VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

VYTVÁŘENÍ MOBILNÍCH APLIKACÍ METODOU REAKTIVNÍHO PROGRAMOVÁNÍ

REACTIVE PROGRAMMING IN IOS APPLICATIONS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MATYÁŠ KŘÍŽ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. MARTIN HRUBÝ, Ph.D.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Kříž Matyáš**
Program: Informační technologie
Název: **Vytváření mobilních aplikací metodou reaktivního programování**
Reactive Programming in iOS Applications
Kategorie: Modelování a simulace

Zadání:

1. Prostudujte programování aplikací pro iOS. Prostudujte metody reaktivního programování a knihovny třídy ReactiveX.
2. Navrhněte vlastní knihovnu a programovací styl založený na reaktivním programování. Inspirujte se knihovnami ReactiveX.
3. Implementujte navrženou knihovnu.
4. Knihovnu a programovací styl demonstrujte na sadě ukázek aplikací.

Literatura:

- Keur, Ch., Hillegass, A.: iOS Programming: The Big Nerd Ranch Guide, Big Nerd Ranch Guides; 6 edition (January 6, 2017), ISBN-13: 978-0134682334
- Dokumentace ReactiveX.

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Hrubý Martin, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 1. listopadu 2018

Abstrakt

Programování mobilních aplikací v systému iOS zavádí nebo doporučuje používání asynchronních volání procedur a paralelismus. Bývá obtížné v takovýchto aplikacích udržet přehled o toku událostí v programu. Zavádí se různé metodiky návrhu aplikací založených na asynchronnosti a paralelismu. Jednou z nich je tzv. reaktivní programovaní. Ve své práci se inspiruji metodikou nazývanou ReactiveX a odvozuji z ní metodiku podobnou, o níž se domnívám, že vystihuje metodiku ReactiveX v podstatných ohledech, je však jednodušší.

Abstract

iOS development is built on asynchronous calls and parallelism. Keeping the asynchronous code clear and consise becomes increasingly more difficult with the size of an application. Multiple methodologies emerged to combat this problem. One of them is the reactive programming methodology. In my thesis I focused on creating a reactive methodology inspired by the core ReactiveX concepts with a simpler, more streamlined implementation.

Klíčová slova

Swift, iOS, RxSwift, ReactiveX, reaktivní programování, funkcionální programování, asynchronnost, StreamLibrary, Apple

Keywords

Swift, iOS, RxSwift, ReactiveX, reactive programming, functional programming, concurrency, StreamLibrary, Apple

Citace

KŘÍŽ, Matyáš. *Vytváření mobilních aplikací metodou reaktivního programování.* Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Martin Hrubý, Ph.D.

Vytváření mobilních aplikací metodou reaktivního programování

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Hrubého. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Matyáš Kříž
12. května 2019

Poděkování

Chtěl bych poděkovat doktoru Hrubému za věcné připomínky a pevné vedení při vypracovávání práce.

Obsah

1 Úvod **3**

2 Vyvíjení aplikací na iOS **4**
2.1 Konstrukce Future a Promise 4
2.2 Asynchronní návrhové vzory 5
2.2.1 Vzor Delegate 5
2.2.2 Vzor Observer 6
2.3 Funkcionální Reaktivní Programování 6
2.3.1 Klasické FRP 7
2.3.2 Real-Time FRP 7
2.3.3 Reactive Extensions 8
2.4 Asynchronní operace na iOS 10
2.4.1 Grand Central Dispatch 10
2.4.2 Operation a OperationQueue 11

3 Návrh datových toků **13**
3.1 Třída DataStream 13
3.2 Výčet Event 14
3.3 Třída Watcher 14
3.4 Třída Disposable 15
3.5 Třída DisposeBag 15
3.6 Třída Streamer 17
3.7 Operátory nad datovými toky 17
3.7.1 Operátor Map 17
3.7.2 Operátor Redirect 20
3.7.3 Operátor Filter 21
3.7.4 Další operátory 21

4 Implementace knihovny StreamLibrary **23**
4.1 Rozhraní StreamType 23
4.2 Třída DataStream 24
4.3 Rozhraní WatcherType 26
4.4 Třída Watcher 27
4.5 Výčet Event 27
4.6 Třída Streamer 27

5 Distribuce knihovny StreamLibrary **29**
5.1 Správce Swift Package Manager 29

5.2 Správce Carthage 30
5.3 Správce CocoaPods 31

6 Demonstrační ukázky použití knihovny StreamLibrary v kódu 33

7 Případová studie - Použití knihovny StreamLibrary v Prohlížeči uživatelů GitHubu 37

8 Závěr 42

Literatura 43

Kapitola 1

Úvod

Při vývoji mobilní aplikace je její uživatelské rozhraní nejdůležitější. Základem je, aby se aplikace neočekávaně nezastavovala, další úrovní kvality je, když je používání aplikace intuitivní a jednoduché, ovšem nejvyšší kvalita je zajištěna tím, že jsou všechny operace plynulé a dochází k minimálnímu počtu zpomalení při uživatelově cestě za cílem.

K dosažení plynulosti aplikaci se náročné operace často přesouvají na vedlejší vlákna, které dají hlavnímu vláknu znovu vědět až operace dospěje ke konci. V programovacím jazyce **Swift** na iOS není díky technologii **Grand Central Dispatch** (dále GCD) příliš složité přenést operaci na pozadí. Pokud bychom potřebovali více kontroly nad celým procesem, případně vytvářet závislosti jednotlivých prací na pozadí, lze použít objekty typu **Operation** a **OperationQueue**, jež jsou stejně jako GCD definovány v knihovně **Foundation**, která je vždy k dispozici na platformách Apple. Tyto metody jsou popsány v kapitole 2, zároveň s výčtem jejich výhod a nevýhod.

Problém je v uchovávání příliš mnoha stavů aplikace, kdy jeho změna může mít za následek náhlé neočekávané chování zbytku aplikace. Přesně proto přišly na svět reaktivní knihovny (pro jazyk Swift je to například **RxSwift**), které pomocí řetězení operátorů krotí toto stavové šílenství tím, že každý datový tok je nezávislá entita, která by v ideálním případě neměla číst stav okolí a ani ho měnit.

V kapitole 3 je navrhnut přímočarý systém datových toků a jejich pozorovatelů, který má za cíl zjednodušit používání asynchronních operací ve vyvíjení mobilních aplikací na iOS a tím motivovat vývojáře k častějšímu přesouvání práce na vedlejší vlákna.

Knihovna **StreamLibrary**, která je výstupem této práce, nabízí několik datových typů pro převedení ať už složitých synchronních operací na asynchronní, anebo asynchronních zpětných volání (callbacků) nebo delegací do světa streamů, které fungují jako potenciálně nekonečný datový tok, kde jsou data přislíbeny v budoucnu, až budou připraveny. V této kapitole rovněž dochází k přestavení několika operátorů, jejich použití a implementace.

Jelikož je výsledkem této práce implementace knihovny *StreamLibrary*, jsou v kapitole 4 podrobněji popsány všechny třídy, které knihovna nabízí.

Kapitola 5 uvádí 3 z nejpopulárnějších možností distribuce knihovny pro vývoj na Apple zařízení. Zvažuje jejich výhody a nevýhody a zároveň ukazuje syntaxi jejich souborů.

Časté úkony, které jsou prováděny asynchronně, buď zpětným voláním či jinými způsoby, jsou demonstrovány na sadě příkladů kódu v kapitole 6 zároveň s operátory, které nejsou tak často používány.

Kapitola 7 je poté věnována bližšímu prozkoumání rozdílů mezi standardními postupy při asynchronních operacích a datových toků přímo v příkladové aplikaci pro zobrazení náhodných uživatelů ze stránky GitHubu a jejich základních informací.

Kapitola 2

Vyvíjení aplikací na iOS

Při vyvíjení aplikací na jakoukoliv mobilní platformu, ne jen na iOS, museli vývojáři odjakživa myslet na využívání prostředků jím dostupných obezřetně, jelikož za účelem dosáhnutí co největší skladnosti a jednoduchosti používání nemají mobilní zařízení zdaleka tak vysoký výkon jako například stolní počítače.

Zajištění dodatečného výkonu se v posledních 15 letech dosahuje pomocí dodatečných výpočetních jader na procesorech. Tento výkon na rozdíl od zrychlování taktovací frekvence procesoru není tak citelný a vyžaduje po vývojářích programů, aby specificky rozdělili program na logické celky, které lze provádět souběžně [16].

Asynchronnost je součástí softwarového inženýrství už od šedesátých let dvacátého století, kdy se prvky asynchronnosti objevily v jazyce Simula 67, ovšem do období začátku dvacátého prvního století se taktovací frekvence procesoru zvyšovala exponenciálně a tudíž byla asynchronnost upozaděna ve prospěch objektového programování. Až když začali výrobci procesorů narážet na fyzické bariéry (např. přehřívání) ve zvyšování taktovací frekvence, rozhodli se znásobit počet výpočetních jader [16].

Hlavní způsob efektivního využívání moderních vícejaderných procesorů je neblokování současného vlákna čekáním na dokončení nějaké složitější práce, jejíž výsledky nejsou potřeba k okamžitému pokračování činnosti. Přesně za tímto účelem se začaly formovat osvědčené principy ve formálně definované návrhové vzory určené k tomu, aby využily co nejvíce potenciálu více jader.

Jedněmi z nejvyužívanějších asynchronních návrhových vzorů jsou vzory *Delegate* a *Observer*. Tyto vzory jsou rovněž hojně využívány v třídách UI knihoven na operačních systémech Apple zároveň se zpětným voláním *lambda funkcí*.

2.1 Konstrukce Future a Promise

Princip tohoto konceptu spočívá v tom, že se hodnota (Future) odděluje od jejího výpočtu (Promise), což umožňuje paralelizaci operací.

Tyto konstrukce jsou dnes součástí mnoha programovacích jazyků, do standardu C++ byly přidány ve verzi C++11. Zdrojový kód 1 představuje příklad použití tohoto konceptu na funkci, která provádí náročný výpočet a po jeho spočítání dá o výsledku vědět objektu typu `promise`. V tomto příkladu jsem použil pomocnou funkci `async`, která zaobaluje volání metody a díky tomu stačí jen vrátit výsledek z funkce a automaticky dojde k poslání výsledku objektu typu `promise`. Mezitím se v původním vlákně provádí jiné operace dokud

není potřeba výsledek této náročné funkce, kdy se vlákno zablokuje pomocí metody `get()` a čeká se na výsledek funkce, dokud nedorazí.

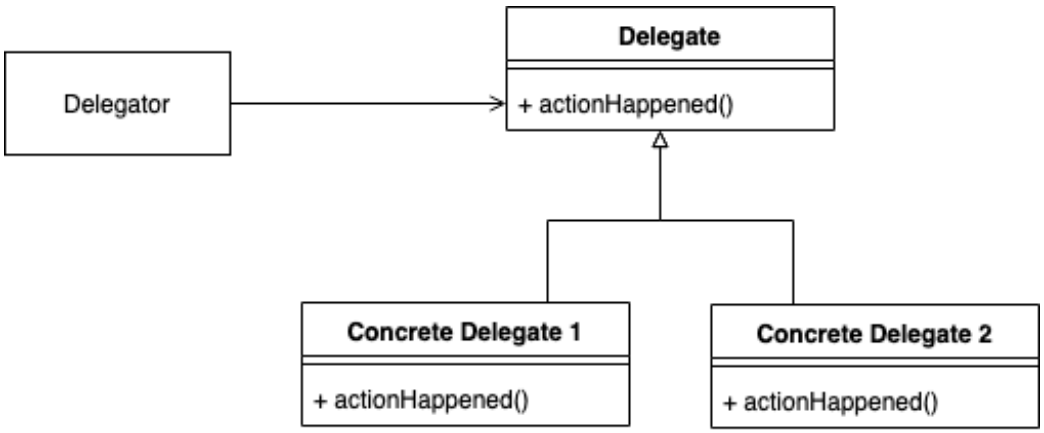
```
int getTrulyRandomNumber(int number) {
    sleep(5000);
    return number + 1;
}

// Výpočet je započat hned po provedení tohoto řádku.
auto future = std::async(getTrulyRandomNumber, 7);
// Zde se provádí nějaký další kód programu.
...
// Získávání hodnoty už je blokující operace, stejně jako
// kdyby se čekalo na výsledek funkce volané synchronně.
// Rozdíl je, že pokud byla hodnota mezitím vypočítána,
// je pouze vrácena bez prodlení.
std::cout << "Number: " << futureRandomNumber.get() << std::endl;
```

Zdrojový kód 1: Příklad použití konstrukcí *Promise* a *Future* v jazyce C++ za použití pomocné funkce `async`. Importování potřebných knihoven bylo z příkladu vynecháno kvůli zkrácení.

2.2 Asynchronní návrhové vzory

2.2.1 Vzor Delegate

Delegace je způsob jak zajistit stejnou znovupoužitelnost při kompozici jako při dědění. Při delegaci jsou zodpovědné při zpracovávání požadavku dva objekty, kdy jeden objekt (delegátor) vlastní referenci na svého delegáta, kterému přeposílá operace, jak lze vidět na Obrázku 2.1. Princip je totožný jako při volání metody rodičovské třídy, akorát zde neprobíhá dědění a objekt, který posílá operace nevolá metody sebe sama, ale příslušné metody delegáta [12].



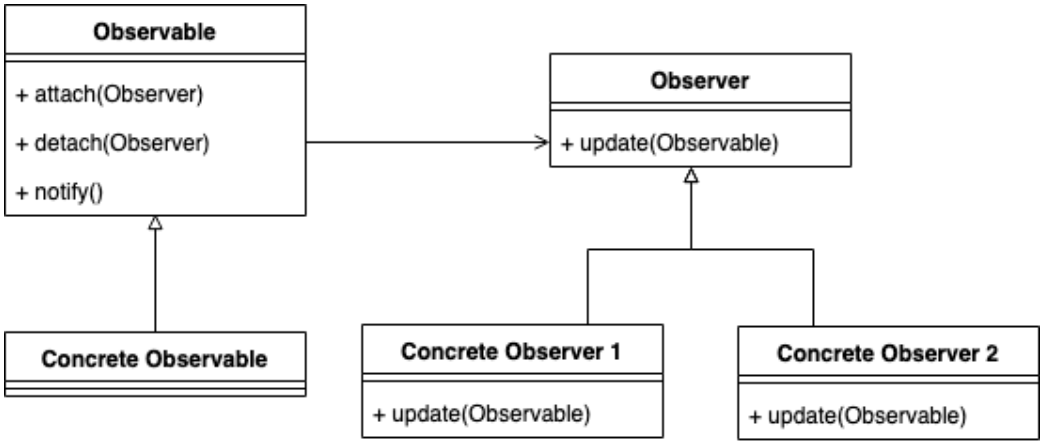
Obrázek 2.1: UML diagram návrhového vzoru Delegate.

Mezi výhody tohoto návrhového vzoru patří možnost snadno změnit delegáta za běhu aplikace a v případě knihoven je to definování vlastního chování pro vyřizování operací tříd, do kterých nelze zasahovat.

Delegát ovšem neví, kolikrát se daná metoda zavolá, kdy v běhu aplikace a ani na jakém vlákně bude vykonávána.

2.2.2 Vzor Observer

Návrhový vzor *Observer* definuje objekt observable, jenž notifikuje při změně svého stavu všechny naslouchající objekty observer [12]. Jedná se tedy o závislost 1 ku N, jak ukazuje Obrázek 2.2.



Obrázek 2.2: UML diagram návrhového vzoru Observer.

V interaktivních aplikacích, kde je potřeba sledovat stále se měnící stav, tento návrhový vzor stále převažuje a není tomu jinak ani na systémech Apple, kde se používá pro naslouchání na změny stavu instančních proměnných pomocí mechanismu Key-Value Observing (KVO). Potenciální problém tohoto přístupu je nezrušení naslouchání, což vede k úniku paměti (memory leak) během celého běhu aplikace.

Na iOS lze také nastavit při interakci s uživatelským rozhraním (např. stisknutí tlačítka) zavolání metody, která má provést nějakou akci. Tento způsob eliminuje možnost zapomenutí zrušení naslouchání, ovšem ani ten není bezchybný – problém je v udržování stavu.

Návrhový vzor totiž svádí k vytváření mnoha stavových proměnných [14], které obvykle musí být na úrovni objektu, tedy jako instanční. To nabízí vytváření závislostí na proměnných, které slouží pouze jako pomocné pro zvolení reakce na změny stavu observable. Potom už stačí jen změnit způsob, jakým se zachází s těmito pomocnými proměnnými a na světě je nové, chybné, chování.

2.3 Funkcionální Reaktivní Programování

Čím více stavů může program nabývat za použití proměnných, tím více mentálního žonglování musí vývojář dělat, aby mohl psát kód, který se nerozbije při nasazení do produkce.

Tento problém nespočtu pomocných proměnných řeší funkcionální programování, kde lze vytvářet pouze konstanty. Ve spojení s reaktivním programováním přesahuje užitečnost vzoru observer a nabízí jednodušší kód.

Funkcionální reaktivní programování (dále jen FRP) je asynchronní programovací vzor, který staví na základech deklarativního funkcionálního programování. V reaktivním programovacím vzoru je hlavním konceptem propagace změny, která je zajištěna pomocí datových toků.

Deklarativní programování rozlišuje mezi "co naprogramovat"a "jak to naprogramovat". Tím umožňuje programátorovi definovat vztahy mezi daty (co) s cílem automatizovat způsob implementace (jak).

2.3.1 Klasické FRP

FRP původně vzniklo pro potřeby reakce na nepřetržitý uživatelský vstup a animace poněkud přímočařeji oproti imperativnímu způsobu. Cílem bylo dosažení stejných výhod jako nabízí deklarativnost funkcionálního programování; kromě přehlednosti také snadné vytváření a znovupoužití kódu [13].

FRP pro modelování dat definuje dva typy:

- **Behavior** je entitou první kategorie, tudíž ji lze použít jako argument nebo vracet jako výsledek funkce. Popisuje hodnotu měnící se v čase, kdy její aktuální hodnota je definována jako funkce v čase.
- **Event** je taktéž entitou první kategorie a slouží pro popis dějů v reálném světě (např. interakce s tlačítkem), nebo jako podmínka založená na animačních parametrech (např. vzdálenost nebo kolize).

Nejjednodušší primitivní chování (*Behavior*) je funkce identity na čase, kdy pouze dojde k vrácení hodnoty času, která byla vložena jako parametr funkce *Behavior*.

2.3.2 Real-Time FRP

Na základech klasického FRP vzniklo Real-Time FRP. V tomto programovacím vzoru Paul Hudak a spol. definují nový typ *Signál*, jenž spojuje jak *Behavior*, tak *Event*. Za použití *Signálů* odstranili nedostatky klasického FRP omezením používání na způsoby, které měly zaručeně efektivní implementaci [17].

Jako návrh zlepšení Real-Time FRP přišlo Event-Driven FRP, jenž fungovalo v podstatě totožně. Rozdíl byl v tom, že *Signál* se stal diskrétním a hodnoty se propagovaly pouze tehdy, došlo-li ke změně [18].

Existují dva způsoby implementace FRP: *pull* a *push*.

- **pull-based** neprovádí žádné operace do té doby, dokud nejsou výsledky potřeba a poté zpětně počítá z nejaktuálnějších dostupných hodnot pomocí zpětného získávání závislostí.
- **push-based** počítá definované výpočty při jakékoliv změně dat. Tudíž se propagace změn děje při dostupnosti nových dat namísto toho, aby se čekalo, až data někdo bude chtít.

2.3.3 Reactive Extensions

Reactive Extensions (ReactiveX nebo jen Rx) je knihovna pro sestavování asynchronních programů založených na událostech za použití observable sekvencí. Rozšiřuje návrhový vzor *Observer* tak, že přidává podporu potenciálně nekonečných sekvencí dat nebo událostí a poskytuje operátory k jejich deklarativnímu řetězení bez nutnosti přemýšlení nad vícevláknovým paralelním zpracováním operací, bezpečností přístupu z více vláken, asynchronními datovými strukturami a neblokujícím čtením a zápisem.

ReactiveX nevyžaduje žádný specifický zdroj asynchronnosti. Sekvence observable mohou být implementovány pomocí fondů vláken, událostní smyčky (angl. event loop), neblokujícího čtení a zápisu nebo jakékoliv další implementace. Klientský kód bere jakékoliv interakce se sekvencemi observable jako asynchronní ať už jsou operace implementovány jako blokující či ne [7].

I přesto, že byla knihovna ReactiveX výrazně inspirována funkcionálním reaktivním programováním, není součástí tohoto návrhového vzoru. Nejvíc se blíží Event-Driven FRP tím, že ReactiveX pracuje pouze na diskrétní bázi. Zatímco však tento typ FRP používá funkce v čase, ReactiveX pouze posílá vyvolané události.

ReactiveX dává programátorovi k dispozici sekvence observable a pestrou škálu operátorů pro různé spojování, transformování a vytváření observable sekvencí. Na konec této sekvence lze napojit naslouchač jménem observer, který může reagovat na data nebo události procházející sekvencí.

Zdrojový kód 2 ukazuje příklad zřetězeného zpracování v jazyce Swift za použití knihovny RxSwift.

```swift
// Instanční konstanta `rx` slouží ke sjednocení přistupování
// k observable sekvencím jednotlivých prvků uživatelského rozhraní.
// `rx.text` posílá při změně textu ve vyhledávacím poli současné znění hledaného řetězce.
let searchResults = searchBar.rx.text
    // `throttle` funguje jako časové síto sloužící k tomu,
    // aby se datové události neposílaly příliš často.
    // Použitím tohoto operátoru dojde k omezení posílání hledaných řetězců
    // na každých 0.3s od předchozí událost s tím, že se vždy posílá poslední událost.
    .throttle(0.3, scheduler: MainScheduler.instance)
    // `distinctUntilChanged` propouští datovou událost pouze
    // pokud se liší od poslední události.
    .distinctUntilChanged()
    // `flatMap(Latest)` je používaný k transformování dat na další observable sekvenci.
    // Přídavek "Latest" značí, že když dojdou nová data, má se zahodit výsledek funkce
    // `searchGitHub`, pokud ještě nepřišel, zahodit a místo toho se má brát výsledek
    // této funkce zavolaný s novým hledaným řetězcem.
    .flatMapLatest { query -> Observable<[User]> in
        if query.isEmpty {
            // Pokud je hledaný řetězec prázdný, pouze vrátíme
            // konstantní hodnotu prázdného pole.
            return .just([])
        }
        // Funkce `searchGitHub` znovu vrací observable sekvenci,
        // jelikož se jedná o síťovou operaci a výsledek vrátí až v budoucnu.
        return searchGitHub(query)
            // Pokud nastane chyba (např. není přístup k internetu),
            // tak se má vrátit pouze prázdné pole místo chyby.
            .catchErrorJustReturn([])
    }

// Na rozdíl od konvenčního asynchronního volání je v ReactiveX možné uložit
// sekvenci observable do proměnné, dále ji transformovat a na úplně jiném místě
// v programu naslouchat na události, které jí projdou.
searchResults
    // Sekvence observable se dá na konci napojit buď na určitou `rx`
    // proměnnou pomocí `bind`, nebo jen naslouchat na události pomocí `subscribe`.
    // Zde `subscribe(onNext:)` naslouchá pouze na datové události (onNext).
    .subscribe(onNext: { users in
        print("Found \(users.count) users.")
    })
    // Po napojení nebo naslouchání na sekvenci observable je vrácen tzv. objekt
    // `Disposable`, jenž je použit ke zrušení naslouchání a celkového provádění
    // zřetězení sekvence observable.
    .disposed(by: disposeBag)
```

Zdrojový kód 2: Příklad zřetězeného zapojení operátorů na sekvenci observable při naslouchání na změny vyhledávacího textového pole a automatické získání aktuálních dat pomocí provedení síťové operace při změně hledaného řetězce.

2.4 Asynchronní operace na iOS

Jazyk Swift, respektive knihovna Foundation, nabízí několik způsobů k provádění asynchronních operací. Mezi nimi nejpoužívanější je technologie **Grand Central Dispatch**, která skýtá možnost přenést nepojmenovaný blok kódu na určitý typ vlákna nebo vykonat ho po určité době. Pokud vývojáři potřebují spojit několik operací do řady (respektive do stromu) na sobě závislých operací nebo potřebují z jiných důvodů mít více kontroly nad tím, v jakém pořadí a jakým způsobem se operace provádějí, mohou sáhnout po objektech typu **Operation** ve spojení s **OperationQueue**, které navíc umožňují asynchronní operaci vyřadit z fronty, pokud se ještě nezačala vykonávat.

2.4.1 Grand Central Dispatch

Grand Central Dispatch (dále jen **GCD**) je nejpoužívanější a zároveň nejjednodušší na používání ze všech nástrojů pro asynchronní operace nabízených v základu jazykem Swift.

DispatchQoS

Tato struktura určuje kvalitu služby, nebo prioritu, dané operace. Používá se ke kategorizaci práce vykonávané pomocí `DispatchQueue`. Priorita ovlivňuje přednost operace a zároveň množství prostředků, jaké je k vykonávání práce využito [3]. Z důvodů uživatelské přívětivosti a efektivitě využití energie je třeba použít vždy správnou třídu služby.

Knihovna `Foundation` definuje 4 předem připravené třídy služeb podle priority, zde jsou vypsány od nejvyšší [6]:

- **User-Interactive** reaguje na uživatelovu interakci. Pokud by se neprovedla rychle, mohlo by dojít k zamrznutí uživatelského rozhraní. Práce na této úrovni by měla být téměř instantní.
- **User-Initiated** je započnuta uživatelem a vyžaduje výsledek bezprostředně po zavolání. Pokračování uživatele závisí na dokončení operace na této vrstvě. Operace na této úrovni by měly být dokončeny skoro okamžitě, méně než za pár vteřin.
- **Utility** vyřizuje operace, které mohou trvat nějakou dobu a jejich výsledek potřebný okamžitě. Úkoly na této vrstvě jsou obvykle sledovány uživatelem v podobě indikátoru průběhu. Tato vrstva přináší výhodu v podobě rovnováhy mezi rychlostí odezvy, množstvím zdrojů a efektivitou využití energie. Na této úrovni by třída měla zabírat v řádu několika vteřin až minut.
- **Background** se stará o operace na pozadí, které uživatel nevidí vůbec. Například indexování, synchronizace nebo zálohování. Zaměřuje se na efektivitu využití energie. Operace vykonávané touto vrstvou mohou trvat velmi dlouho, od minut až po hodiny.

Každé struktuře `DispatchQoS` lze také nastavit relativní prioritu vůči vybrané třídě služby. Tato relativní priorita nemůže posunout operaci do jiné třídy použitím velmi velkých nebo malých čísel – funguje pouze v třídě, ve které se nachází, pro seřazení operací.

Mimo tyto 4 třídy připravené pro programátory existují ještě 2 zvláštní třídy:

- **Default**, jejíž priorita spadá mezi třídy **User-Initiated** a **Utility**. Tato třída neslouží ke klasifikaci typu operace vývojářem, je použita v případě, že práce nemá nastavenou žádnou třídu služby. Tato třída je používána globální frontou **GCD**.

- **Unspecified** označuje absenci informací o třídě služby a značí systému, že by se měla třída vyvodit. Vlákna mohou být takto označeny, pokud používají staré API, které nemusí nutně využívat systému tříd služeb.

DispatchQueue

`DispatchQueue` objekty jsou fronty typu "první dovnitř, první ven"(FIFO), kam může aplikace vkládat bloky kódu v jazyce Swift. Vložené operace jsou prováděny buď synchronně nebo asynchronně. Pro vykonávání operací jsou využívány fondy vláken, které jsou spravovány systémem.

V případě naplánování práce na synchronní vykonání čeká kód, jenž práci naplánoval, na její vykonání a následně pokračuje. Asynchronní vykonávání pouze vloží blok kódu do fronty a dále pokračuje, zatímco se práce vykonává na nějakém jiném vlákně.

Z důvodu, že systém limituje, kolik vláken má aplikace přidělena, je potřeba co nejméně používat synchronní vykonávání, jelikož GCD musí vytvořit další vlákno pro provedení této práce. Během toho musí spravovat jiné vlákna, která provádějí asynchronní operace.

Dokumentace taktéž doporučuje vytvářet co nejméně vlastních objektů `DispatchQueue`, které jsou soukromé, jejich vlákna se počítají do celkového limitu vláken, která systém aplikaci umožňuje vytvořit [4].

Pokud má mít asynchronní operace nějaký zpětný efekt na uživatelské rozhraní (např. chybová hláška v případě neúspěchu), je potřeba uvnitř tohoto bloku vykonat další volání GCD, které tentokrát pošle danou akci do fronty hlavního vlákna, které jako jediné může provádět jakékoliv změny uživatelského rozhraní [10].

Technologie GCD rovněž umí nastavit zavolání daného bloku kódu za určitý čas od synchronního zavolání metody pro přidání do fronty určitého vlákna.

```
1 DispatchQueue.global(qos: .background).async {
2     print("Hello, asynchronous world!")
3 }
```

Zdrojový kód 3: Vložení asynchronní operace do globální fronty `DispatchQueue` lze vejít jen na pár řádků kódu.

Jak ukazuje Zdrojový kód 3, použití je přímočaré a API jednoduché, ale z toho důvodu, aby byl GCD snadný na používání, také neumožňuje určité složitější operace jako navazování závislostí a jiné komplikovanější nastavení.

Posílání operace přes GCD z hlavního vlákna na hlavní vlákno synchronně způsobí deadlock, jelikož hlavní vlákno čeká na dokončení bloku kódu, které se ale nezačne vykonávat, jelikož hlavní vlákno čeká.

2.4.2 Operation a OperationQueue

Použití tohoto páru objektů je o něco složitější než za pomoci **GCD**, ale nabízí mnohem větší kontrolu nad asynchronní operací od začátku do konce. Další výhodou oproti výše zmíněné technologii je vyřazení operace z fronty, pokud ještě nezačalo její vykonávání.

Vlastní operaci lze definovat děděním od třídy `Operation`, jinak nelze udělat žádnou reálnou práci [5]. Alternativou je použití předem definované třídy `NSInvocationOperation`, která slouží pro zavolání metody v jazyce Objective C (nebo na metodu v jazyce Swift za

použití atributu `@objc`) nebo `BlockOperation`, jenž umožňuje spustit blok kódu v jazyce Swift.

Pro synchronní operace stačí jen přetížit metodu `main()`. Zároveň se doporučuje vytvořit inicializační funkci, která umožní jednodušší vytváření instancí. Mimo to, všechny proměnné, které třída definuje by měly být bezpečné pro přístup z více vláken.

Pro asynchronní operace je potřeba přetížit minimálně:

- metodu `start()`, jež je zavolána pro vykonání operace,
- příznak `isAsynchronous`, který udává, zda je operace asynchronní, v tomto případě bude nastavený na pravdivou hodnotu,
- příznak `isExecuting`, který ukazuje, že je operace uprostřed vykonávání,
- příznak `isFinished`, který značí, že operace již doběhla do konce. Musí být nastaven na pravdivou hodnotu i při zrušení operace.

Vytvořený objekt, který dědí od `Operation` lze zavolat přímo pomocí instanční metody `start()`, ovšem je potřeba se ujistit, že je operace připravená k provedení, tuto informaci lze najít v příznaku `isReady`, což může značně komplikovat přehlednost kódu. Operace může být označena jako nepřipravena v tom případě, že je závislá na nějaké jiné operaci, která ještě není dokončena. Z tohoto důvodu se spíše využívají objekty `OperationQueue` tak, že se do nich operace vloží jako do fronty, jak napovídá název.

Jelikož umožňuje `Operation` mnohem více kontroly nad provedením asynchronní operace, nechává zároveň na vývojáři reakce na nastavené příznaky, například, že byla operace zrušena, aby mohl vývojář provést vlastní ukončení operace, nebo tyto příznaky zcela ignorovat.

Po dokončení operace definované v metodě `start()` musí objekt zaktualizovat příznaky `isExecuting` a `isFinished`. Objekt operace nelze odstranit z fronty, dokud není hodnota `isFinished` pravdivá, tudíž nezáleží na tom, zda operace skončila chybou, dokončena de facto je.

Thread

Ve jazyce Swift lze pracovat přímo s vlákny za použití třídy `Thread`, jenž nabízí příznaky, které se sémanticky velmi podobají příznakům v `Operation` [9].

`Thread` slouží pro spouštění metod jazyka Objective C (nebo atribut `@objc`) nebo bloku kódu v jazyce Swift, ovšem k oběma účelům je lépe vybavený systém **Grand Central Dispatch**.

Pro definování vlastního chování je třeba vytvořit třídu dědící od třídy `Thread` a v přetížené metodě `main()` vykonat požadovanou operaci. Při jejím přetížení není třeba volat rodičovskou stejnojmennou metodu.

Kapitola 3

Návrh datových toků

Syntaxe a používání knihovny, která je výsledkem této práce, je výrazně inspirována reaktivní knihovnou ReactiveX a kromě nejdůležitějších objektů obsahuje i nejpoužívanější operátory. Cílem zjednodušené implementace jádra ReactiveX a zahrnutí pouze těch nejpoužívanějších operací je, aby se pomocí této knihovny čtenář přesvědčil o výhodách používání reaktivní knihovny a necítil se zahlcen velkým počtem operátorů a datových typů. Složitější reaktivní knihovny, jako například již zmíněná ReactiveX, slouží potom hlavně k dosažení komplexnějších cílů, když už si je programátor jistý, že chce využít reaktivní programování i v produkčních aplikacích.

Samotná knihovna není optimalizovaná do takové míry jako ReactiveX knihovna pro jazyk Swift – RxSwift. Na jednu stranu to znamená, že je pomalejší, ale na druhou to umožňuje nahlédnout na zjednodušenou implementaci fungování reaktivní knihovny a zjistit, proč funguje tak jak funguje.

Zároveň se snažím názvy přiblížit k reálnému světu, přesněji streamování videa. Roli observable sekvencí zde zastupují objekty `DataStream`, kterými proudí události typu `Event`. Po veškeré transformaci je třeba datový tok začít sledovat (obdoba naslouchání ze světa ReactiveX), aby se vykonaly operace s ním spojené. Sledující typu `Watcher` může i reagovat na události nebo data, které projdou.

3.1 Třída DataStream

Třída *DataStream* je základní stavební blok knihovny. Jedná se o datový typ, který popisuje sekvenci událostí v čase. Nemá žádnou aktuální hodnotu, která by se dala vyčíst. Dalo by se říct, že se jedná o referenci na datový tok, kam v budoucnu přijdou nějaké události, na které upozorňuje objekty typu `Watcher`.

Nejjednodušší způsob převedení synchronní nebo asynchronní operace do asynchronního světa datových toků je zaobalit práci blokem kódu, který bude zavolán v okamžiku, když nějaký Watcher započne sledování (příklad tohoto způsobu je Zdrojový kód 4). V tomto bloku kódu je k dispozici reference na sledující, kterému lze sdělovat průběh operace, ať se jedná o úspěch s určitými daty, chybu, nebo konec datového toku.

```
1  func getDocument(from reference: DocumentReference) -> DataStream<DocumentSnapshot> {
2      return DataStream { watcher in
3          // Proměnná `watcher`, která je k dispozici v tomto rámci, je reference
4          // na sledujícího, kterému se posílají na řádcích #7 a #9 události.
5          reference.getDocument(completion: { snapshot, error in
6              if let error = error {
7                  watcher.error(error)
8              } else if let snapshot = snapshot {
9                  watcher.data(snapshot)
10             }
11         })
12
13         return Disposables.create()
14     }
15 }
```

Zdrojový kód 4: Příklad vytvoření nového datového toku převedením ze zpětného volání požadavku na dokument z databáze Firestore.

3.2 Výčet Event

Objekt typu `Event` (dále jen událost) za pomocí generiky zaobaluje datový typ, který bude proudit datovým tokem.

Událost vždy nabývá jedné ze tří možných hodnot:

- **Data** generického datového typu.
- **Chyba**, ke které došlo ať už během vykonávání operace nebo při transformaci, typu `Error`.
- **Konec** datového toku, jenž je poslán, když už se nemají očekávat další data.

3.3 Třída Watcher

Třída `Watcher` zaujímá funkci naslouchače na události datového toku ať už jsou nějak transformované či nikoliv. Naslouchá na všechny typy událostí, ovšem nemusí na žádnou z nich reagovat; hlavním účelem sledování datového toku je započetí jeho vykonávání.

Započetí naslouchání na události datového toku objektem typu `Watcher` vede k zavolání metody, která byla definována při vytvoření datového toku, výsledkem tohoto zavolání je objekt typu `Disposable`.

Mimo naslouchání na standardní události dat, chyby a konce streamu lze také naslouchat na ukončení sledování streamu. Toto není jedna z událostí, které jsou popsány výše. Lambda funkce naslouchající na ukončení sledování je zavolána potom, co je zavolána metoda `dispose` na `Disposable`.

3.4 Třída Disposable

Třída `Disposable` je pouze abstrakcí nad lambda funkcí, jejímž zavoláním se ukončí naslouchání, zároveň lze v této funkci zavolat zrušení operací, které jsou naplánovány (nebo se vykonávají). Tyto `Disposable` objekty lze shromažďovat v objektu typu `DisposeBag`.

V této lambda funkci doporučuji uvést věci do stavu před zavoláním, pokud došlo k vedlejším efektům, které dává smysl zvrátit. Například pro zastavení asynchronní operace, pokud již skončilo sledování datového toku, jak je tomu ve Zdrojovém kódu 5

```
1  static func GETraw(url: URL) -> DataStream<Data> {
2      return DataStream { watcher in
3          var request = URLRequest(url: url)
4          request.httpMethod = "GET"
5
6          let dataTask = URLSession.shared
7              .dataTask(with: request) { data, response, error in
8                  if let error = error {
9                      watcher.error(error)
10                     return
11                 }
12                 guard let data = data else { return }
13                 watcher.data(data)
14                 watcher.end()
15             }
16
17         // Započetí síťového požadavku.
18         dataTask.resume()
19
20         return Disposable {
21             // Tento kód se zavolá až při zrušení sledování datového toku.
22             dataTask.cancel()
23         }
24     }
25 }
```

Zdrojový kód 5: Příklad převedení asynchronní síťové operace na datový tok s důrazem na zrušení vykonávání požadavku při zavolání lambda funkce `Disposable`.

3.5 Třída DisposeBag

Třída `DisposeBag` abstrahuje pole funkcí Disposable. Slouží pro zavolání všech Disposable při de-inicializaci sebe sama. Díky tomuto principu lze definovat DisposeBag v UIViewControlleru a potom už jen při započetí sledování Streamu výsledné Disposable vkládat.

Hlavní použití této třídy je k usnadnění držení referencí na jednotlivé naslouchání datových toků (`Disposable`) a možnost jejich hromadného zavolání v případě potřeby.

Většinou je třída `DisposeBag` inicializována jednou jako instanční konstanta (pomocí klíčového slova `let` v jazyce Swift). Díky systému **Automatic Reference Counting**[1] se pak `DisposeBag` automaticky deinicializuje spolu s objektem, ve kterém je definován. To způsobí zavolání všech `Disposable` objektů, které obsahuje a zrušení naslouchání datových toků, ze kterých pocházejí.

Ovšem není pravidlem vytvořit objekt typu `DisposeBag` jen jednou za život třídy, ve které se nachází. Ve vysoce dynamických aplikacích může docházet pouze k recyklaci již inicializovaných prvků uživatelského rozhraní. Například u tabulek je toto velmi obvyklé, a pokud záleží typ datového toku na tom, o jakou buňku se jedná, je potřeba pokaždé zahodit předchozí naslouchání, jak je využito ve Zdrojovém kódu 6.

```
1  final class CryptoCurrencyCell: UITableViewCell, StreamUpdatable {
2      // Zde je typ stavu datový tok, kterým prochází opakovaně
3      // aktualizovaná data určité kryptoměny.
4      typealias T = DataStream<CryptoCoin>
5
6      private var updateDisposeBag = DisposeBag()
7
8      func update(newValue stream: T) {
9          // Tím, že dojde k přepsání objektu typu `DisposeBag` novým dojde k zavolání
10         // deinicializační funkce předchozího objektu typu \code{DisposeBag} a tím
11         // zrušení sledování předchozího datového toku.
12         updateDisposeBag = DisposeBag()
13
14         // Zde záleží na pořadí a sledování nového datového toku musí přijít
15         // až po nahrazení objektu typu `DisposeBag`, jinak by byla funkce `Disposable`
16         // přidána do starého objektu a její sledování by skončilo
17         // hned při nahrazení starého objektu.
18         stream.watch(...).disposed(by: updateDisposeBag)
19     }
20 }
```

Zdrojový kód 6: Příklad vytvoření objektů typu `DisposeBag` při každém obnovení stavu buňky. Rozhraní `StreamUpdatable` očekává vnější nastavování stavu pouze přes metodu `update(newValue:)`, což umožňuje například vložení datového toku do buňky. Na tento datový tok se poté naslouchá do té doby, než je buňka recyklována a je jí pomocí již zmíněné metody zvenčí nastaven nový stav. Nahrazením předchozího objektu typu `DisposeBag` se zruší naslouchání na předchozí tok a nové naslouchání je přidáno do tohoto nového objektu. Toto umožňuje vyhnout se únikům paměti při nastavování stavu a zároveň před dealokací třídy opět dojde k automatickému zrušení naslouchání na současný datový tok.

[1] Automatic Reference Counting je systém správy paměti na operačních systémech Apple (např. iOS nebo Mac OS), jenž u každého objektu udržuje počet referencí a pokud klesne na nulu, je objekt dealokován z paměti [1].

3.6 Třída Streamer

Třídu `Streamer` lze použít buď jako objekt `DataStream`, nebo jako objekt `Watcher`. Díky této třídě lze shromažďovat události z více zdrojů, ze stále se opakujících akcí nebo z operací, jenž by se příliš složitě zaobalovaly do DataStreamu.

3.7 Operátory nad datovými toky

Do knihovny jsem se rozhodl implementovat pouze ty nejpoužívanější operátory nad datovými toky, aby nebyl vývojář zbytečně zahlcen.

Jako grafickou reprezentaci operátorů StreamLibrary využiji principů RxMarbles [15], jenž se používá pro vysvětlení ReactiveX operátorů pomocí grafů, příkladem je Obrázek 3.1. V těchto grafech je datový tok definován jako časová osa zleva doprava, na které se nacházejí události v podobě dat (kroužek s daty), chyby (křížek), nebo konce toku (svislá čára).

Obrázek 3.1: Příklad jednoduchého celočíselného datového toku, kterým za jeho život prošly postupně hodnoty 1, 2, 3 a 4. Po čísle 4 následuje vertikální čára, jenž denotuje konec toku. Šipka směrující doprava pouze značí, že čas v grafu teče zleva doprava.

3.7.1 Operátor Map

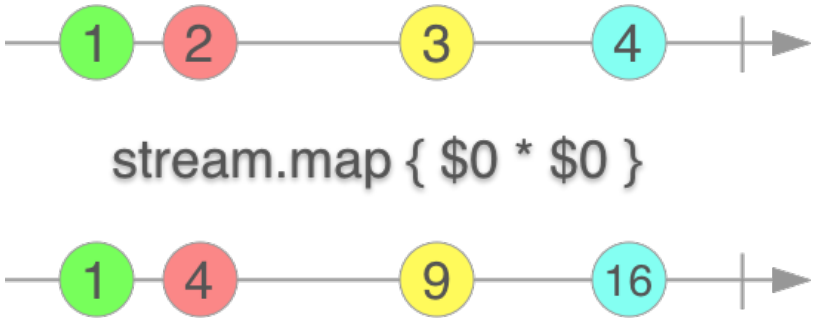
Mezi nejvyužívanější patří `map` – transformace dat v datové události na jiné data. Tato metoda pouze bere lambda funkci, jejímž vstupním parametrem jsou data a výstupem je jakýkoliv typ, se kterým se bude dále pracovat. Tato metoda se nijak neliší od metody `map` definované na kolekcích. Její implementace je okomentována ve Zdrojovém kódu 7. Ve Zdrojovém kódu 8 lze najít pár příkladů použití tohoto operátoru.

```
func map<T>(_ transform: @escaping (E) throws -> T) -> DataStream<T> {
    // Důvod vytváření nového datového toku místo přímého volání metody `watch`
    // je to, že tímto způsobem je zajištěno, že se naslouchání neprovede dokud
    // nedojde k jeho zahájení v kódu aplikace.
    return DataStream<T> { watcher in
        self.watch { event in
            switch event {
            case .data(let data):
                do {
                    // Klíčové slovo `try` je v jazyce Swift vyžadováno na řádku,
                    // kde je zavolána metoda, která může vyhodit výjimku.
                    try watcher.data(transform(data))
                } catch {
                    watcher.error(error)
                }
            // Operátor reaguje pouze na datové události, přičemž ostatní
            // pouze přeposílá, jak je vidět v případech chyby a konce toku.
            case .error(let error):
                watcher.error(error)
            case .end:
                watcher.end()
            }
        }
    }
}
```

Zdrojový kód 7: Implementace operátoru `map`. Tento kód je jeden z jednodušších, protože jediné, co musí udělat je reagovat na datovou událost tím, že na její data aplikuje lambda funkci předanou parametrem. Při aplikaci transformace může dojít k vyhození výjimky a tím pádem se dále datovým tokem musí propagovat chybová událost s touto výjimkou místo datové.

Vzhledem k řetězící podstatě datových toků lze aplikovat virtuálně nekonečné množství operátorů. Použití operátoru `map`, následná práce s daty a další aplikování tatáž operátoru je zvláště na místě, pokud pracuje vývojář s rozsáhlou datovou strukturou a postupně se zbavuje přebytečných dat, aby nemusel zbytečně příliš hluboko přistupovat k instančním proměnným přes několik úrovní.

Například na Obrázku 3.2 je aplikován operátor k umocnění čísla na druhou. Pokud by byl aplikován znovu, výsledné čísla by byly umocněny podruhé.
V operátoru `map` lze vyhodit výjimku beze strachu, že někde dojde k problému, protože tato výjimka je automaticky zachycena a převedena na chybovou událost, která je dále poslána datovým tokem pro zpracování dalšími operátory, které reagují na chybové události a nakonec sledujícímu, který na ní může, ale nemusí, reagovat.



Obrázek 3.2: Aplikace operátoru `map`, který každé číslo celočíselného datového toku vynásobí samo sebou a výsledek posílá na výsledný datový tok.

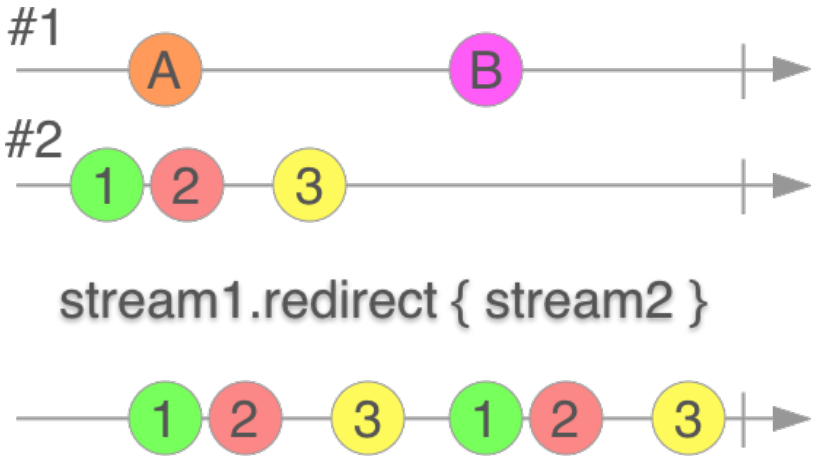
```
// Přepsání konstantou pomocí operátoru `map`.
let onStream = buttonOn.stream.click.map { true }
let offStream = buttonOff.stream.click.map { false }
DataStream.merge(onStream, offStream)
    .watch(data: { notificationsOn in
        label.text = "Notifications are \(notificationsOn ? "enabled" : "disabled")."
    })
    .disposed(by: ...)

// Výsledný typ operátoru `map` není předem daný a lambda funkce může vrátit
// jakýkoliv typ. S tímto typem potom pracují další zřetězené operátory.
dataService.randomUser()
    .map { user in
        // Zde je vrácen pár jména uživatele a pokud má adminské oprávnění.
        // Typ (name: String, isAdmin: Bool)
        (name: user.name, isAdmin: user.role.isAdmin)
    }

// V operátoru lze provádět jakkoli komplexní operace.
dataService.allUsers()
    .map { users in
        var friends = [] as [User]
        for user in users {
            if user.relationship == .friend {
                friends.append(user)
            }
        }
        return friends
    }
```

Zdrojový kód 8: Příklady využití operátoru `map`. Nejjednodušší případ je přepsání konstantní hodnotou (nebo i typem `Void`). Další možnost je pouhý přístup k instanční proměnné dat pro snazší napojení na další operátory. Využití tohoto operátoru ale není samozřejmě omezeno pouze na jednořádkové transformace. Lze využít jakékoliv konstrukce, které jsou dostupné v kódu, ale vývojář by se měl vyvarovat provádění vedlejších efektů mimo datový tok stejně tak jako číst jiná data, než jen ty, které jsou předány funkci parametrem.

3.7.2 Operátor Redirect

Metoda `redirect` slouží k nahrazení současného datového toku datovým tokem vráceným danou lambda funkcí. Podobně jako při metodě `map` probíhá k určité transformaci, avšak tato metoda vyžaduje vrátit typ `DataStream`, aby mohlo dojít k náhradě. Konceptuální model zobrazený grafem je na Obrázku 3.3. Příklad použití lze najít ve Zdrojovém kódu 9.

Z operátoru `redirect` lze vrátit i konstantní hodnotu použitím funkce `DataStream.just(data:)`. Ovšem tuto funkci (a její obdoby) je vhodné používat pouze pro určité komplexní operace, které jsou neřešitelné za použití operátoru `map`.



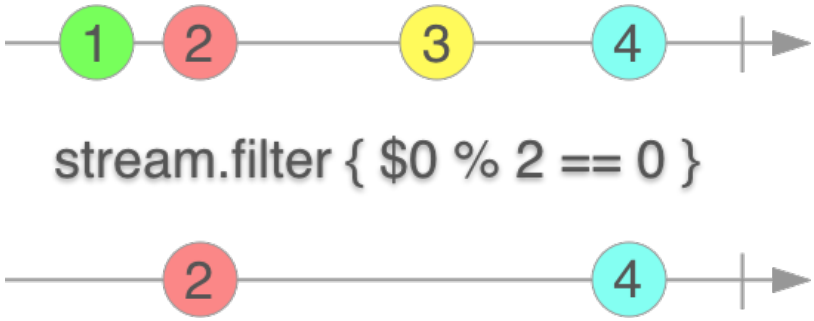
Obrázek 3.3: Aplikace operátoru `redirect` na vrchní stream (číslo 1) tak, že při průchodu datové události dojde k jejímu nahrazení jiným datovým tokem, na který se začne naslouchat.

```
1  // I přesto, že z toku získaného z tlačítka (stream.click) žádné data neproudí
2  // (z ťuknutí na tlačítko není jak vyprodukovat smysluplné data kromě samotného ťuknutí),
3  // výstupem operátoru `redirect` budou nové data, až přijdou ze serveru.
4  let newDataStream = refreshButton.stream.click
5      .redirect { [dataService] in
6          // Metoda `refreshData` vrací další tok, zde například tok polí novinek.
7          dataService.refreshData()
8      }
9
10 // Podmínkou zavolání metody `refreshData` je naslouchání na výsledek,
11 // stejně jako u všech jiných datových toků.
12 // Zde pouze dojde k napojení dat, která projdou tokem,
13 // na tabulku tak, aby vždy zobrazovala ty aktuální.
14 tableView.stream.connect(to: newDataStream, ...).disposed(by: ...)
```

Zdrojový kód 9: Příklad využití operátoru `redirect`. V tomto scénáři se má při ťuknutí na tlačítko poslat dotaz na back-end o nové data. Například v případě, že chce uživatel aktualizovat novinky. Získání nových dat automaticky způsobí překreslení buněk tabulky.

3.7.3 Operátor Filter

Stejně jako operátor `map`, i operátor `filter` funguje velmi podobně jako u kolekce. Jako jediný parametr bere lambda funkci, které jsou předány data události a na jejich základě vrací pravdivou či nepravdivou hodnotu. V případě pravdivé pokračuje událost datovým tokem dále, jak je vidět na Obrázku 3.4. Pokud vrátí funkce nepravdivou hodnotu, tak je zde pouze tato konkrétní událost zastavena a nepokračuje dále datovým tokem, tudíž všechny operátory (včetně naslouchací funkce) pod operátorem `filter` nereagují na tuto událost. Pár příkladů využití tohoto operátoru je ve Zdrojovém kódu 10.



Obrázek 3.4: Aplikace operátoru `Filter` tak, aby prošly pouze sudé čísla celočíselného datového toku.

```
let stream: Stream<User>
...
// Operátor `filter` nemá vliv na datový tok v případě vracení
// konstantní pravdivé hodnoty - všechny hodnoty jsou propuštěny.
stream.filter { _ in true }
// Při rozhodování lze samozřejmě přistoupit k proměnným datového objektu.
// Zde jsou propuštěni pouze uživatelé s řetězcem "admin" ve jméně.
stream.filter { user in user.name.contains("admin") }
    .watch(data: { admin in
        // Zde už se jistě pracuje pouze s administrátorskými účty.
    })
    .disposed(by: ...)
```

Zdrojový kód 10: Příklad použití operátoru `filter` v kódu jako identita a následně jako naivní kontrola adminského účtu.

3.7.4 Další operátory

Tuto čtveřici operátorů doplňuje řada dalších podřadnějších operátorů jako například `toArray`, `reduce`, `skip`, `uniqueTransition`, atd. Všechny tyto operátory jsou podrobně popsány v dokumentaci.

Mimo tyto funkcionální operátory má vývojář k dispozici i operátor `call`, který je určený k vyvolávání vedlejších účinků, které nesouvisí přímo s datovým tokem. Normálně jsou tyto vedlejší účinky prováděny na konci toku v lambda funkci naslouchající na události, ale může nastat situace, kdy je datový tok předáván dál, ale i tak je potřeba reagovat na události bez

jakéhokoliv zasahování do toku samotného. Vyvolávání vedlejších účinků nebo volání funkcí s vedlejšími účinky samozřejmě nic nebrání ani v transformovacích operátorech jako je `map` nebo `redirect`, ale tyto metody jsou především určeny pro funkcionální transformace bez vedlejších účinků.

Uživatel knihovny zároveň může samozřejmě jednoduše vytvořit své vlastní operátory pomocí rozšíření typu, jak lze vidět ve Zdrojovém kódu 11. Může rozšířit protokol `StreamType`, od kterého dědí obě třídy `DataStream` i `Streamer` nebo jen používanější třídu `DataStream`. Všechny operátory jsou definovány tímto způsobem, konkrétně rozšířením protokolu `StreamType`.

```
extension StreamType {
    public func log(_ message: String? = nil) -> DataStream<E> {
        return call { event in
            print([message, String(describing: event)]
                .filterNil()
                .joined(separator: ": "))
        }
    }
}
```

Zdrojový kód 11: Všechny operátory jsou definovány tímto způsobem, konkrétně rozšířením protokolu `StreamType`. K definování nového operátoru lze samozřejmě využít i již existujících operátorů. Například zde je využito operátoru `call` k definování debugovacího operátoru `log`.

Kapitola 4

Implementace knihovny StreamLibrary

Jak už bylo naznačeno, cílem této práce není jen návrh reaktivní knihovny, ale také její implementace. V této kapitole budou předvedeny nejdůležitější prvky knihovny a snad pomůže kód doplnit případné nejasnosti.

4.1 Rozhraní StreamType

Rozhraní `StreamType` implementují třídy, které mají mít vlastnosti datového toku. Přesněji možnost naslouchání na jejich události. Rozhraní požaduje pouze implementaci jediné metody, a to `watch(watcher:)`, která bere jako jediný parametr objekt typu `Watcher`, kde je za pomoci generiky zajištěna kompatibilita typů.

Ve Zdrojovém kódu 12 klíčové slovo `associatedtype` následované identifikátorem generického typu, který bude využíván v rozhraní. Tento typ je později definován v třídě implementující toto rozhraní pomocí klíčového slova `typealias` následovaného konkrétním typem.

Dále jsou na řádcích 15 a 20 jsou definovány standardní implementace metod požadovaných rozhraním na řádcích 6 a 8. Díky tomuto stačí třídě, která implementuje toto rozhraní, splnit požadavek metody na řádku 4 a ty další zmíněné dostane "zdarma".

```
protocol StreamType {
    associatedtype E

    func watch(using watcher: Watcher<E>) -> Disposable

    func watch(_ eventHandler: @escaping Watcher<E>.EventHandler) -> Disposable

    func watch(
        data: @escaping (E) -> Void,
        error: @escaping (Error) -> Void,
        end: @escaping () -> Void) -> Disposable
}

extension StreamType {
    func watch(_ eventHandler: @escaping Watcher<E>.EventHandler) -> Disposable {
        let watcher = Watcher<E>(eventHandler: eventHandler)
        return watch(using: watcher)
    }

    func watch(
        data: @escaping (E) -> Void = { _ in },
        error: @escaping (Error) -> Void = { _ in },
        end: @escaping () -> Void = { }) -> Disposable {
        let watcher = Watcher<E>(eventHandler: { event in
            switch event {
            case .data(let dataValue):
                data(dataValue)
            case .error(let errorValue):
                error(errorValue)
            case .end:
                end()
            }
        })

        return watch(using: watcher)
    }
}
```

Zdrojový kód 12: Implementace rozhraní `StreamType`.

4.2 Třída DataStream

Generická třída datových toků `DataStream` obsahuje identifikátor (ve Zdrojovém kódu 13 na řádku 2) pro jednoznačné označení daatového toku. Toto je využito v některých operátorech. Například v operátoru `merge`.

Proměnná `watchHandler` je lambda funkce, která bere jako parametr objekt typu `Watcher` a vrací objekt typu `Disposable`. Tato lambda funkce je předána při vytváření objektu a slouží k vytvoření datového toku nebo převedení zpětného volání na datový tok.

```
final class DataStream<E>: StreamType {
    let id = UUID().uuidString

    typealias WatchHandler = (Watcher<E>) -> Disposable
    let watchHandler: WatchHandler

    init(watchHandler: @escaping WatchHandler) {
        self.watchHandler = watchHandler
    }

    func watch(using watcher: Watcher<E>) -> Disposable {
        return watchHandler(watcher)
    }
}
```

Zdrojový kód 13: Implementace generické třídy `DataStream`.

Pro jednodušší vytváření datových toků z konstantních hodnot jsem rozšířil typ `DataStream` o pomocné statické funkce (Zdrojový kód 14).

```
extension DataStream {
    static func just(event: Event<E>) -> DataStream<E> {
        return DataStream<E>(watchHandler: { watcher in
            watcher.event(event)
            watcher.end()
            return Disposable()
        })
    }

    static func just(data: E...) -> DataStream<E> {
        return just(data: data)
    }

    static func just(data: [E]) -> DataStream<E> {
        return DataStream<E>(watchHandler: { watcher in
            watcher.data(data)
            watcher.end()
            return Disposable()
        })
    }

    static func just(error: Error) -> DataStream<E> {
        return just(event: .error(error))
    }
}
```

Zdrojový kód 14: Pomocné inicializační funkce třídy `DataStream`.

4.3 Rozhraní WatcherType

Rozhraní `WatcherType` implementují třídy, které mají mít vlastnosti naslouchače datového toku, a to přijímat události datového toku. Toto rozhraní, podobně jako `StreamType` požaduje pouze implementaci jediné metody, a to `event(event:)` (ve Zdrojovém kódu 15 na řádku 4), která bere jako jediný parametr objekt typu `Event`, kde je za pomoci generiky zajištěna kompatibilita typů.

Stejně jako v případě rozhraní `StreamType` jsou v rozšíření rozhraní (blok kódu začínající klíčovým slovem `extension` a následovaný typem, který rozšiřuje) definovány 4 metody, které nemusí implementovat sama třída, která implementuje rozhraní `WatcherType`.

```
1  protocol WatcherType {
2      associatedtype E
3
4      func event(_ event: Event<E>)
5
6      func data(_ data: [E])
7
8      func data(_ data: E...)
9
10     func error(_ error: Error)
11
12     func end()
13 }
14
15 extension WatcherType {
16     func data(_ data: [E]) {
17         for information in data {
18             event(.data(information))
19         }
20     }
21
22     func data(_ data: E...) {
23         // call the `data` function that deals with arrays
24         self.data(data)
25     }
26
27     func error(_ error: Error) {
28         event(.error(error))
29     }
30
31     func end() {
32         event(.end)
33     }
34 }
```

Zdrojový kód 15: Implementace generického rozhraní `WatcherType`.

4.4 Třída Watcher

Třída určená pro sledování datového toku, třída `Watcher`, implementuje pouze rozhraní `WatcherType` a definuje svůj vlastní typ `EventHandler` ve Zdrojovém kódu 16 na řádku 2 pro snazší znovupoužití a případně jednoduché změny.

Další zajímavost může být na řádku 10 použití třídy `DispatchQueue` ke spuštění lambda funkce předané parametrem metodě `watch`. Důvod použití hlavního vlákna je ten, že se očekává v této metodě hlavně interakce s uživatelským rozhraním, jenž nelze provádět ve většině případech na jiném než hlavním vlákně [10].

```
1  final class Watcher<E>: WatcherType {
2      typealias EventHandler = (Event<E>) -> Void
3      private let eventHandler: EventHandler
4
5      init(eventHandler: @escaping EventHandler) {
6          self.eventHandler = eventHandler
7      }
8
9      func event(_ event: Event<E>) {
10         DispatchQueue.main.async {
11             self.eventHandler(event)
12         }
13     }
14 }
```

Zdrojový kód 16: Implementace generické třídy `Watcher`.

4.5 Výčet Event

Výčet typu `Event` není nijak složitý a jak bylo již zmíněno v návrhu, obsahuje pouze 3 případy, jejíž přesné znění lze vidět ve Zdrojovém kódu 17.

```
1  enum Event<T> {
2      case data(T)
3      case error(Error)
4      case end
5  }
```

Zdrojový kód 17: Implementace generického výčtu `Event`.

4.6 Třída Streamer

Generická třída `Streamer` implementuje, jak lze vidět ve Zdrojovém kódu 18, obě rozhraní `StreamType` i `WatcherType`. Tudíž se chová jako datový tok, na který lze naslouchat pomocí metody `watch` a zároveň také jako sledující, kterému lze v libovolný čas poslat nějakou

událost pomocí metody `event` (nebo jakékoliv jiné pomocné metody, která byla implementována v rozhraní `WatcherType` a odvíjí se od této metody).

Sledování funguje jednoduše tak, že si při započetí sledování objekt typu `Streamer` uloží referenci na objekt typu `Watcher` a jakékoliv události jsou přeposlány tomuto objektu.

```
1  final class Streamer<E>: StreamType, WatcherType {
2      private var watcher: Watcher<E>?
3
4      func watch(using watcher: Watcher<E>) -> Disposable {
5          self.watcher = watcher
6          return Disposable {
7              self.watcher = nil
8          }
9      }
10
11     func event(_ event: Event<E>) {
12         DispatchQueue.main.async {
13             self.watcher?.event(event)
14         }
15     }
16 }
```

Zdrojový kód 18: Implementace generické třídy `Streamer`.

Kapitola 5

Distribuce knihovny StreamLibrary

Knihovna *StreamLibrary* musí být pro uživatelskou přívětivost jednoduchá ke stáhnutí, vyzkoušení a následné používání. Jelikož nemá žádné závislosti kromě knihoven *UIKit* a *Foundation* vestavěných v mobilních Apple zařízeních, není třeba řešit žádné závislostní stromy, ovšem manuální připojení knihovny *StreamLibrary* do projektu a následné udržování kompatibilních verzí je časově náročné.

Jako řešení těchto zdlouhavých operací slouží správce závislostí (nebo také správce balíčků). Jedná se o nástroj pro automatickou instalaci, aktualizaci, nastavení a případnou odinstalaci knihoven obvykle definovaných v nějakém souboru.

Při vyvíjení aplikací na iOS si lze vybrat ze 3 nejpoužívanějších správců závislostí. Sada mobilních aplikací, která je součástí této práce, využívá pouze *Carthage*, aby se kvůli otestování instalovalo co nejméně závislostí a zároveň bylo používání intuitivní.

Jediné, co se vyžaduje po vývojáři, který využívá jeden z těchto správců, je vytvořit soubor závislostí, ve kterém jsou deklarovány všechny přímé závislosti projektu. Všechny nepřímé jsou vyhledány a stáhnuty podle potřeby.

5.1 Správce Swift Package Manager

Swift Package Manager (také SwiftPM nebo jen SPM) je součástí jazyka Swift od verze 3.0 [8]. Jedná se o nový standard vyvíjený přímo společností Apple.

Systém SwiftPM funguje tak, že jsou nejprve definovány produkty, které jsou výsledky jednotlivých cílů projektu. Poté se definují všechny závislosti projektu zároveň s jejich relativní či absolutní URL a verzí. Následuje seznam cílů projektu, kde se rozlišuje mezi standardním a testovacím cílem. Každý cíl definuje pole svých závislostí podle jejich jmen knihoven. Příklad takového souboru lze najít ve Zdrojovém kódu 19.

V současné době tento správce závislostí nepodporuje závisení na knihovnách specifických platforem Apple (např. UIKit). Další nevýhodou je, že vyžaduje od vývojáře, aby používal specifickou strukturu adresářů.

```swift
// swift-tools-version:4.0
import PackageDescription

let package = Package(
    // Jméno celého balíčku v případě jeho referencování.
    name: "TexasHoldem",
    products: [
        // Výsledek překladu pomocí `swift build`.
        .executable(name: "Texas Holdem", targets: ["TexasHoldem"]),
    ],
    // Pole závislostí, kam se nahlíží v seznamu cílů `targets`.
    dependencies: [
        // Závislost, která může být odkázána přes její název.
        // URL může být také lokální. Verze se bere z git tagů.
        .package(url: "https://gitlab.com/MatyasKriz/stream-library", from: "1.0.0"),
    ],
    targets: [
        // Hlavní cíl k překladu.
        .target(
            name: "TexasHoldem",
            dependencies: ["StreamLibrary"]),
        // Cíl k testování, zde pouze k testování hlavního cíle.
        .testTarget(
            name: "TexasHoldemTests",
            dependencies: ["TexasHoldem"]),
    ]
)
```

Zdrojový kód 19: Příklad pomocného souboru v aplikaci, která by využila knihovny *StreamLibrary* za použití Swift Package Manager. Soubor je poněkud delší oproti alternativám z toho důvodu, že SwiftPM potřebuje znát všechny informace k tomu, aby mohl úspěšně vygenerovat projekt se všemi závislostmi uvnitř.

5.2 Správce Carthage

Carthage je decentralizovaný správce závislostí, který používá pouze informace z gitu a nevyžaduje deklaraci žádných pomocných souborů. Klientská aplikace pak už jen definuje soubor *Cartfile*, ve kterém jsou závislosti deklarovány. Příklad tohoto souboru je ve Zdrojovém kódu 20.

Princip fungování Carthage je v tom, že automaticky vyřeší závislosti, stáhne všechny potřebné zdrojové soubory, přeloží je a výsledný binární framework musí vývojář najít a připojit ke svému projektu. Tento proces může být poněkud zdlouhavý pro více závislostí a navíc je třeba ho opakovat při přidání nových závislostí.

Tento správce závislostí podporuje spravování závislostí pro vývoj na všechny platformy Apple, ať už je kód v jazyce Swift či Objective C. Samotný program `carthage` je spustitelný pouze na Mac OS.

```
1 git "https://gitlab.com/MatyasKriz/stream-library" ~> 1.0
```

Zdrojový kód 20: Příklad pomocného souboru v aplikaci, která by využila knihovny *StreamLibrary* za použití Carthage. Šipka ~> značí, že chce vývojář aplikace aktualizovat verze pouze do 2.0, kdy je podle sémantického verzování zvykem očekávat změny API knihovny. Soubor není v žádném programovacím jazyce, a tudíž je do určité míry limitován v komplexnosti nastavení závislostí. Soubor je velmi krátký díky tomu, že Carthage negeneruje žádný projekt ani jeho kombinaci se závislostmi, pouze přeloží závislosti a nechá na vývojáři, aby je přidal do projektu manuálně.

5.3 Správce CocoaPods

Na rozdíl od Carthage je CocoaPods centralizovaný správce závislostí, jehož princip spočívá v tom, že obsahuje jeden velký spravovaný repozitář zvaný *Specs*, kde jsou všechny závislosti (tzv. *pody*) nahrané vývojáři knihoven. Repozitář obsahuje přes 60 tisíc knihoven, které jsou používány ve více než 3 milionech aplikací [2].

Pro použití CocoaPods musí klientská aplikace definovat stejně jako u Carthage pomocný soubor se závislostmi, zde pojmenovaný *Podfile*. Příklad tohoto souboru je ve Zdrojovém kódu 21.

Na rozdíl od Carthage ale tento správce nepřekládá zdrojové soubory, ale pouze automaticky vytvoří další projekt, ve kterém se nachází všechny závislosti a spolu se současným projektem vytvoří nový soubor (tzv. workspace). Po jeho otevření je vše připraveno k používání. Nevýhodou tohoto přístupu je, že při vyčištění přeložených souborů se musí všechny závislosti znovu přeložit. Na druhou stranu tento přístup umožňuje snadné a rychlé nahlédnutí do zdrojových kódů závislostí v případě nesprávného chování nebo pádů.

Díky centralizované charakteristice systému nemusí uživatel tohoto správce definovat URL git repozitáře, jelikož názvy knihoven v repozitáři *Specs* musí být unikátní. Navíc lze použít i URL git repozitáře stejně jako je tomu u Carthage.

Co se týče podpory platforem a programovacích jazyků je na tom CocoaPods stejně jako Carthage, kdy program `pod`, který řeší samotnou správu závislostí, je spustitelný pouze na Mac OS.

Nevýhodou tohoto správce závislostí je, že při prvním spuštění (nebo po smazání pomocných souborů) trvá proces získávání závislostí déle, protože se musí stáhnout repozitář *Specs*.

```
platform :ios, '9.0'
target 'TexasHoldem' do
    use_frameworks!
    pod 'StreamLibrary', '~> 1.0'
end
```

Zdrojový kód 21: Příklad pomocného souboru v aplikaci, která by využila knihovny *StreamLibrary* za použití CocoaPods. Zde není třeba žádného URL. Šipka ~> zde má stejný význam jako v Carthage. Soubory jsou v jazyce Ruby, což umožňuje mnohem komplexnější nastavení závislostí. Není třeba definovat testovací cíl, pokud nemá žádné závislosti (kromě jiného z cílů), protože CocoaPods negeneruje projekt, ale jen vytvoří kombinaci současného projektu a závislostí.

Kapitola 6

Demonstrační ukázky použití knihovny StreamLibrary v kódu

Tato kapitola se věnuje demonstraci reaktivního programování na krátkých příkladech, které popisují použití datových toků a operátorů na ně aplikovaných.

Způsoby, jakými jsou tyto situace řešeny nejsou jediné, jak tomu většinou v programování je. Tudíž některé přístupy či použité operátory by se daly nahradit jinými, možná ve prospěch čistějšího kódu.

V případě, že je třeba uživatele zaregistrovat, lze na pár řádcích kódu přidat podporu pro jeho přivítání již během toho, co píše své jméno. K tomu, aby se pouze na začátku objevilo přivítání s nějakým generickým oslovením, lze použít operátory `concat`, `skip` a pomocnou funkci `just` tak, jako je znázorněno ve Zdrojovém kódu 22.

```
1  let nameStream = nameField.stream.text
2      .skip(1)
3
4  let messageStream = DataStream.concat(.just(data: "Handsome"), nameStream)
5      .map { name in "Hello, \(name)!" }
6
7  welcomeLabel.stream.connect(stream: messageStream)
8      .disposed(by: disposeBag)
```

Zdrojový kód 22: Vítání nových uživatelů jejich aktuálním jménem už při jeho zadávání. Na řádku 4 lze vynechat identifikátor třídy `DataStream` díky překladači, který si dokáže dovodit správný typ, protože tato funkce nebere nic jiného než objekt typu `DataStream`.

Další použití na registrační obrazovce je tlačítko "zaregistrovat", které má být neaktivní, dokud délka hesla nepřekročí minimální povolený limit. Vývojář může aplikovat operátor `map` pro získání délky řetězce a následně při sledování událostí aktivovat tlačítko jako je ukázáno ve Zdrojovém kódu 23. Toto řešení okamžitě reaguje na změny textu a v případě, že uživatel opět zkrátí heslo pod minimální povolený limit, tlačítko je opět deaktivováno. Díky podstatě datového toku `stream.text` na uživatelském prvku `UITextField` se posílá i

počáteční hodnota, tedy prázdný řetězec, což vede k tomu, že hned při započetí naslouchání se tlačítko deaktivuje a není třeba nastavovat jeho stav kdekoliv jinde v kódu.

```
passwordField.stream.text
    .map { password in password.count }
    .watch(data: { passwordLength in
        self.signUpButton.isEnabled = passwordLength >= 6
    })
    .disposed(by: ...)
```

Zdrojový kód 23: Aktivace tlačítka jen v tom případě, že má heslo požadovanou délku.

Pro zjednodušení aktualizace dat za použití uživatelského prvku `UIRefreshControl` lze využít proměnnou `refresh` na objektu `stream`. Konvenční způsob řešení tohoto problému je nastavení odposlouchávání na změnu stavu pomocí metody `addTarget(target:action:events:)`, která zavolá metodu při aktivaci prvku. Toto sledování by mělo být odstraněno, když už není potřeba na něj reagovat.

```
let usersStream = refreshControl.stream.refresh
    .redirect { [dataService] in
        dataService.users()
    }

tableView.stream.connect(
    to: usersStream,
    with: TypedTableViewCellIdentifier<UserCell>(identifier: "User")
).disposed(by: disposeBag)
```

Zdrojový kód 24: Použití uživatelského prvku `UIRefreshControl` pro aktualizaci dat za pomoci operátoru `redirect` a následného napojení těchto nových dat na tabulku.

Za pomoci operátoru `redirect` lze aktualizovat tabulku při jakékoliv akci, například při zmáčknutí tlačítka nebo při zatáhnutí uživatelského prvku `UIRefreshControl` dolů. Tento způsob využívá, jak lze vidět ve Zdrojovém kódu 25, objekt typu `Streamer` pro snadné zaslání zprávy kdykoliv, kdy uživatel žádá o nová data.

```
let loadUsersTrigger = Streamer<Void>()

...

let friendCountStream = DataStream.merge(
        loadUsersTrigger,
        refreshControl.stream.refresh,
        manualRefreshButton.stream.click
    )
    .redirect { [dataService] in
        dataService.users()
    }
    .map { users in
        // Spočítání přátel mezi uživateli.
        let friendCount = users.reduce(0) { result, user in
            result + (user.isFriend ? 1 : 0)
        }
        return "You currently have \(friendCount) friends."
    }

friendCountLabel.stream.connect(stream: friendCountStream)
    .disposed(by: ...)
}

...

// Pro obnovení dat může být objektu typu `Streamer` poslána
// událost s prázdnými daty - `()`.
loadUsersTrigger.data(())
```

Zdrojový kód 25: Kdykoliv se zatáhne uživatelský prvek `UIRefreshControl`, zmáčkne tlačítko `UIButton` nebo projde objektem typu `Streamer` datová událost, zavolá se metoda pro získání nových dat. Tyto data lze poté napojit na tabulku nebo s nimi provádět jiné akce. Zde dochází k převodu uživatel na počet přátel současného uživatele za pomoci operátoru `map` a funkcionální metody na poli `reduce`, která funguje jako akumulátor.

Knihovna *StreamLibrary* počítá s tím, že se občas něco nepodaří. K chybám dochází v každé aplikaci ať už v případě neoprávněného přístupu k obsahu či nepřipojení k internetu. Chyby, které nastanou už při vytváření datového toku, nebo až při aplikování transformačního operátoru, jsou propagovány datovým tokem až na konec, kde na ně může čekat sledující a reagovat na ně.

Výhoda datových toků ovšem nekončí jen u toho, že na ně lze reagovat při naslouchání. To samotné není dostatečně přidaná hodnota k tomu, aby byla knihovna *StreamLibrary* užitečná. Koncept datových toků umožňuje vytváření zjednodušujících operátorů, které reagují na určitý typ událostí, mezi ně patří i chybové události. Například operátor `retry` použitý ve Zdrojovém kódu 26 zajistí, že než projde chybová událost k dalším operátorům a případně ke sledujícímu, zkusí provést alespoň jedno (avšak počet lze nastavit na jakékoliv kladné číslo) zopakování asynchronní operace.

```
dataService.users()
    // Pokud se operace nepovede, zkus to ještě 3x.
    .retry(3)
    .watch(
    data: { users in
        print("We have all of our users!")
    },
    error: { error in
        print("Not even 3 retries helped here.")
    })
    .disposed(by: ...)
```

Zdrojový kód 26: Příklad použití operátoru opakování operace. Tohoto se využívá hlavně u internetových požadavků, které můžou mít problémy při horším internetovém připojení.

Další případ je bezstarostné převedení získaných dat pomocí JSON deserializéru na model. V případě úspěchu se pracuje dále s datovou událostí s modelem, v případě neúspěchu s vyvolanou chybovou událostí. Tento způsob práce je naznačen ve Zdrojovém kódu 27. Zároveň pokud má vývojář nějaké smysluplné data, kterými dává smysl nahradit žádné data v případě chyby, může tak učinit pomocí operátoru `recover`, který nahrazuje chybovou událost datovou událostí s předanými daty. Všechny další operátory a následně sledující již neobdrží informace o chybě, ale pouze tyto data, které mají případnou chybu nahradit.

```
dataService.rawUserData()
    .map { data in
        try JSONDecoder().decode(User.self, from: data)
    }
    .recover(User(name: "Randy the Rando", isFriend: true))
    .map { user in "Hello, \(user.name)!" }
```

Zdrojový kód 27: Příklad možného vyhození výjimky v operátoru `map`. Případné chybové události (tedy pokud je výjimka vyhozena a převedena) řeší operátor `recover` nahrazením chybové události datovou s výchozím uživatelem.

Kapitola 7

Případová studie - Použití knihovny StreamLibrary v Prohlížeči uživatelů GitHubu

Hlavní výhodou reaktivního programování pomocí datových toků je udržování stavu uvnitř toku místo toho, aby byl tento stav k dispozici na místech, které s ním vůbec nesouvisí a neměly by do něj zapisovat ani ho číst.

V této aplikaci byla použita knihovna *StreamLibrary* pro demonstraci použití datových toků a zároveň zjednodušení psaní aplikací při použití tabulky (`UITableView`), což je v mobilních aplikacích nejpoužívanější prvek uživatelského rozhraní, pro zobrazení strukturovaných nebo hierarchických informací [11].

Po zapnutí aplikace je uživatel uvítán načítáním tabulky uživatelů GitHubu. Toto načítání je implementováno jako získání uživatelských informací na náhodné pozici z API GitHubu. Získané data obsahují obecné informace o uživateli – přezdívka, jméno, email, URL profilového obrázku, URL repozitářů, atd. V těchto datech je sice k dispozici URL obrázku, ale ne samotná data, ze kterých by bylo možné vytvořit bitmapovou reprezentaci a ukázat ji uživateli. Z toho důvodu je potřeba pro každého uživatele stáhnout data obrázku zvlášť.

Ve Zdrojovém kódu 28 je použita statická funkce `DataStream.concat`, jenž stahuje obrázky sériově, tedy při úspěchu stažení jednoho obrázku započne stahování dalšího. Pokud by se ale vývojář rozhodl, že chce stahovat všechny obrázky paralelně, stačí místo toho použít funkci `DataStream.merge`, jenž zajistí, že při započetí sledování tohoto toku se začnou stahovat obrázky všech uživatelů současně. Operátor `toArray` potom zajistí, že se počká na všechny obrázky než se pošle pole uživatelů s potřebnými daty pro správné zobrazení v aplikaci.

```
let url = URL(string: "\(Constants.apiUrl)/users")!
return NetworkService.GET(url: url, type: [UserDTO].self)
    .redirect { userDTOs -> DataStream<[User]> in
        DataStream.concat(
            // Jelikož `DataStream.concat` spojuje toky dat, dochází zde
            // k transformaci pole neúplných uživatelských dat na pole toků.
            userDTOs.map { userDTO in
                // Metoda `avatar` bere URL obrázku a vrací `DataStream`,
                // kam pošle obrázek sestavený ze stažených dat.
                self.avatar(url: userDTO.avatarUrl).map { image in
                    // Po přijetí obrázku jsou data transformována
                    // na strukturu uživatele za použití neúplných
                    // dat uživatele v proměnné `userDTO`.
                    User(avatar: image,
                         login: userDTO.login,
                         accountUrl: userDTO.accountUrl)
                }
            }
        )
        // `toArray` akumuluje datové události a při ukončující události
        // pošle datovou událost s polem akumulovaných dat.
        .toArray()
    }
```

Zdrojový kód 28: Tělo funkce pro sériové stahování uživatelských obrázků za použití *StreamLibrary*.

Sériového stahování obrázků pomocí *GCD* není racionálně možné. Pokud bychom chtěli dosáhnout stejného výsledku za použití `Operation`, nedostali bychom se ani zdaleka k tak flexibilnímu kódu jako za použití datových toků. Příklad takového snažení je Zdrojový kód 29, kde jsou definovány potřebné třídy a následně Zdrojový kód 30, kde jsou tyto třídy využity k dosáhnutí podobného výsledku.

```swift
class UsersOperation: Operation {
    private let callback: ([UserDTO]) -> Void

    override func main() {
        let randomPosition = Int(arc4random_uniform(Constants.randomNumberLimit))
        var request = URLRequest(url: URL(string: "\(Constants.apiUrl)/users")!)
        request.httpMethod = "GET"
        request.send { data in
            self.callback(try! JSONDecoder().decode([UserDTO].self, from: data))
        }
    }
}

class UserAvatarOperation: Operation {
    private var _finished: Bool = false
    override var isFinished: Bool {
        get { return _finished }
        set {
            guard _finished != newValue else { return }
            willChangeValue(forKey: "isFinished")
            _finished = newValue
            didChangeValue(forKey: "isFinished")
        }
    }

    private let avatarUrl: URL
    private let callback: (UIImage?) -> Void

    override func main() {
        var request = URLRequest(url: avatarUrl)
        request.httpMethod = "GET"

        request.send(callback: { data in
            self.callback(UIImage(data: data))
            self.isFinished = true
        })
    }
}
```

Zdrojový kód 29: Podtřídy `Operation` (pro zkrácení bez inicializačních funkcí), které jsou potřeba k tomu, aby se dalo využít závislostí. Kromě potřebných proměnných také využívám lambda funkcí `callback` pro definování vlastního chování se získanými daty po dokončení operace.

```
1  var users = [] as [User]
2  let queue = OperationQueue()
3  queue.addOperation(UsersOperation(callback: { userDTOs in
4      let operations = userDTOs.map { userDTO in
5          UserAvatarOperation(avatarUrl: userDTO.avatarUrl, callback: { image in
6              users.append(
7                  User(avatar: image, login: userDTO.login, accountUrl: userDTO.accountUrl)
8              )
9          })
10     }
11
12     var iterator = operations.makeIterator()
13     var previousOperation: Operation?
14     while let operation = iterator.next() {
15         if let previousOperation = previousOperation {
16             operation.addDependency(previousOperation)
17         }
18         queue.addOperation(operation)
19         previousOperation = operation
20     }
21 }))
```

Zdrojový kód 30: Získání uživatelů a následně jejich profilových obrázků pomocí podtříd `Operation` definovaných výše. Pro zkrácení je vynecháno upozornění tabulky, aby se překreslila s novými daty při získání posledního obrázku. Detekce posledního uživatele by se dala provést v lambda funkci `callback` operace `UserAvatarOperation`, kde bychom upozornili tabulku, pokud se ID uživatele, jehož obrázek se stáhl, rovná ID posledního uživatele v poli neúplných uživatelských dat – `userDTOs`.

Pokud by se vývojář rozhodl, že chce, aby se obrázky stahovaly paralelně, musí minimálně zakomentovat řádek 16. Po testovací fázi je pravděpodobné, že se nerelevantní konstrukce jako cyklus a pomocné proměnné, které zde slouží pro nastavení závislostí, smažou.

V případě, že bude v budoucnu potřeba stahovat obrázky znovu sériově, jde o mnohem více práce než jen přepsání `DataStream.merge` zpět na `DataStream.concat`.

I přesto jsou změny mezi sériovým a paralelním spouštěním asynchronních operací vedlejší oproti tomu, co všechno je třeba udělat pro správné fungování za použití `Operation` a `OperationQueue`.

Aby fungoval systém závislostí `Operation` správně, musí vývojář sám přetížit alespoň proměnnou `isFinished` a spravovat její stav. Zároveň s tím musí volat příslušné metody při její změně, aby `OperationQueue` věděl, že může spustit závislé operace a zároveň mohl správně dealokovat již dokončené.

Po získání potřebných dat je dalším cílem obnovit uživatelské rozhraní tak, aby obsahovalo aktuální informace. Obvyklý způsob je využitím metod rozhraní `UITableViewDelegate`. Celkově jsou potřeba 3 – počet sekcí, počet položek a následně získání samotné buňky pro určitý index.

Knihovna *StreamLibrary* toto usnadňuje přímočarým programovacím rozhraním, které pouze potřebuje identifikátor buňky a aby byla buňka zaregistrovaná mezi buňkami tabulky.

Navíc je třeba, aby buňky implementovaly rozhraní `StreamUpdatable`, které definuje pouze metodu `update(newValue:)` jako jednotné rozhraní pro nastavení stavu při inicializaci buňky. Tento způsob aktualizace buněk je přístupný přes instanční proměnnou `stream`, jak je naznačeno ve Zdrojovém kódu 31.

```
tableView.stream.connect(
    to: users(),
    with: TypedTableViewCellIdentifier<UserCell>(identifier: "User")
).disposed(by: disposeBag)
```

Zdrojový kód 31: Napojení toku uživatelů na tabulku.

Zobrazování dat v podobě buněk je důležitá část tabulky, ale ještě důležitější je reakce na uživatelovy podněty. Reakce je obvykle vykonána tak, že se opět implementuje metoda `tableView(tableView:indexPath:)` z rozhraní `UITableViewDelegate`, kde má programátor stále k dispozici jen index buňky a musí ještě sám zjistit hodnotu modelu na daném indexu.

Knihovna *StreamLibrary* toto abstrahuje za použití třídy, která získává tyto zprávy a převádí je do světa datových toků, kde jsou k dispozici opět přes objekt `stream`. Ve Zdrojovém kódu 32 je tohoto využito.

```
tableView.stream.modelSelected(User.self)
    .watch(data: { [navigationController] user in
        // Přechod na jinou obrazovku za použití `UINavigationController`.
        if let detailController =
        self.storyboard?.instantiateViewController(withIdentifier: "Detail")
        as? DetailController {
            detailController.update(newValue: user)
            navigationController?.pushViewController(
                detailController,
                animated: true
            )
        }
    })
    .disposed(by: disposeBag)
```

Zdrojový kód 32: Reakce na dotek určité buňky. Pomocí jejího indexu se zjistí model na daném indexu a ten je poslán datovým tokem. Zde se na získané uživatelské data reaguje přechodem na obrazovku s detailem uživatele.

Kapitola 8

Závěr

Ve své práci jsem zmínil několik současných metod vykonávání asynchronních operací na iOS jako například technologie *Grand Central Dispatch* a diskutoval jejich nedostatky v moderních aplikacích.

Mým hlavním cílem této bakalářské práce byl návrh a implementace jednoduché softwarové knihovny *StreamLibrary* pro asynchronní práci s datovými toky a operátory k jejich transformaci a jiným komplexnějším úkonům. Dále jsem představil nejdůležitější operátory nad datovými toky a jejich implementaci.

Dalším mým cílem bylo představit metodiku práce s datovými toky v aplikaci předvedením několika příkladů datových toků při častých úkonech v moderních aplikacích na iOS.

Ucelenější pohled na datové toky jsem prezentoval v kapitole 7 s případovou studií aplikace pro načtení náhodných uživatelů ze stránky GitHub a následně zobrazení jejich základních informací. Tato aplikace hojně využívá datové toky jak k naslouchání na uživatelské akce, tak k vykonávání asynchronních operací.

Osobně se domnívám, že popularita reaktivního programování stále poroste, což usuzuji z rostoucího počtu uživatelů technologií *React* a *Vue* ve světě webových stránek. Proto cítím, že je důležité vyzkoušet tyto koncepty i na platformách Apple, protože jde o koncept bezpečnější a hlavně méně náchylný k chybám.

Jako pokračování práce na knihovně *StreamLibrary* bych označil větší volnost výběru vlákna, na kterém se má kód provádět a podporu alespoň pro Mac OS. Knihovnu sice lze používat na Mac OS bez jakýchkoliv úprav, avšak bez podpory datových toků na prvcích uživatelského rozhraní.

Literatura

[1] *Automatic Reference Counting*. [Online; navštíveno 28.04.2019].
URL https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html

[2] *Cocoapods*. [Online; navštíveno 22.04.2019].
URL https://cocoapods.org

[3] *DispatchQoS documentation*. [Online; navštíveno 20.03.2019].
URL https://developer.apple.com/documentation/dispatch/dispatchqueue

[4] *DispatchQueue documentation*. [Online; navštíveno 20.03.2019].
URL https://developer.apple.com/documentation/dispatch/dispatchqos

[5] *Operation documentation*. [Online; navštíveno 22.03.2019].
URL https://developer.apple.com/documentation/foundation/operation

[6] *Prioritize work with DispatchQoS*. [Online; navštíveno 20.03.2019].
URL https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/PrioritizeWorkWithQoS.html

[7] *ReactiveX introduction*. [Online; navštíveno 03.04.2019].
URL http://reactivex.io/intro.html

[8] *Swift Package Manager*. [Online; navštíveno 22.04.2019].
URL https://swift.org/package-manager

[9] *Thread documentation*. [Online; navštíveno 22.03.2019].
URL https://developer.apple.com/documentation/foundation/thread

[10] *UIKit*. [Online; navštíveno 05.05.2019].
URL https://developer.apple.com/documentation/uikit

[11] *UITableView documentation*. [Online; navštíveno 14.04.2019].
URL https://developer.apple.com/documentation/uikit/uitableview

[12] *Design patterns: elements of reusable object-oriented software*. Addison-Wesley professional computing series, Boston: Addison-Wesley, 1995, ISBN 0-201-63361-2.

[13] Elliott, C.; Hudak, P.: Functional reactive animation. *ACM SIGPLAN Notices*, ročník 32, č. 8, 1997: s. 263–273, ISSN 1558-1160.

[14] Maier, I.; Odersky, M.: Deprecating the Observer Pattern with Scala.React. 2012.

[15] Staltz, A.: *RxMarbles: Interactive diagrams of Rx Observables.* [Online; navštíveno 13.04.2019].
URL https://rxmarbles.com

[16] Sutter, H.: *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software.* [Online; navštíveno 12.04.2019].
URL http://www.gotw.ca/publications/concurrency-ddj.htm

[17] Wan, Z.; Taha, W.; Hudak, P.: Real-time FRP. *ACM SIGPLAN Notices*, ročník 36, č. 10, 2001, ISSN 03621340.

[18] Wan, Z.; Taha, W.; Hudak, P.: Event-driven FRP. Springer Verlag, 2002, ISBN 354043092X, ISSN 03029743, s. 155–172.