledovací algoritmus

Pro detekci postupu v trase, který umožňuje zmíněné překreslování sledované trasy a také informování cyklisty o zbývající vzdálenosti do konce trasy, bylo potřeba navrhnout algoritmus. K tomuto účelu jsem vymyslel systém, který porovnává aktuální zaznamenané lokace projížděné trasy s lokací sledované trasy naposledy určenou jako dosaženou a pěti lokacemi následujícími (pokud jsou). Taková lokace sledované trasy ze zmíněných šesti, která se nachází ve vzdálenosti menší než 12 metrů od zaznamenané lokace a zároveň ta, která je blíže této lokaci, je označena za dosaženou. Pokud uživatel dosáhne poslední lokace ve sledované trase, je uživateli automaticky zobrazen dialog pro uložení právě projeté trasy.

5.8 Sekce statistika a práce s databází

5.8.1 Popis sekce statistika

Sekce statistika zobrazuje jednoduchou statistiku najetých jízd, konkrétně počet projetých tras, celkovou ujetou vzdálenost, celkový čas na cestách (vyjma pauz), délku nejdelší trasy a nejdelší čas strávený na cestě (taktéž po odečtení pauz). Statistiku je možné stisknutím tlačítka vymazat. Pro ukládání záznámů statistiky využívám v Androidu vestavěnou databázi SQLite.

5.8.2 Práce s databází

Databázové třídy jsou uloženy v balíku `database`. Základní třídy jsou tři a pak je pro každý ukládaný typ objektu (v současném stavu aplikace je to jen záznam pro statistiku) vlastní třída v podbalíku `dao` (zkratka Database access object).

První základní třída `DbSimpleScheme` nezahrnuje žádnou funkční logiku a slouží k uložení databázových jmenných prvků. Pro databázovou tabulku je v této třídě vytvořena vlastní podtřída a v ní jsou obsažena konstantní pole s názvem tabulky a okomentovanými názvy sloupců tabulky. Tato pole jsou pak užita jak v následující zmíněné třídě, tak v příslušné třídě ukládající k tabulce relevantní objekt do databáze.

Další třídou v balíku `database` je třída `DbHelper`, což je implementace abstraktní systémové třídy `SQLiteOpenHelper`, sloužící k vytváření a aktualizaci databázových tabulek, jak je již popsáno v části textu věnující se perzistenci dat na Androidu. Pokud databáze ještě nebyla vytvořena, metoda `onCreate(database:SQLiteDatabase)` sestaví a vykoná příkaz pro vytvoření tabulky statistiky a metoda `onUpgrade(database:SQLiteDatabase, oldVersion:int, newVersion:int)` aktualizuje existující tabulku, pokud se její struktura v novější verzi aplikace změní (pokud se zvýší udané číslo verze databáze).

Poslední třídou je `DbProvider`. Tato třída se stará o vytvoření, otevření a uzavření spojení do databáze a předává instanci databáze volajícímu objektu. Instance této třídy, potažmo spojení s databází, existuje v jednom okamžiku nejvýše jediná. Využit je návrhový vzor jedináček (singleton) a synchronizace požadavků na přístup a ukončení spojení. Instance obsahuje čítač, který sleduje požadavky na přístup k objektu `DbProvider` a požadavky na ukončení spojení, a pokud dosáhne hodnoty nula, je databáze uzavřena a dojde k odhlášení této instance.

Třídy pro ukládání objektů v podbalíku `dao` dědí od abstraktní třídy `AbstractDAO`, která poskytuje metody pro komunikaci s `DbProvider` a databází a několik dalších pomocných metod.

5.9 Nastavení aplikace

5.9.1 Sekce nastavení

Sekce nastavení se dělí na tři základní části, obecné nastavení, správu offline Mapsforge map a informace o aplikaci. Obecné nastavení slouží k úpravě chování a výchozích nastavení některých funkcí aplikace a bude více rozepsáno dále. Správa Mapsforge map byla již popsána v části textu věnující se mapám. Informace o aplikaci pak poskytují základní informace o realizované aplikaci, konkrétně verzi aplikace, datum vydání dané verze, autora a informace o licencích užitých děl.

5.9.2 Obecné nastavení

Podsekce Obecné nastavení používá k uložení hodnot službu Androidu s názvem sdílené nastavení (SharedPreferences), popsané již podrobněji v části o perzistenci dat. Aktuálně obsahuje tyto možnosti nastavení:

- Výchozí mapy (výchozí mapový zdroj), lze je přepnout v trekovací Android aktivitě. Možnosti jsou Google mapy, OpenStreetMap, Mapsforge mapy.
- Mapový podklad OpenStreetMap, možnosti jsou cyklistická mapa a Mapnik.
- Měrný systém, možnosti jsou metrický a imperiální.
- Úroveň přiblížení mapy, možnosti 11, 13, 15, 17 a 19, odpovídající úrovním přiblížení mapových podkladů, u OpenStreetMap se tato hodnota mírně liší, proto je použit srovnávající offset o hodnotě 2.
- Interval žádostí o novou GPS lokaci, možnosti 500 ms, 1 000 ms, 1 500 ms, 2 000 ms a 3 000 ms.
- Šířka vykreslené čáry trasy, možnosti úzká, normální, široká.
- Optimalizace trasy, buď zapnuto, nebo vypnuto.
- Automatické připojování aktuálního data k zadanému názvu trasy, buď zapnuto, nebo vypnuto.
- Zobrazování potvrzujících dialogů, buď zapnuto, nebo vypnuto. Pokud je zobrazování potvrzujících dialogů vypnuto, jsou veškeré akce, jako například mazání map či statistik, provedeny přímo bez potvrzení.

Vyjma nastavení optimalizace trasy a nastavení intervalu žádostí o novou GPS lokaci se všechna nastavení projeví ihned po opuštění obrazovky obecného nastavení, a to i během aktivního trekování.

5.10 Služba pro zobrazování hromadných Toastů

Jak již bylo zmíněno v popisu Androidu, zobrazuje se vždy maximálně jeden informační dialog Toast a případné další čekají ve frontě. To může znamenat značnou prodlevu a uživateli se zobrazují infomace, které již nemusí být pravdivé. V realizované aplikaci například zobrazuji informaci o zdroji právě zvoleného mapového podkladu. Uživatel mezi nimi může přepínat v rychlém sledu, ovšem informační dialogy se budou měnit až po každých 2 s.

Android API nabízí rozhraní pro předčasné ukončení zobrazení Toastu, takže by to teoreticky částečně šlo řešit, pokud bychom zobrazované zprávy tagovali. U těch se stejným tagem by se zobrazovala jen ta nejnovější a případně právě zobrazený Toast se zprávou se stejným tagem by se v případě takové nové příchozí ukončil. Pořád ale zůstává problém s velkým časovým odstupem zpráv z nezávislých zdrojů.

Pro řešení tohoto problému jsem tak implementoval paralelně běžící službu (nejde o `Service`), které jsou zasílány požadavky na zobrazení zprávy v dialogu Toast, a ta je zobrazuje hromadně. V pravidelném krátkém časovém intervalu (zlomek času zobrazení Toastu) tedy kontroluje přítomnost nových požadavků a v případě, že jsou k dispozici, je spojí do jedné zprávy a zobrazí. Pokud přijde další zpráva dříve, než je zobrazení Toastu po uplynutí zadaného intervalu ukončeno, je tento Toast zrušen a vytvořen nový, který obsahuje novou zprávu a následně všechny zprávy právě zrušeného Toastu (Toast je zobrazen při spodní hraně obrazovky, proto toto pořadí).

Tato služba běží paralelně, ale zpracování zpráv a přijímání nových požadavků je synchronizováno. V případě nesynchronizace by hrozilo přinejmenším to, že některé zprávy budou ztraceny a nebudou tak zobrazeny.

Zajištění asynchronního zpracování se mezi OOP od FRP verzí aplikace liší, jejich popis proto uvedu až v kapitole věnující se rozdílům mezi těmito verzemi.

5.11 Prostor pro budoucí vylepšení aplikace

Jak bylo zmíněno v úvodu, snažil jsem se při vývoji o vysokou použitelnost aplikace v praxi. Přesto jsou však možnosti dalšího rozvoje a vylepšení aplikace velmi široké.

Z hlediska funkcionality jde například o online ukládání zaznamenaných tras na webový server, jak to umožňují taktéž na cyklisty zaměřené Android mobilní aplikace Strava či MapMyRide.

Značné rezervy jsou zcela jistě i v algoritmech pro sledování pokroku ve sledované trase či optimalizaci trasy. V případě optimalizace trasy se lze například pokusit i o začlenění online Google Maps Road API, které poskytuje funkci Snap to Roads, snažící se mírně modifikovat souřadnice z GPS, aby odpovídaly pozici na silnici či cestě.

Kapitola 6

Specifika OOP a FRP verze

6.1 Specifika z hlediska implementace

6.1.1 Úvod

Nejdříve zkusím více obecně popsat místa, kterých se rozdílná implementace týká a následně rozeberu jednotlivá místa více podrobně.

Jak vyplývá z popisu realizace, aplikace nabízí pokročilé možnosti a funkce. Ovšem o příliš složitý systém nejde. Vyjma trackovací Android aktivity jsou všechny ostatní aktivity poměrně jednoduché, bez složitých vztahů a závislostí mezi vytvářenými objekty. Zřejmým kandidátem na změnu mezi OOP a FRP verzí jsou asynchronní procesy. Ty mohou na pozadí celé aplikace běžet tři – získávání údajů z GPS, které je zajištěno objektem systémové třídy `LocationManager`, stahování souborů (map) z internetu a služba zobrazující informační Toasty. V trackovací Android aktivitě pak dále běží asynchronně proces aktualizující časomíru a proces zjemňující pohyb mapy.

Realizovaná aplikace obsahuje spoustu výpočetně jednoduchých operací nad daty, které jsou silně propojeny s uživatelským rozhraním. Jak bylo ovšem ukázáno v kapitole 3, změna uživatelského rozhraní musí probíhat z hlavního vlákna a přesunutí operací na výpočetní vlákno si žádá určitou režii. Většina výsledků je pak spojena s jedním konkrétním cílem, takže vytváření vazby pozorovaný-pozorovatel postrádá většího smyslu. Tyto operace jsou ovšem mnohdy vhodné k převedení do funkcionální podoby, některé i paralelní, k čemuž je vhodnější použít knihovny StreamSupport.

Samotná View zobrazují v drtivé většině unikátní hodnoty a výsledky pocházejí z příslušné Android aktivity či Fragmentu, takže ani změna architektury není nezbytná, ovšem v případě rozšíření aplikace by se situace mohla změnit. Změnu architektury na Model-View-ViewModel jsem tak u FRP verze provedl.

6.1.2 Použité knihovny pro FRP verzi

Při realizaci FRP verze jsem použil knihovny RxJava 2 s rozšířením RxAndroid 2, RxBinding 2 a StreamSupport. Knihovny RxBus a RxLifecycle jsem vzhledem k jejich omezením, jejichž obejití by v případě realizované aplikace převážilo přínosy, nepoužil. V případě RxBus se jedná o to, že jednorázově zpracovávané úkony vhodné pro převod do FRP jsou zpracovávány na tom samém místě a naopak vazby pozorovatelů a pozorovaných umístěných v různých třídách je třeba navazovat a používat opakovaně, přičemž u obojího knihovna RxBus usnadnění a zlepšení přehlednosti nepřinese. Knihovna RxLifecycle pak standardně nepodporuje například třídu `MapFragment`, kterou v aplikaci používám s FRP prvky.

6.1.3 Zpracování údajů z GPS

Samotné získávání údajů z GPS má na starosti systém. Rozdílný přístup jsem ovšem uplatnil v dalším zpracování. Zatímco u OOP verze informuje posluchač voláním určité metody instanci `RuntimeEnvironment` a následně trackovací Android aktivitu. Pokud je aktivní, u FRP verze je informován prostřednictvím emitované zprávy pozorovaným typu `BehaviorSubject` pouze `RuntimeEnvironment` a aktivní Android aktivita se registruje k němu. To je provedeno z důvodu zachování pořadí operací, neboť `RuntimeEnvironment` musí v obou případech předat novou lokaci ke zpracování aktivní instanci `TrackEnvironment`.

6.1.4 Vykreslování a zobrazování informací o průběhu trasy

V OOP verzi jsou metody pro vykreslování trasy a zobrazování informací o průběhu trasy volány z metody pro zpracování přijaté GPS lokace trackovací Android aktivity. Informační Fragment a Mapové Fragmenty implementují metodu `update(...)` rozhraní `IUpdate`, pomocí které jsou o požadavku na aktualizaci stavu informovány. U FRP verze se Fragmenty registrují k notifikačnímu pozorovateli nové lokace v `RuntimeEnvironment`.

6.1.5 Asynchronní procesy pro zpracování Toastů a pohyb mapy

Tyto procesy běží v obou případech na výpočetním vlákně a v momentě potřeby přístupu ke grafickému uživatelskému prostředí přepínají voláním metody `runOnUiThread(...)` Android aktivity na hlavní vlákno. V případě OOP verze je implementace provedena pomocí instancí tříd `HandlerThread`, `Handler` a `Runnable`. V případě FRP verze je pak využito objektu typu `PublishSubject`.

Pohyb mapy má na starosti instance třídy `MapMover`. Ta je u OOP verze volána jednou z metod aktivní trackovací Android aktivity, a to po zpraco-

vání nové lokace aktivním prostředím `TrackEnvironment`. U FRP verze se `MapMover` registruje jako pozorovatel na lokaci u `RuntimeEnvironment`.

V případě služby zobrazování Toastů, kterou zajišťuje třída `ToastService`, jsou další odlišnost v provedení cyklování a zpracování zpráv. U OOP verze je pro cyklování využito metody `postDelayed(...)` Handleru, FRP verze pak využívá operátoru `buffer`. A zatímco OOP verze vyžaduje manuální ukládání nově přijatých zpráv, u FRP verze se o jejich zadržení do doby zobrazení postará automaticky stream.

6.1.6 Asynchronní proces aktualizace časomíry

Co se týče procesu na aktualizaci časomíry, ten je také implementován pomocí tříd `Handler` a `Runnable`, avšak na hlavním vlákně. FRP verze je založena na běžném `Observable`, který vhodným použitím operátorů `repeatWhen`, `concatMap` a `timer` opakovaně emituje s daným časovým rozestupem zprávu, která je následně přeměněna ve vykonání požadovaného kódu.

Příklad spuštění procesu aktualizace časomíry z třídy `MapFrameActivity`, OOP verze

```
...
  timeHandler = new Handler();
  timeRunnable = new Runnable() {
      @Override
      public void run() {
          "update code"
          timeHandler.postDelayed(timeRunnable,
                ...getTimeUpdateInterval());
      }
};

timeHandler.post(timeRunnable);
...
```

Příklad spuštění procesu aktualizace časomíry z třídy `MapFrameActivity`, FRP verze

```
...
  timeDisposable = Observable
    .just(1)
    .repeatWhen(o ->
      o.concatMap(v -> Observable.timer(
        ...getTimeUpdateInterval(), TimeUnit...))
    )
    .forEach(integer -> {
```

```
        "update code"
    });
...
```

6.1.7 Pomocné třídy z balíčků `handlers` a `helpers`

Ve zmíněných balíčcích se objevuje cyklické zpracování dat. V OOP verzi tedy využívám běžných cyklů a pro FRP verzi aplikace jsem zvolil streamy knihovny StreamSupport. Například ve třídě `StatisticsHandler`, metodě `saveNewTrackRecord(...)` a v úseku vyhledávajícím minimální, respektive maximální nadmořskou výšku, se kód značně zjednodušil, viz následující příklad:

Příklad z třídy `StatisticsHandler`, OOP verze

```
...
  float lowestAltitude = Float.MAX_VALUE;
  float highestAltitude = Float.MIN_VALUE;

  Iterator<LatLngAltTime> locationIterator
     = trackEnvironment.getLocationIterator();
  while (locationIterator.hasNext()) {
  LatLngAltTime location = locationIterator.next();
     if (location.getAltitude() < lowestAltitude) {
        lowestAltitude = location.getAltitude();
     }
     if(location.getAltitude() > highestAltitude) {
        highestAltitude = location.getAltitude();
     }
  }
...
```

Příklad z třídy `StatisticsHandler`, FRP verze

```
...
  float lowestAltitude
     = (float) getAltitudeStream(trackEnvironment)
                        .min()
                        .orElse(Float.MAX_VALUE);
  float highestAltitude
     = (float) getAltitudeStream(trackEnvironment)
                        .max()
                        .orElse(Float.MIN_VALUE);
...
```

Ne u všech tříd však k takovému zjednodušení došlo. Například kód metody `getWantedPairFiles` ve třídě `MapsforgeMapsFilesHandler` působí v OOP verzi s cyklem a v FRP verzi se streamem podobně rozsáhle a totéž se dá říci i o přehlednosti. Jednotlivé řádky kódu si přitom funkčně zhruba odpovídají. Ke zvětšení objemu kódu a zhoršení přehlednosti pak došlo například ve třídách `AvailableMapsJSONHandler` a `DownloadedMapsJSONHandler`. Tento fakt je dán zejména ošetřením výjimek, které je, jak jsem uvedl již v kapitole 3, nutné ošetřovat v každém operátoru zvlášť, jinak by došlo v případě chyby k ukončení zpracování.

6.1.8 Stahování souborů

Stahování dostupných mapových podkladů a informací o nich z internetu je v případě OOP verze implementováno pomocí AsyncTasku. FRP verze je implementováno jako rozšíření třídy `SingleObserver`, jejíž instance zpracovávají vždy pouze jednu emitovanou hodnotu. U AsyncTasku je pro návrat na hlavní vlákno využíváno překrytých metod AsyncTasku, FRP verze využívá metody `runOnUiThread(...)` Android aktivity. A zatímco OOP verze informuje o ukončení stahování pomocí zpětného volání (callback), FRP verze využívá svého pozorovaného typu `PublishSubject`, ke kterému se zdrojová Android aktivita registruje.

6.1.9 View model

Poslední podstatnou odlišnou částí je View model. U OOP verze je ve třídách Android aktivit obsažena vnořená třída ViewModel, která obsahuje prvky pro všechna dostupná View. FRP verze má tyto modely přesunuty do samostatných tříd balíčku `viewmodel` a v Android aktivitách jsou místo nich vnořené třídy `SubjectModel` s prvky typu `PublishSubject` pro každou hodnotu, která je určena k zobrazení grafickým rozhraním. Jak jsem však napsal v úvodu, v současném stavu aplikace zde pro toto rozdělení není využití.

6.2 Specifika z hlediska testování

6.2.1 Testy chování

Obě implementace obsahují stejnou funkcionalitu a zcela stejné uživatelské rozhraní, z čehož vyplývá, že i v testech chování nebude rozdílu. Chování bylo testováno převážně manuálně v emulátoru, v balíčku testů chování jsou však pro demonstraci umístěny dva testy chování. V tomto případě jsou, až na identifikaci balíčku, zcela shodné.

6.2.2 Jednotkové testy

Jednotkové testy nejsou implementovány zdaleka pro všechny třídy, avšak přítomné testy se dotýkají i tříd využívajících FRP. Rozdíl mezi testováním OOP a FRP je zejména v tom, že u OOP testujeme především správnou návratovou hodnotu metody. U FRP naproti tomu potřebujeme testovat zasílání zpráv mezi pozorovaným a pozorovatelem iniciovaným uvnitř metody. Z toho důvodu musíme do testované metody injektovat mockovaného pozorovatele či pozorovaného a na nich testovat, zdali došlo k pokusu o vyvolání emitace či zpracování emitované hodnoty. Pro toto testování již základní knihovna Mockito nestačí a je potřeba využít knihovny PowerMockito, která si poradí třeba i s finálními třídami, mezi které patří mimo jiné třeba i třídy některých FRP prkvů (například subjekty, tedy `PublishSubject` a další).

6.3 Specifika z hlediska ladění

Z hlediska ladění jsem nenašel další zjištění oproti kapitole 3. Tam zmíněná zjištění ovšem mohu potvrdit.

Kapitola 7

Shrnutí zkušeností s FRP

7.1 Vývoj, testování a ladění

Při vývoji je důležité předem podrobněji analyzovat, ve kterých případech má funkcionálně reaktivní programování potenciál být přínosem. Jak ukázala praxe, FRP má své silné, ale také slabé stránky. S oběma přístupy lze sice většinou dosáhnout cíle, ale použití jiného konceptu by mělo být z celkového pohledu přínosem. To se týká i funkcioálního programování v podobě streamů, jejichž problém je například dán zejména způsobem pracování s chybami a výjimkami. Funkcionálně reaktivní programování je přínosem zejména tam, kde je potřeba pracovat s více vzájemně komunikujícími asynchronními procesy. U jednodušších mobilních aplikací pro systém Android si lze vystačit i s nabídkou systémových prvků (tříd) pro asynchronní zpracování dat.

Shrnutí testování a ladění je stručné. Rozdíly v testování vychází z potřeby testovat u FRP zasílané zprávy, což vede k nutnosti mockovat FRP prvky. Nejde o větší komplikaci, jen musí vývojař vědět, jakým způsobem injektáž provést, respektive osvojit si práci s pokročilejší mockovací knihovnou PowerMockito.

V případě ladění je třeba počítat s rozsáhlejším výpisem trasovacích informací, jinak se ladění aplikace významně neliší.

7.2 Nasazení, údržba a budoucí rozvoj

V textu jsem se příliš nevěnoval funkcionálně reaktivnímu programování z hlediska nasazení, údržby a budoucího rozvoje.

V případě nasazení takřka žádná omezení nejsou. Například v případě výpočetně náročných her či jiných výpočetně náročných aplikací by mohlo vadit, že má knihovna RxJava určitou režii co se týče výkonu. Jinak ovšem další rozdíly nevidím.

Z hlediska údržby a dalšího rozvoje aplikace je vhodné, co se týče streamů, složitější streamy dokumentovat a určitě doporučuji několikrát zmíněné

řazení operátorů streamu pod sebe. Taktéž je dobré při předpokladu rozvíjení funkcionality aplikace používat architekturu Model-View-ViewModel místo běžného MVP či MVC, jejíž výhody nemusí být využitelné hned od začátku, jako je tomu ostatně i v případě realizované aplikace, ale do budoucna může být výhodou.

Při rozvoji aplikace jiným vývojářem se pro něj objevuje požadavek na znalost funkcionálně reaktivního programování. Protože, jak zmíním v závěrečném shrnutí, FRP není těžké používat, ale zabere určité úsilí dobře ho pochopit, naučit se ho používat a v neposlední řadě také vědět o možných úskalích (vztaženo zejména ke konkrétním knihovnám), kterých si nezkušený programátor nemusí být vědom.

7.3 Závěrečné shrnutí

Funkcionálně reaktivní programování se ukázalo jako velmi silné paradigma. Nejde však o přístup, který by měl obecně nahrazovat objektově orientované programování. Oba přístupy jsou zaměřené trochu jinak a podle toho mají své silné a slabé stránky. To platí i pro součást FRP v podobě funkcionálního programování. Důležité je chápat, v čem jsou které přístupy silné a podle toho je při vývoji kombinovat.

S poslední větou souvisí i to, co je na použití funkcionálně reaktivního programování nejtěžší. Není to samotné programování v tomto konceptu, ale pochopení a zjištění specifik tohoto konceptu. Sám jsem se reálně setkal s FRP v rámci této diplomové práce poprvé, byť streamy mi něco říkaly již předtím. A byť se mi povedlo FRP aplikovat, pro plné uvědomění si plných možností a správných způsobů jeho použití by bylo potřeba nasbírat více zkušeností na více projektech.

Závěr

Hlavní cíle této diplomové práce byly dva, vzájemně propojené. Jedním z cílů bylo implementovat mobilní aplikaci s GPS trekingem pro mobilní operační systém Android. Druhým cílem bylo aplikovat a porovnat objektově funkcionálně reaktivní přístup a porovnat ho s čistě objektově orientovaným přístupem.

Začnu s funkcionálně reaktivním programováním. Jeho základy se mi snad podařilo pochytit, popsat i použít. Na druhou stranu, jak jsem zmínil v předchozí kapitole, jeho plné a správné využívání vyžaduje větší praxi.

Pokud jde o použití funkcionálně reaktivního přístupu v realizované aplikaci, mnohé části aplikace dokázaly tento přístup využít, ale jen u některých z nich se dá považovat za skutečný přínos. To silně souvisí s tím, že realizovaná aplikace není co se týče propojení jednotlivých komponent příliš složitá a ani asynchronních procesů není mnoho. Síla FRP se z mého pohledu plně projeví v jinak koncipovaných aplikacích, založených právě na složitějším systému vazeb a asynchronních procesech.

Zbývá zhodnotit samotnou mobilní aplikaci. Ta v obou provedeních na základě mého krátkého testování funguje dobře a poskytuje nejen to, co se od ní očekává. Vykreslování trasy a její obarvení je implementováno v souladu s možnostmi podpůrných knihoven a úspěšnost exportu dat do GPX souboru si lze ověřit třeba i pomocí funkce sledování trasy.

Literatura

[1] Google Inc.: Develop Apps | Android Developers. [online]. Dostupné z: https://developer.android.com/develop/index.html

[2] The Android Open Source Project: Git repositories on android. [online]. Dostupné z: https://android.googlesource.com

[3] Evan Czaplicki: Elm: Concurrent FRP for Functional GUIs. [online]. Dostupné z: https://www.seas.harvard.edu/sites/default/files/files/archived/Czaplicki.pdf

[4] Jonas Bonér, Dave Farley, Roland Kuhn, Martin Thompson: The Reactive Manifesto. [online]. Dostupné z: http://www.reactivemanifesto.org

[5] D'Arcy Lussier: MVVM Compared To MVC and MVP. [online]. Dostupné z: http://geekswithblogs.net/dlussier/archive/2009/11/21/136454.aspx

[6] Alexander Mostovenko: rx.js make async programming simpler. [online]. Dostupné z: http://www.slideshare.net/alexandermostovenko/odessa-rx

[7] Netflix, Inc.: RxJava – Github. [online]. Dostupné z: https://github.com/ReactiveX/RxJava

[8] Netflix, Inc.: RxJava Javadoc. [online]. Dostupné z: http://reactivex.io/RxJava/javadoc/overview-summary.html

[9] Thomas Nield: RxJava – Achieving Parallelization. [online]. Dostupné z: http://tomstechnicalblog.blogspot.cz/2015/11/rxjava-achieving-parallelization.html

[10] Jake Wharton: RxBinding – Github. [online]. Dostupné z: https://github.com/JakeWharton/RxBinding

[11] Evan Tatarka: Retrolambda – Github. [online]. Dostupné z: https://github.com/evant/gradle-retrolambda

[12] Andrew Chen: StreamSupport – Github. [online]. Dostupné z: https://github.com/streamsupport/streamsupport

[13] Trello: RxLifecycle – Github. [online]. Dostupné z: https://github.com/trello/RxLifecycle

[14] HwangJR, Inc.: RxBus – Github. [online]. Dostupné z: https://github.com/AndroidKnife/RxBus

[15] Berklas: Waypoints, Routes and Tracks recorded by GPS receivers. [online; cit. 2016-12-22; licence CC BY-SA 3.0]. Dostupné z: https://commons.wikimedia.org/w/index.php?curid=5067700

Příloha **A**

Požadavky a pokyny pro běh a instalaci aplikace

A.1 Požadavky pro běh aplikace

Obě verze aplikace vyžadují zařízení s mobilním operačním systémem Android 4.1 a vyšší a zabudovaným senzorem GPS.

Doporučeno je zařízení s alespoň dvoujádrovým procesorem a s alespoň 1 GiB operační paměti. Česká offline Mapsforge mapa vyžaduje 385 MiB dostupného volného prostoru na externím úložišti.

Aplikace vyžaduje následující oprávnění:

- Přístup k poloze
- Přístup k internetu
- Přístup k externímu úložišti
- Povolení zabránit zařízení v uspání procesoru (wake lock)

A.2 Pokyny pro instalaci aplikace

V cílovém mobilním zařízení je v sekci „Zabezpečení“ jeho nastavení potřeba nejprve povolit instalaci z neznámých zdrojů. Poté je potřeba překopírovat příslušný soubor `.apk` z CD do zařízení a pomocí některého souborového manažeru následně iniciovat instalaci aplikace.

Aplikace je také možné stáhnout z internetu pomocí přímo na `.apk` soubory odkazujících QR kódů na následující straně.

Obrázek A.1: QR kód ke stažení aplikace FRP Examples (`.apk`)
(http://android.voparil.cz/gpstracking/apk/frpexamples.apk)

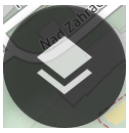
Obrázek A.2: QR kód ke stažení aplikace OOP GPS Tracking (`.apk`)
(http://android.voparil.cz/gpstracking/apk/gpstracking-oop.apk)

Obrázek A.3: QR kód ke stažení aplikace FRP GPS Tracking (`.apk`)
(http://android.voparil.cz/gpstracking/apk/gpstracking-frp.apk)

Příloha **B**

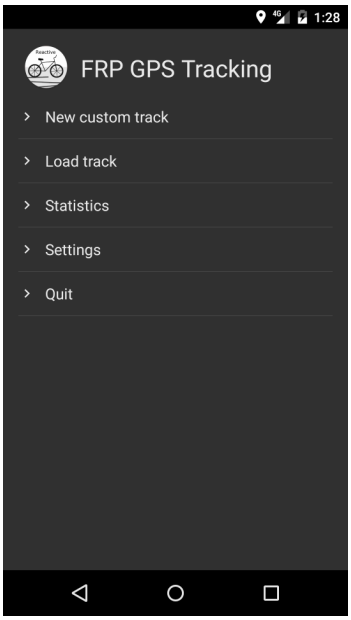
Ovládací prvky trackovací obrazovky

	Přepnout zobrazované informace o trase
	Nastavit střed mapy na aktuální GPS pozici (pro použití mimo aktivní trackování)
	Spustit trackování
	Pozastavit trackování
	Obnovit trackování (po pozastavení)
	Ukončit trackování

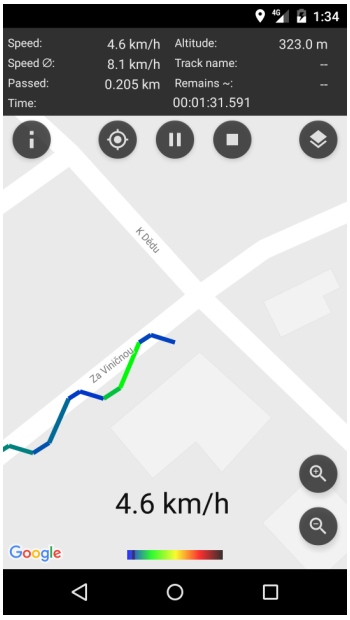
	Přepnout mapový podklad
	Přiblížit mapu
	Oddálit mapu

Příloha C

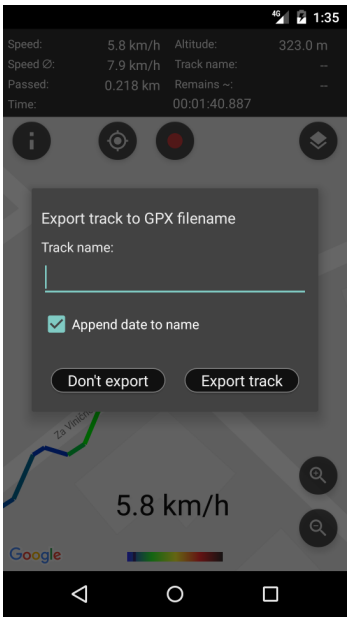
Screenshoty aplikace



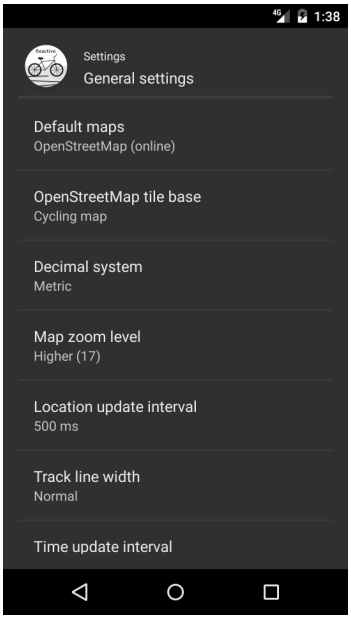
Obrázek C.1: Screenshot úvodní obrazovky



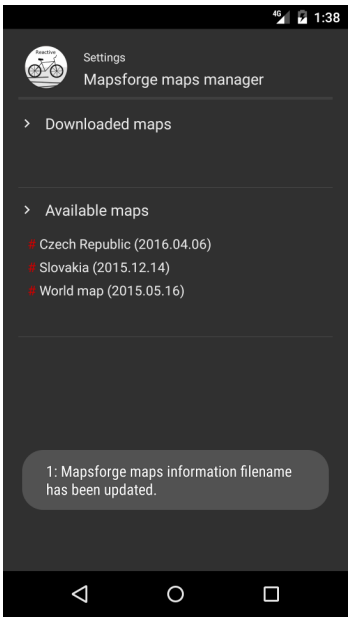
Obrázek C.2: Screenshot trackovací obrazovky



Obrázek C.3: Screenshot exportu zaznamenané trasy do GPX souboru



Obrázek C.4: Screenshot obrazovky nastavení



Obrázek C.5: Screenshot obrazovky pro správu Mapsforge map

Příloha D

Seznam použitých zkratek

API Aplikační rozhraní (anglicky Application Programming Interface)

DAO Data access object

FRP Funkcionálně reaktivní programování

GPS Global Positioning System

GPX GPS Exchange Format

JSON JavaScript Object Notation

MVC Model-View-Controller (architektura)

MVP Model-View-Presenter (architektura)

MVVM Model-View-ViewModel (architektura)

OOP Objektově orientované programování

SDK Software Development Kit

XML Extensible Markup Language

Příloha **E**

Obsah přiloženého CD

```
readme.txt ..................................... stručný popis obsahu CD
app-apk ....................................... instalační .apk soubory
  frpexamples ............................... aplikace FRP Examples
  gpstracking-frp ........... aplikace FRP verze trackovací aplikace
  gpstracking-oop ........... aplikace OOP verze trackovací aplikace
app-src ....................................... Android Studio projekty
  frpexamples ....................... projekt aplikace FRP Examples
  gpstracking-frp ........... projekt FRP verze trackovací aplikace
  gpstracking-oop ........... projekt OOP verze trackovací aplikace
text ........................................................ text práce
  thesis.pdf ........................... text práce ve formátu PDF
text-src ...................... zdrojová forma práce ve formátu LaTeX
```