FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Aplikace funkcionálního programování ve vývoji podnikových aplikací
Student:	Jan Hanuš
Vedoucí:	Ing. Jiří Daněček
Studijní program:	Informatika
Studijní obor:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

Prostudujte vhodnost použití funkcionálního programování v jazyku Scala pro tvorbu enterprise aplikací a doménového modelování.
Vyberte vhodné Scala frameworky pro implementaci streamoveho a reaktivniho API a CQRS persistence.
Zvolené frameworky použijte pro vytvoření ukázkové aplikace client-server, ve které budete ilustrovat použití funkcionálních návrhových vzorů a funkcionální přístup k persistenci dat.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 3. února 2018

Poděkování

Chtěl bych poděkovat všem, kteří se přímo nebo nepřímo podíleli na vytvoření této práce. Především bych chtěl poděkovat vedoucímu práce, Ing. Jiřímu Daněčkovi, za odorné vedení práce a poskytnutou volnost. Dále bych rád poděkoval svým rodičům za podporu.

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, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 14. května 2018 .....

Odkaz na tuto práci

Hanuš, Jan. *Aplikace funkcionálního programování ve vývoji podnikových aplikací*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Cílem práce je vytvořit přehled funkcionálních principů v kontextu vývoje podnikových aplikací. Teoretická část se zaměřuje na popis obecných funkcionálních principů i jejich dopadů na kvalitu aplikace, věnuje se také problematice měnného stavu a referenční transparentnosti v návaznosti na zjednodušení čitelnosti, udržovatelnosti, škálovatelnosti a testování aplikace. Tato část se rovněž zabývá možnou využitelností matematiky v rámci modularizace a návrhových vzorů, a vytvářením programů na vysoké úrovni abstrakce pomocí vydefinování DSL. V neposlední řadě tato část popisuje, jak jednoduše vytvářet reaktivní aplikace a jak lze funkcionálně přistupovat k persistenci pomocí event-sourcingu a CQRS. Praktická část práce poté demonstruje některé tyto principy na jednoduché client-server aplikaci.

Klíčová slova Podniková aplikace, výuka, funkcionální paradigma, reaktivní programování, CQRS, scala, scalaz, monád, event-sourcing, domain driven design, side-effect, referenční transparentnost.

Abstract

The main goal of the thesis is to summarize functional principles in context of developing enterprise applications. Theoretical part is focused on the description of general functional principles and their impact on quality of an application. Also it is focused on explaining the impact of mutable state and referential transparency on readability, maintainability, scalability and testability of an application. It is clarified how Math is used in a field of modularization, design patterns and in a way programs can be written on high level of abstraction using defined DSL. Last but not least, reactive principles and functional approach of persistence using event-sourcing and CQRS are described. Practical part is focused on demonstrating these princeples on simple client-server application.

Keywords Enterprise application, education, functional and reactive priciples, CQRS, scala, scalaz, monad, event-sourcing, domain driven design, side-effect, referential transparency.

Obsah

Seznam obrázků

Úvod

Softwarové inženýrství je inženýrská disciplína zabývající se praktickými problémy vývoje rozsáhlých softwarových systémů. Ačkoli je to poměrně mladá disciplína, hodně společností po celém světě investuje spoustu peněz do informačních systémů. Velké procento IT projektů však končí neúspěchem. Podle statistik jsou ztráty za neúspěšné implementace informačních systémů ročně mezi 50-150 miliardami amerických dolarů[1].

Podstatnou částí vývoje softwaru je jeho návrh a implementace. Cílem této práce je vytvoření přehledu funkcionálních principů, které tyto dvě části softwarového inženýrství pokrývají, za účelem vylepšení procesu vývoje podnikových aplikací. Za neúspěchem projektů samozřejmě nestojí pouze samotný návrh a implementace, nicméně implementace zajišťuje reálnou podstatu aplikace, čímž se stává klíčovou částí projektu.

Funkcionální programování existuje již mnoho let - předek funkcionálního programování je lambda calculus, který byl vymyšlen dlouho před sestrojením prvního počítače. V dnešní době se funkcionální programování dostává čím dál více do popředí, a to díky několika velkým výhodám (udržovatelnost, možnosti škálování, nižší chybovost, ...). Funkcionální programování je silně propojeno s matematikou. Osvojit si tyto principy a naučit se aplikovat funkcionální vzorce na problémy reálného světa je obtížné a i to může přispívat k nízké rozšířenosti. Z toho vyplývá, že neexistuje tolik zdrojů informací a ukázkových příkladů, jako je tomu v nejpoužívanějším objektovém programování.

Vzhledem k tomu, že se funkcionální programování stává čím dál významnějším pro praktické využití, soustředí se tato práce samozřejmě i na rozšíření povědomí (a propagaci) základních principů funkcionálního přístupu. Výstupem práce je stručný přehled funkcionálních návrhových vzorů, obecných programovacích principů, přístupu k modularizaci, a v neposlední řadě i funkcionální přístup k persistenci dat. V praktické části jsou některé principy demonstrovány na jednoduché podnikové aplikaci.

Kapitola 1

Cíl práce

Cílem práce je ukázat nekonvenční přístup k tvorbě podnikových aplikací za účelem zvýšení kvality a úspěsnosti implementace pomocí funkcionálních principů. Použitím funkcionálního programování je možné:

- snížit chybovost aplikace,
- zlepšit udržovatelnost kódu,
- zlepšit možnost automatizovaného testování,
- zlepšit možnosti horizontálního škálování,
- zlepšit přepoužitelnost kódu.

Dalším cílem je ukázat, jak lze zlepšit kvalitu aplikace z pohledu koncového uživatele, který systém využívá. Použití funkcionálních principů se dobře kombinuje s reaktivním programováním, které má za cíl zlepšit responizivitu, rezilienci a elasticitu. Aplikace je z uživatelského pohledu rychlá a sníží se dopad na uživatele při chybách - programátorských i hardwarových[2].

Cílem práce je vytvořit přehled základních principů funkcionálního programování v kontextu vývoje rozsáhlých aplikací. Takové aplikace jsou vyvíjeny v týmech a je nesmírně důležité, aby bylo možné práci paralelizovat. To vyžaduje, aby byl kód dekomponován po logických celcích a byl čitelný a udržovatelný pro každého člena týmu. Práce předkládá, jak lze tyto vlastosti zlepšit algebraickou modularizací a použitím funkcionálních návrhových vzorů.

Součástí praktické části je vytvoření jednoduché klient-server aplikace. Hlavním účelem práce je ale vytvořit teoretický základ, který je důležitý pro vývoj funkcionálních aplikací. Práce je tedy soustředěna především na teoretickou část. Praktická část pouze ilustruje některé popsané principy.

Kapitola 2

Funkcionální paradigma

Existuje hodně definic funkcionálního paradigmatu, které se snaží jednoduše popsat základní funkcionální principy. Hodně z nich se pouze lehce přiblíží hlavním vlastnostem funkcionálního programování, nebo zabíhají do témat, která jsou úplně irelevantní. V této práci není snaha o jednoduchou definici, ale jsou zde popsány základní principy funkcionálního programování a výhody, které přináší jejich dodržování.

Funkcionální paradigma spadá do kategorie deklarativního programování. To se od imperativního programování liší tím, že kód nepopisuje jednotlivé kroky, které vedou k výsledku, ale popisuje, jaký má být výsledek. Ilustrace rozdílu je ukázána na následujícím jednoduchém příkladu. Program, který v kolekci řetězců každý prvek převede na malá písmena. V imperativním programování kód musí popisovat jak iterovat přes prvky v kolekci a každý prvek převést na malá písmena. Z těchto vytvořených řetězců se v lepším případě vytvoří nová kolekce, nebo v horším se modifikují přímo prvky vstupní kolekce (dále v práci je vysvětleno proč je to horší přístup). Kód napsaný v jazyku Java potom vypadá následovně:

```
List<String> originalList = Arrays.asList("pRVNi", "DRUHy");
List<String> lowerCaseList = new ArrayList<>();
for (int i = 0; i < originalList.size(); i++) {
 lowerCaseList.add(originalList.get(i).toLowerCase());
}
```

Samozřejmě je možné kód napsat jednodušeji pomocí for-each cyklu, ale tohle je klasický imperativní zápis.

Ve funkcionálním programování se vynechá postup a popíše se jen, jak má vypadat výsledek. Stejná funkcionalita napsaná deklarativně v jazyku Scala vypadá následovně:

```
val originalList = List("pRVNi", "DRUHy", "TreTi")
val lowerCaseList = originalList.map(_.toLowerCase)
```

Zde je pouze popsáno, že se nad každým prvkem původního seznamu provede funkce *toLowerCase*. Není zde nic o tom, jak se má iterovat přes kolekci a jak toho dosáhnout, je zde jen deklarativní popis toho, jak má vypadat výsledek. Další zajímavá vlastnost tohoto kódu je, že fragment:

```
originalList.map(_.toLowerCase)
```

je výraz s návratovou hodnotou. *map* je příkladem high-order funkce, což jsou takové funkce, které přijímají funkce jako parametr nebo funkce vrací.

Mezi základní principy funkcionálního programování patří:

- bezstavovost,
- neměnnost dat,
- referenční transparence.

Všechny tyto body spolu úzce souvisí. Funkcionálně psaná aplikace se snaží používat pouze neměnná data. Každá změna je realizována vytvořením nového objektu v požadovaném stavu a původní objekt zůstává nazměněný. Referenční transparentnost funkcí znamená, že výstup je závislý pouze na vstupních parametrech. Pro stejný vstup je vrácen vždy stejný výstup. Funkce ve funkcionálním programování jsou hodně podobné funkcím z matematiky. Díky tomu lze využívat matematické operace s funkcemi, jako je například skládání.

Tyto základní principy zjednodušují minimálně:

- čitelnost kódu,
- údržbu,
- paralelizaci,
- testování.

V objektovém programování je běžné, že metoda přistupuje ke stavu objektu a poté je velmi obtížné pochopit její chování. Je nutné přemýšlet nad všemi stavy, které mohou nastat. Nad pořadím volání metod, protože můžou přistupovat k společnému měnnému stavu a tím se vzájemně ovlivňují. V důsledku narůstá počet všech možných kombinací a je obtížné takové metody testovat. To ve funkcionálním programování odpadá. Funkce je závislá pouze na svých vstupních parametrech. Nemůže nastat změna stavu, která by ovlivnila výstup a proto stačí v testech pokrýt pouze kombinace vstupních parametrů.

Mutabilní stav dále znesnadňuje paralelizaci. Pokud k měnným datům přistupuje více vláken, tak je potřeba použít synchronizační mechanismy, které ještě více znepřehledňují kód, nebo se mohou vyskytnout časově závislé chyby. Naopak k neměnným datům může bez problému přistupovat libovolný počet vláken.

Další důležitý pojem ve funkcionálním programování je **side-effect**. Side-effect funkce je každá operace, která může ovlivnit výsledek, ale nesouvisí se vstupními parametry. Může se jednat o přístup k globální proměnné, přístup k databázi, volání externího systému, čtení souborů a další. Side-effect je i vyhození výjimky. Ve funkcionálním programování je snaha snížit počet side-effectů na minimum, protože zvyšují komplexitu a chybovost systému. Samozřejmě není možné se jim úplně vyhnout, v rozsáhlejší aplikaci se musí přistupovat k databázi nebo volat externí systémy. Je ale vždy možné použít abstrakce, které nám zjednoduší práci se side-effecty tak, že se sníží režie pro správu chyb a zlepší se možnosti kompozice.

Ve funkcionálním programování je snaha tvořit vše pomocí výrazů, které mají návratovou hodnotu a jsou referenčně transparentní. V imperativním programování se běžně používají například podmíněné bloky kódu pomocí *if*, pomocí kterých se aplikace větví do různých stavů. Ve funkcionálním programování podmínky samozřejmě také existují, ale jsou to výrazy s návratovou hodnotou.

2.1 Modularizace funkcionální aplikace

Ve funkcionálním programování jsou data modelována pomocí algebraických datových typů (dále jen ADT). ADT jsou neměnná data bez jakékoliv logiky. Funkce jsou rozčleněny do modulů a ty naopak nedrží žádný stav (data). Funkce pracují pouze se svými vstupnímy parametry.

ADT se dělí na **sum** a **product** typy. Sum typ je například enumerace. Počet všech možných kombinací, které můžou nastat, je počet všech možných hodnot (proto název sum). Product typ naopak obsahuje prvky, které sami mohou nabývat různých hodnot. Počet kombinací je potom součin počtu kombinací všech prvků.

Business logika aplikace je implementována pomocí funkcí. Je dobré dekomponovat aplikaci do menších logických celků. Každý logický celek je v kódu reprezentován modulem, který slučuje funkce, které tuto část aplikace realizují. Při návrhu modulů je vhodné dodržovat obecné principy, jako jsou high cohesion, low coupling a single responsibility principle. Modularizace zlepšuje přehlednost a udržovatelnost kódu. Další velký přínos je možnost paralelizace vývoje. U rozsáhlých aplikací je nesmírně důležité, aby bylo možné vyvíjet v týmu. Vývoj správně navržených modulů, které jsou málo provázané, může probíhat paralelně.

K návrhu funkcionální aplikace se používá tzv. **algebraická modularizace**. Každý modul je definován svou algebrou, která představuje API modulu (podobné interface z OOP). Algebra představuje pouze rozhraní, nejsou zde implementace funkcí ani konkrétní implementace typů - využívají se generické parametry. Od algebry je striktně oddělen interpreter, který implementuje funkce a přidává konkrétní implementace generických typů, se kterými alge-

bra pracuje. Jedna algebra může mít více nezávislých interpreterů, ale všechny musí splňovat axiomy algebry, což při kompilaci ověří testy. Algebra může obsahovat i kompletní implementaci funkcí, které ale vzniknou kompozicí jiných funkcí modulu a implementace tedy jsou platné pro všechny interpretry. Pokud má například algebra funkce *A: X => Y* a *B: Y => Z*, tak bez znalosti implementace funkcí *A, B* i konkrétních typů *X, Y, Z* je možné vytvořit funkci *C: X => Z*, která je implementována jako kompozice funkcí *A* a *B*. Tato implementace je potom platná pro všechny interpretery.

Název algebra vychází z matematiky a opravdu je to obdobné jako v matematice. Algebra se v matematice skládá z množiny, na které je definována, operací (funkcí) a axiomů, které musí operace splňovat. Ve funkcionálním programování je množina dat reprezentována pomocí ADT, operace reprezentují funkce a axiomy lze ověřit pomocí testů (unit nebo property-based testy)[2].

Mezi hlavní výhody algebraického přístupu k modelování patří přepoužitelnost, možnosti kompozice a testování. Díky obecné definici algebry, lze vytvořit dost odlišné interpretery, který pracují s odlišnými ADT. Algebry se dají dále skládat pomocí mixin kompozice a tím se z menších algeber vytváří rozsáhlejší a složitější moduly.

2.2 Domain driven design

Pojem Domain Driven Design (dále DDD) byl zaveden Ericem Evansem v jeho stejnojmené knize[3]. DDD je metoda, která má za cíl zlepšit čitelnost a strukturovanost kódu. Metoda je hojně využívána jak v objektovém, tak ve funkcionálním světě.

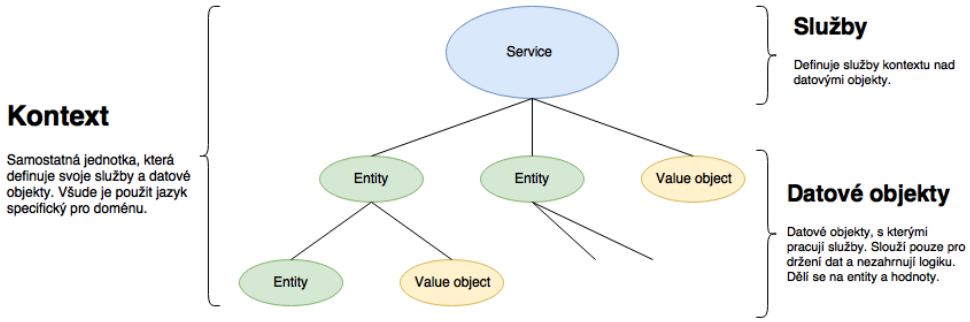
Mezi základní prvky patří tzv. všudypřítomný jazyk. Pojmenování v kódu by mělo odpovídat pojmenování v reálné doméně řešeného problému. To dělá kód přehlednější, čitelnější a usnadňuje to komunikaci mezi programátorem a ostatními.

Dalším prvkem DDD je strukturovanost kódu. Podle DDD se aplikace může dělit na víc logických celků - kontextů (originálně bounded context). Každý kontext může obsahovat stejně pojmenované objekty, ale v každém mají odlišný význam. Každý kontext se dále dělí na služby a datové objekty, graficky je to znázorněno na obrázku 2.1.

Datové objekty drží pouze data a neobsahují žádnou logiku. Ty se dále dělí na:

- hodnoty (value objects),
- entity.

Hodnotové objetky modelují taková data, která jsou určena pouze svou hodnotou a nepotřebují identifikátor. Typický příklad je adresa. Na konkrétní adrese může bydlet více osob, ale pořád se jedná o stejnou adresu - stejný



Obrázek 2.1: Grafické znázornění kontextu.

objekt. Při porovnávání takových hodnot jsou porovnávány pouze atributy objektu. V některém kontextu se samozřejmě může hodit i adresu obohatit o identifikátor. Hodnotové objekty jsou neměnné a můžou být volně sdílené napříč entitami.

Entity jsou naopak data, která jsou určena identifikátorem. Příkladem může být člověk. Pokud se dva lidé jmenují stejně (mají stejné atributy), tak se určitě nejedná o stejnou osobu. Je potřeba tedy člověka obohatit o jednoznačný identifikátor, který jasně určí konkrétní osobu. Entity mají většinou definovaný životní cyklus a jsou měnné. Můžou v sobě zahrnovat další entity a hodnotové objekty. Komplexnější entity tvoří stromovou strukturu a kořenu tohoto stromu se říká agregát.

Služby jsou obecně bezstavové a slouží k manipulaci s daty. Služby se dělí na:

- repozitáře,
- fabriky,
- služby.

Repozitář přistupuje k databázi a stará se o persistenci dat.

Pomocí fabrik jsou vytvářeny datové objekty. V DDD by jediné místo, které volá konstruktor libovolného objektu, měla být právě fabrika. To je užitečné například pro validace vstupů. Konstruktor vždy vrací instanci dané třídy. Pokud konstruktor dostane nevalidní data, tak jediná možnost signalizace chyby je vyhození výjimky. Ve fabrice lze návratovou hodnotu zabalit do vhodné abstrakce, která přímo signalizuje možnost pádu a validace se provedou před voláním konstruktoru.

Služby představují vystavěné rozhraní daného kontextu. Slouží k manipulacím s daty a jsou bezstavová. Kontext komunikuje s okolním světem pouze prostřednictvím služeb.

Použití metod DDD ve funkcionální aplikaci přináší návod, jak lze strukturovat kód a použití všudypřítomného jazyka zlepšuje čitelnost kódu. Mapování

funkcionálních prvků a DDD je intuitivní - moduly reprezentují služby, ADT reprezentují datové objekty.

2.3 Scala

V této práci se jako implementační jazyk využívá Scala. Scala je moderní funkcionálně-objektový jazyk, který se překládá do bytecodu a běží v JVM. Scala má velkou podporu pro funkcionální konstrukty a existuje mnoho kvalitních knihoven a frameworků pro vývoj podnikových aplikací. V této kapitole jsou popsány zajímavé konstrukty, které usnadňují funkcionálním programování.

Mezi základní stavební kameny funkcionálního programování patří ADT. Ty je možné ve Scale jednoduše realizovat pomocí *case class*. To jsou třídy, které mají všechny své atributy neměnné (pokud se explicitně u atributů neuvede, že mají být modifikovatelné). Další výhoda *case class* je, že se jim automaticky vygenerují metody *equals* a *hashcode* a tzv. **Companion object**, který slouží jako **factory** pro vytváření nové instance. Díky companion object lze *case class* využívat v **pattern matching**.

Moduly lze jednoduše realizovat pomocí *trait*, což je obdobné jako *interface* v jazyku Java. Trait ve Scale umožňuje mixin kompozici, díky které lze vytvářet složitější moduly z více jednodušších.

Další velkou výhodou jazyku Scala je její silný typový systém s velkou podporou generik. Statické typování umožňuje vysokou kontrolu již při kompilaci. Silný systém generik přináší vysokou míru přepoužitelnosti a robustnosti kódu.

Scala má širokou škálu funkcionálních prvků. Má podporu lambda výrazů (anonymních funkcí), high-order funkcí a expression-oriented programming. Ve Scale je téměř vše výraz, včetně podmíněného bloku *if*, který je také výraz s návratovou hodnotu.

V části o návrhových vzorech je ukázáno, jak lze funkcionálně přistupovat k side-effect. Na to se většinou využívá funkcionální návrhový vzor **Monád**, do kterého se zabalí výsledek funkce. Kompozice Monádů se realizuje pomocí funkcí *map* a *flatMap*. Kompozice většího množství Monádů se velmi rychle stává nečitelná, protože dochází k zanořenému volání těchto funkcí. To ve Scale řeší velmi mocný nástroj - **for-comprehension**, což je syntaktická zkratka, která ze zanořeného volání *map* a *flatMap* dělá přehledný a čitelný kód. Pro posloupnost volání *flatMap*:

```
val account = create("123456789")
 .flatMap(a => {
  credit(a, 1000).flatMap(b => {
   debit(b, 100).flatMap(close)
  })
 })
```

je ekvivalentní zápis pomocí **for-comprehension**:

```
val account1 = for {
 a <- create("123456789")
 b <- credit(a, 1000)
 c <- debit(a, 100)
 d <- close(c)
} yield d
```

Druhý zápis je určitě přehlednější. V ukázkách není důležitá implementace funkcí, ale to, že každá vrací implementaci **Monádu**. For-comprehension je opět výraz s návratovou hodnotou.

Další velmi užitečná vlastnost Scaly je, že funkce může mít více sad parametrů. Použití je vystvětleno na následující ukázce:

```
def sum1(a: Int, b: Int) = a + b
def sum2(a: Int)(b: Int) = a + b
```

Funce *sum1* a *sum2* jsou ekvivalentní. Obě pouze sečtou své parametry. Pokud se volá funkce, tak se sady parametrů doplňují zleva. Každá sada parametrů musí mít určeny všechny své parametry (tedy pokud nemají defaultní hodnoty), ale není potřeba plnit všechny sady parametrů. Funkci *sum2* je možné zavolat pouze s první sadou parametrů:

```
val fun = sum2(1)
```

Návratový typ takového volání je zase funkce, která má jako vstup zbytek sad parametrů (v tomto případě jedna sada s číselným parametrem) a návratovou hodnotu původní funkce. To má praktické využití. Je obvyklé, že modul definuje množství funkcí, které vykonají nějakou logiku a poté přisupují do repozitáře. Pokud je repozitář vyčleněn v samostatné sadě parametrů, tak je možné provést více funkcí, vytvořit jejich kompozici a až poté přidat parametr s repozitářem.

2.4 Shrnutí

V této kapitole jsou velmi okrajově popsány základní koncepty, které jsou v této práci dále rozvíjeny a použity v praktické části. Jsou zde popsány některé čistě funkcionální prvky, ale i obecné principy vývoje aplikací.

DDD je obecný princip vývoje aplikací, který má za cíl pomocí zásad a návrhových vzorů zvýšit čitelnost a udržovatelnost kódu. Aplikace je členěna do kontextů, které se skládají z datových objektů a služeb provádějících business logiku nad datovými objekty. Pokud se spojí principy DDD a funkcionálního programování, tak datové objekty jsou realizovány jako ADT a služby jako moduly. Fabriky jsou v jazyku Scala při použití **case class** zadarmo, protože se u nich automaticky generuje **companion object**, který slouží jako fabrika.

Je samozřejmě možné automaticky generovaný **companion object** přepsat a doplnit vytváření instancí o validace. Dodržováním dalších konceptů DDD, jako je odčlenění repozitáře a používání všudypřítomného jazyka domény, se zlepšuje udržovatelnost a čitelnost kódu.

Jako další je v kapitole ukázáno, proč je jazyk Scala dobrý kandidát pro realizaci funkcionálních aplikací. Má vysokou podporu pro funkcionální konstrukty a umožňuje psát stručný čitelný kód.

Kapitola 3

Funkcionální návrhové vzory

Návrhový vzor je přepoužitelné řešení obvyklého problému v daném kontextu[4]. Při vývoji aplikací se často opakují podobné situace a návrhové vzory slouží jako předpis řešení takové situace. Aplikace návrhového vzoru je tedy přepoužití již vzniklé šablony řešení v určitém kontextu.

V objektovém programování, v drtivé většině případů, existují návrhové vzory pouze jako popis, jak daný problém řešit. Je nutné je implementovat znovu pro každý kontext. Tím se liší od funkcionálních návrhových vzorů. Jejich implementace, které řeší problém, jsou generické a přepoužitelné. Do hotových řešení se pouze vloží konkrétní kontext.

Používání návrhových vzorů je přínosné tím, že:

- poskytuje řešení problému,
- formuluje slovník.

Druhý bod je často opomíjen. Existuje problém a návrhový vzor ho řeší - použije se. Druhý bod je ale z pohledu údržby neméně důležitý. Používání vzorů zlepšuje samopopisnost a čitelnost kódu. Pokud se dodržuje slovník a struktura řešení, který vzor přináší, tak každý, kdo použitý vzor zná, snadno pochopí, co se v kódu děje. Je tedy velmi důležité využívat slovník a předepsané struktury vzoru (pokud existují).

V této kapitole jsou popsány základní funkcionální návrhové vzory s jednoduchými ukázkami jejich použití. Je důležité zmínit, že jsou více abstraktní a mají blíže k matematice, než vzory z objektového programování. Většina vychází z matematické teorie kategorií. Axiomy matematických struktur poskytují návod, jak lze testovat návrhové vzory. Teorie kategorií se zabývá matematickými strukturami a zkoumá jejich vzájemné vztahy a morfismy.

3.1 Lens

Ve funkcionálním programování se pro držení dat využívají ADT, které jsou imutabilní. Každá změna tedy znamená vytvoření nové instance s novými daty. V triviálním případě je to jednoduché a čitelné, ale pokud se mění hluboce zanořený atribut agregátu, tak je nutné vytvořit nové instance pro celou cestu stromem od kořene až k změněnému atributu. To je při velké hloubce zanoření nepřehledné. To řeší návrhový vzor **Lens**.

Použití vzoru je ilustrováno na příkladu, který pracuje s následujícími ADT:

```
case class Address(city: String, street: String, houseNo: Int)
case class Person(id: String, name: String, address: Address)
```

Pokud je potřeba změnit číslo popisné adresy u agregátu *Person*, tak kód vypadá:

```
val a1 = Address("Praha", "Dlouha", 3)
val p1 = Person("1", "Petr Marek", Add)

val a2 = a1.copy(houseNo = 4)
val p2 = p1.copy(
 address = p1.address.copy(
  houseNo = 4
 )
)
```

Zde je pouze jedna úroveň zanoření a již to začíná být komplikované a zdlouhavé. S každou úrovní zanoření komplexita a délka zápisu roste.

Lens slouží k funkcionálnímu přístupu k atributu objektu. Tento atribut je možné pomocí Lens číst nebo modifikovat. Při modifikaci je vrácena nová instance ADT s novou hodnotou atributu a původní instance zůstane nezměněná.

Funkcionální návrhové vzory jsou generické a je možné je přepoužívat. Výhoda obecných návrhových vzorů je, že je možné vytvářet obecné kombinátory, které umožnují užitečné operace (např. skládání). Obecná algebra návrhového vzoru Lens může vypadat takto:

```
case class Lens[O, V](
 get: O => V,
 set: (O, V) =>
O~)
```

Parametr O představuje typ ADT a V typ atributu, ke kterému se pomocí Lens přistupuje.

Vzor má dva atributy:

get je funkce z typu objektu do typu atributu, ke kterému se váže. Funkce pouze vrací hodnotu tohoto atributu pro danou instanci.

set je funkce, která jako parametr bere instanci objektu a novou hodnotu atributu, pomocí které vytvoří novou instanci a vrátí ji.

Vytvoření konkrétního Lens a jeho použití je následující:

```
val addressCityLens: Lens[Address, String] = Lens[Address, String](
 get = _.city,
 set = (o, v) => o.copy(city = v)
)

val a1 = Address("Prah", "Radlicka", 5)
val a2 = addressCityLens.set(a1, "Praha")
```

Takto nemá Lens žádnou výhodu, stále je potřeba nastavovat nový atribut pro celou cestu stromem. Zde je vhodné vytvořit kombinátor, který umí Lens skládat a tedy vytvořit složený Lens, který již umí modifikovat zanořené atributy. Implementace funkce *compose* vypadá následovně:

```
def compose[Outer, Inner, Value](
 outer: Lens[Outer, Inner],
 inner: Lens[Inner, Value]
) = Lens[Outer, Value](
 get = outer.get andThen inner.get,
 set = (obj, value) => outer.set(obj, inner.set(outer.get(obj), value))
)
```

Pomocí této funkce je možné vytvářet složený Lens, který již umí přímo modifikovat zanořený atribut:

```
val personAddressLens = Lens[Person, Address](
 get = _.address,
 set = (o, v) => o.copy(address = v)
)
val personCityLens = compose(personAddressLens, addressCityLens)
val person = personCityLens.set(person, "Praha")
```

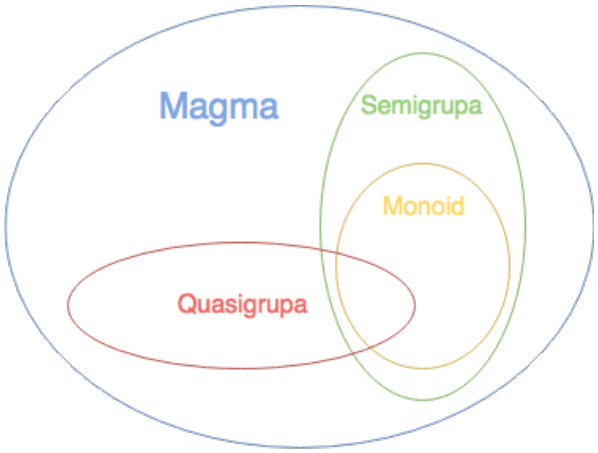
Jelikož má návrhový vzor zcela generický předpis nezávislý na konkrétním případu užití, existuje spousta již hotových implementací v knihovnách. Většina knihoven navíc obsahuje kombinátory, které s Lens pracují. **Scalaz** je hojně využívaná knihovna, která zahrnuje mnoho funkcionálních návrhových vzorů včetně Lens.

Některé knihovny navíc využívají makra a vytvoření Lens je potom jeden řádek kódu. Mezi tyto knihovny patří **Shapeless**. Vytvoření Lens pro zanořený atribut pomocí této knihovny vypadá následovně:

```
val personHouseNoLens = lens[Person].address.houseNo
```

3.2 Monoid a semigrupa

V teorii kategorií je **monoid** a **semigrupa** společně s **quasiqrupami** podmnožina obecnější struktury, která se nazývá **magma**. Množinové znázornění je zobrazeno na obrázku 3.1. Tyto struktury obecně zkoumají uzavřené binární operace nad daty stejného typu, tedy operace typu *(A, A) => A*. Jak je patrné z 3.1, magma je nejobecnější struktura, která má pouze axiom uzavřenosti operace. V této práci je popsán pouze návrhový vzor **monoid** a **semigrupa**, protože jsou nejvíce využívány[5].



Obrázek 3.1: Vztah mezi strukturami magma, quasigrupa, semigrupa a monoid.

3.2.1 Monoid

Jako první je popsán návrhový vzor **monoid**. Monoid se skládá z množiny dat, na které je definovaný a binární operace, která je na dané množině uzavřena. Aby daná struktura byla monoid, musí navíc splňovat následující axiomy[6]:

- asociativita,
- existence neutrálního prvku.

Jednoduchý příklad monoidu je klasické sčítání nebo násobení reálných čísel. Obě operace jsou asociativní, uzavřené a existuje neutrální prvek. Axiomy jsou intuitivní a každý nějakou formu monoidu používá na denní bázi, aniž by si to uvědomoval.

V programování je monoid abstrakce, která má jeden generický parametr, určující množinu dat, na které je monoid definovaný. Dále definuje funkci, která splňuje axiomy (axiomy je vhodné ověřit pomocí testů) a nulový prvek, vůči kterému je funkce neutrální. Obecný a přepoužitelný předpis vzoru může ve Scale vypadat takto:

```
trait Monoid[T] {
 def zero: T
 def op(t1: T, t2: T): T
}
```

Zákony, které musí každá implementace monoidu splňovat jsou tedy následující:

- pravá identita – op(zero, t) == t
- levá identita – op(zero, t) == t
- asociativita – op(t1, op(t2, t3)) == op(op(t1, t2), t3)

Jednoduchý příklad monoidu může být například spojování řetězců.

```
val stringMonoid = new Monoid[String] {
 override def zero: String = ""

 override def op(t1: String, t2: String): String = t1 ++ t2
}
```

Potom pro všechny řetězce *s1, s2 a s3* musí platit:

```
assert(stringMonoid.op(s1, stringMonoid.zero) == s1)
assert(stringMonoid.op(stringMonoid.zero, s1) == s1)
assert(stringMonoid.op(s1, stringMonoid.op(s2, s3))
 == stringMonoid.op(stringMonoid.op(s1, s2), s3))
```

3.2.2 Semigrupa

Semigrupa je nadmnožinou **monoidu** (každá semigrupa je zaroveň monoid, ale každý monoid není semigrupa). Jako každá podmnožina magmy má semigrupa množinu, na které je definována a binární operaci, která je na množině uzavřená. Platí zde i asociativní zákon, ale narozdíl od monoidu nemusí existovat neutrální prvek vůči operaci[7].

Přepoužitelná algebra vzoru může vypadat následovně:

```
trait Semigroup[T] {
 def op(t1: T, t2: T): T
}
```

Příklad jednoduchého použití semigrupy může být hledání většího čísla. Operace nalezení většího ze dvou čísel je určitě asociativní, ale zde již nedává smysl neutrální prvek.

```
val maxIntSemigroup = new Semigroup[Int] {
 override def op(t1: Int, t2: Int): Int = math.max(t1, t2)
}
```

Zde musí platit, že pro každé *i1, i2 a i3*:

```
assert(maxIntSemigroup.op(i1, maxIntSemigroup.op(i2, i3))
 == maxIntSemigroup.op(maxIntSemigroup.op(i1, i2), i3))
```

3.2.3 Shrnutí

V této části jsou popsány návrhové vzory **monoid** a **semigrupa**, které se zabývají binárními operacemi uzavřenými vůči množině, na které jsou definovány. V semigrupě musí operace splňovat asociativní zákon a monoid je její podmnožina, která má navíc neutrální prvek vůči operaci.

Na jednoduchých příkladech je ukázáno použití vzorů. Možná přijde méně intuitivní proč vůbec tyto vzory používat, když abstrahují tak primitivní operace. K tomu jsou při nejmenším čtyři důvody:

Čitelnost Abstrahovat se může diametrálně komplexnější operace. Zabalení do vzoru znalému člověku hned na první pohled řekne, co od operace očekávat a jaké splňuje zákony. To vše bez použití komentáře a hlubšího zkoumání komplikovaného kódu.

Kompozice Zabalení problému do obecné abstrakce umožňuje daleko lepší kompozici. Je možné vytvářet obecné kombinátory, které pouze očekávají splnění axiomů dané abstrakce.

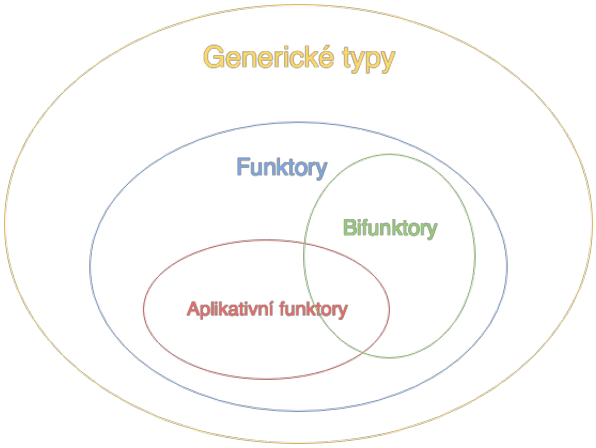
Testování Axiomy algebry dávají přímo návod jak testovat daný problém.

Jednotnost Používáním přepoužitelných abstrakcí skrz celou aplikaci vytváří jednotný koncept pro psaní kódu a tím se aplikace stává přehlednější.

Algebra monoidu a semigrupy je opět generická a přepoužitelná, takže není potřeba vytvářet si svůj vlastní předpis. Implementace existuje například v knihovně **Scalaz**.

3.3 Funktor

Funktor je podmnožina **generického typu**. Všechny podmnožiny generického typy se zabývají abstrakcemi (kontejnery) nad daty, a jejich morfismy. Množinově je závislost struktur znázorněna na obrázku 3.2.



Obrázek 3.2: Vztah mezi strukturami generické typy, funktory, aplikativní funktory a bifunktory.

Funktor slouží k mapování prvků v generické struktuře na jiné prvky (i odlišného typu). Příkladem je *List*, ten přijímá generický parametr a implementuje metodu *map*, která vrací novou instanci *List*.

Funktor je velmi obecná struktura. Její hlavní funkcí je transformování libovolné abstrakce nad typem do té samé abstrakce, která již může obsahovat odlišný typ, tedy morfismy typu *F[A] =>F[B]*, kde *A* a *B* můžou být různé. Vysvětlení je ilustrováno na příkladu:

```
trait Functor[F[_]] {
 def identity[A](a: A): A = a
 def map[A, B](a: F[A])(f: A => B): F[B]
}
```

Funktor přijímá jeden generický parametr vyššího řádu (také je generický). Může se například jednat o *List*, *Try* nebo *Future* – všechny berou jeden generický parametr a obalují ho.

Funktor definuje dvě funkce:

identity jedná se o identické zobrazení.

map přijímá jako parametry abstrakci *F* nad typem *A* a funkci *A => B*. Návratový typ je *F[B]*. Tedy prvky z prvního argumentu se transformují aplikací funkce v druhém argumentu a znovu se zabalí do stejné abstrakce.

Použití je ilustrováno na následující ukázce, kde je vytvořen jednoduchý funktor nad *List*.

```
val functor = new Functor[List] {
```

```
 override def map[A, B](a: List[A])(f: A => B): List[B] = a map f
}
```

Mezi axiomy funktoru patří:

identita funkce *identity* je neutrální vůči *map*.

distributivita funkce *map* je distributivní, tzn. řetězec volání map je ekvivalentní jednomu volání nad kompozicí funkcí.

Na předchozím příkladu funktoru nad abstrakcí *List* je splnění axiomů demonstrovánon následujícím kódem:

```
val s = List(" one ", "TwO ", "thrEE")

val trim: String => String = _.trim
val lower: String => String = _.toLowerCase

assert(functor.map(s)(functor.identity) == s)

assert(
 functor.map(functor.map(s)(trim))(lower)
  == functor.map(s)(trim andThen lower)
)
```

Zde první *assert* ověřuje axiom identity a druhý *assert* distributivní zákon.

Na functor je možné nahlížet ze dvou perspektiv. Zde byl popsán funktor jako nedstavba nad libovolným typem, který má generický parametr a jsou nad ním implementovány operace *map* a *identity*. Dost často se používá i přístup, že sám typ o sobě je funktorem, pokud tyto operace implementuje. Například *List* je funktor, protože implementuje funkci *map* a funkce *get* splňuje axiomy identity.

3.4 Monád

Monád je asi nejpoužívanější a nejznámější funkcionální návrhový vzor. Monád je abstrakce nad objektem (obalová třída), která usnadňuje kompozici a abstrahuje nějaký efekt. Díky monádům je možné úplně zbavit kód try-catch bloků.

Mezi nejpoužívanější implementace monádu patří:

Option abstrahuje nepovinost instance.

Future abstrahuje neblokující vyhodnocení.

Try abstrahuje možnost výskytu chyby.

List abstrahuje mnohonásobnost objektů.

Všechny tyto abstrakce jsou ve Scale velmi používané a ke všem je možné přistupovat při kompozici stejně.

Scala bohužel nemá ve standardní knihovně předpis pro monád, takže tyto abstrakce pouze splňují předpoklady pro to, aby byly monádem, ale nikde to neni explicitně řečeno. To ale neznamená, že se nejedná o monád [8].

Obecná algebra pro vzor monád vypadá takto:

```
trait M[A] {
 def flatMap[B](f: A => M[B]): M[B]
}
def unit[A](x: A): M[A]
```

Zde je schválně vytažená funkce *unit* ven z *trait*. To proto, že má podobnou funkci jako konstruktor. Má jeden argument a z něj vytvoří instanci monádu. Proto nemůže být uvnitř traitu (nedává smysl volání konstruktoru nad již vytvořeným objektem). Obvykle se vůbec neimplementuje, protože ve Scale lze využít funkci *apply* vygenerovaného companion objektu case třídy [8].

Funkce *flatMap* má podobnou signaturu jako funkce *map* u funktoru, ale je silnější. Jako argument bere funkci *A => M[B]*. Ta se vykoná nad každým prvkem monádu. Zde je rozdíl oproti funktoru, ten bere jako argument funkci *A => B*. To znamená, že pokud by byla implementace obdobná jako u funktoru, tak by návratová hodnota měla být *M[M[B]]*, ale není. To proto, že na konci funkce *flatMap* se provede zpološtění. Z vnitřních monádů se vytáhnou abstrahované instance a ty se vloží do vnějšího monádu. To možná vypadá komplikovaně, ale dává to větší sílu při kompozici. Vše, co lze udělat pomocí *map*, je možné i pomocí *flatMap*, ale naopak to neplatí.

Jako každá algebra i monád musí pro správnou funkčnost splňovat své axiomy. Pokud x je instance libovolného objektu, m konkrétní instance libovolného monádu a funkce f a g typu *A => M[A]*, tak musí platit:

levá identita *unit(x).flatMap(f) == f(x)*

pravá identita *m.flatMap(unit) == m*

asociativita *m.flatMap(f).flatMap(g) == m.flatMap(y => f(y).flatMap(g))*

Kompozice více monádů, znamená řadu zanořených volání *flatMap*. Ve Scala je možné pro zpřehlednění využít for-comprehension.

Pro znázornění síly monádů je použití ukázáno na následujícím příkladu z reálného světa. V internetovém bankovnictví se uchovávají uživatelské účty v databázi a je možné převádět peníze z jednoho účtu na druhý.

```
case class Account(no: String, balance: Double)

trait Database {
 def query(no: String): Try[Account]
}
```

```
trait AccountService {
 def transfer(from: Account, to: Account,
        amount: Double): Try[(Account, Account)]
}

class Transfer {
 def main(args: Array[String]): Unit = {
  for {
   from <- Database.query("12345678")
   to <- Database.query("87654321")
   _ <- AccountService.transfer(from, to, 1000.0)
  } yield _
 }
}
```

Ukázka je velmi zjednodušena, reálně by ADT *Account* mělo daleko více parametrů, ale to pro znázornění použití monádů není důležité [2].

V ukázce je zjednodušená algebra pro databázi (interpreter není důležitý), která má funkci reprezentující dotaz do databáze, který hledá účet s daným číslem účtu. To samozřejmě nemusí účet najít, nebo může být databáze nedostupná, proto je návratový typ monád *Try*.

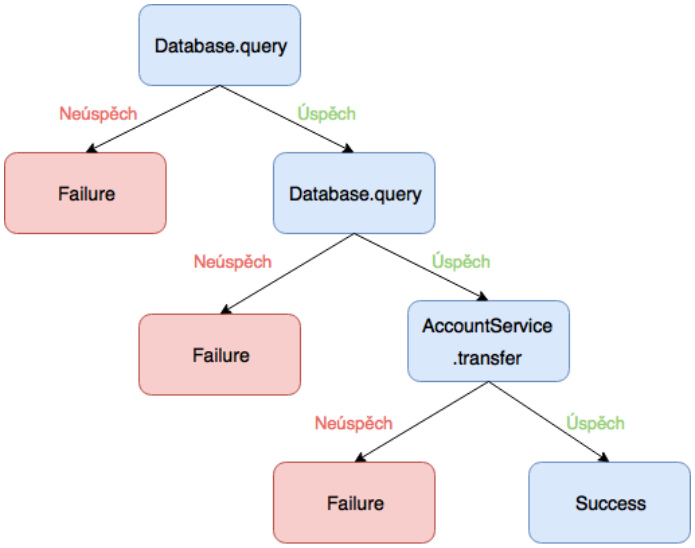
Dále je v příkladu algebra modulu *AccountService*, která implementuje metodu pro převod mezi účty. Volání nemusí skončit úspěšně (zdrojový účet např. nemusí mít dost velký zůstatek), proto je návratový typ také *Try*.

Ve funkci *main* je znázorněna kompozice volání. Nejdříve se provedou dotazy pro zdrojový a cílový účet a následně se provede převod. Kód je samopopisný a dokonce pokrývá i stavy, kdy se nepovede najít některý účet, anebo z nějakého důvodu neprojde převod. To vše bez jediné podmínky a try-catch bloku, pouze pomocí monádů a funkce *flatMap*.

Návratový typ for-comprehension je znovu *Try*. Ten abstrahuje možnost chyby. Dělá to tak, že má dva potomky. Jednoho pro úspěch – *Success* – a jednoho pro neúspěch – *Failure*. Řada volání *flatMap* poté skončí na první chybě a vrátí chybnou instanci *Try*, nebo nedojde k chybě a je vrácena větev *Success*. Vyhodnocení kompozice z příkladu je znázorněno na obrázku 3.3.

3.5 Reader Monád

Reader monád slouží pro vkládání globální hodnoty do funkce. Tento návrhový vzor je možné přirovnat k **dependency injection**, jak je znám z objektových jazyků. Klasické použití dependency injection v Javě je takové, že se dynamicky vkládají závislosti, které se drží jako třídní proměnné. To znamená, že se vytváří stav objektu, na kterém je závislé volání funkcí využívajících tuto třídní proměnnou.



Obrázek 3.3: Monadické vyhodnocení kompozice.

Reader monád zde již není vysvětlen tak matematicky jako to bylo u předchozích vzorů, ale místo toho je zde vysvětlení na ukázkovém příkladu s postupným vývojem algebry od naivní implementace až po použití Reader monádu.

Příkladem běžné globální hodnoty je databáze. Je běžné, že mnoho funkcí v modulu přístupuje k persistentním datům. Takovou situaci je možné řešit tak, že se repozitář přidá jako závislost třídy, ale tím se poruší bezstavovost modulu. Další možnost je přidání repozitáře do argumentu funkce. To má nevýhodu, že je potřeba přidávat databázi každému volání funkce. Toto řešení je znázorněno na následující ukázce:

```
case class Account(no: String, balance: Double)

trait Repository {
 def query(no: String): Try[Account]
}

trait AccountService {
 def open(no: String, repository: Repository): Try[Account]
 def credit(no: String, amount: Double, repository: Repository): Try[Account]
 def debit(no: String, amount: Double, repository: Repository): Try[Account]
 def balance(no: String, repository: Repository): Try[Double]
}

class Test extends App {
 override def main(args: Array[String]): Unit = {
  val no = "12345667"
```

```
  val acc = for {
   account <- AccountService.open(no, Repository)
   _ <- AccountService.credit(account, 500.0, Repository)
   _ <- AccountService.credit(account, 500.0, Repository)
   _ <- AccountService.debit(account, 500.0, Repository)
   b <- AccountService.balance(no, Repository)
  } yield b
 }
}
```

Je zřejmé, že se repozitář musí vkládat do každého volání. To má tu nevýhodu, že kompozice je možná pouze na úrovni Try monádu. Reader monád abstrahuje funkci, která přistupuje k této globální hodnotě. Předpis tohoto návrhového vzoru je náledující:

```
case class Reader[R, A](run: R => A)
```

Reader má dvě generiky. První reprezentuje globální proměnnou, ke které je potřebný přístup. Druhý reprezentuje návratový typ abstrahovaného volání. Takto definovaný vzor by nebylo možné skládat a není to monád. Přidáním metody *flatMap* a případně *map* se stává monádem a tím se zlepší možnosti kompozice.

```
case class Reader[R, A](run: R => A) {
 def map[B](f: A => B): Reader[R, B] =
  Reader(r => f(run(r)))
 def flatMap[B](f: A => Reader[R, B]): Reader[R, B] =
       Reader(r => f(run(r)).run(r))
}
```

Nyní je možné využít reader monád ve for-comprehension a vytvářet kompozice.

Další krok je změna signatury funkcí v *AccountService* s využitím reader monádu. Repozitář se dále nepředává jako argument funkce, ale návratový typ je reader monád, který je parametrizovaný repozitářem a abstrahuje tak přístup funkce k repozitáři.

```
trait AccountService {
 def open(no: String): Reader[Repository, Try[Account]]
 def credit(no: String, amount: Double): Reader[Repository, Try[Account]]
 def debit(no: String, amount: Double): Reader[Repository, Try[Account]]
 def balance(no: String): Reader[Repository, Try[Double]]
}

class Test extends App {
 override def main(args: Array[String]): Unit = {
  val no = "12345667"
  val acc = for {
```

```
   _ <- AccountService.open(no)
   _ <- AccountService.credit(account, 500.0)
   _ <- AccountService.credit(account, 500.0)
   _ <- AccountService.debit(account, 500.0)
   b <- AccountService.balance(no)
  } yield b
  acc.run(AccountRepository)
 }
}
```

Na ukázce je vidět, že for-comprehension nyní neprovede celý výpočet, ale vytvoří pouze kompozici funkcí a vrací opět reader monád. Teprve při volání funkce *run* se vloží závislost na konkrétní repozitář a provedou se všechny komponované kroky. Díky tomu je jednoduché například vkládat odlišnou implementaci repozitáře pro testování. Bylo dosaženo referenčně transparentní **dependency injection** bez použití složitého frameworku.

3.6 State Monád

Funkcionální programování má za cíl minimalizovat funkce, jejichž výsledek je závislý na měnném stavu. Při programování je ale někdy nutné pracovat se stavem. Příkladem je generování náhodného (pseudo-náhodného) unikátního identifikátoru. Generuje se nová hodnota dokud není unikátní – hodnota tedy v čase mění svůj stav. I to je možné vyřešit pomocí neměnných struktur například pomocí tail rekurze, ale to by znemenalo, že pro každé abstrahování stavu je nutné vytvořit unikátní řešení, z kterého hned není patrné, že abstrahuje stav. K jednotnému přístupu pro ošetření měnného stavu slouží návrhový vzor **state monád**.

Podrobné vysvětlení implementace state monádu pro tuto práci není důležité, je to jen nějaká nadstavba nad klasickým vzorem monád. Použití je ilustrováno na implementaci z knihovny scalaz, kde se nachází více variací state monádu a mnoho kombinátorů.

V knihovně scalaz je definován trait *State* s následující signaturou *trait State[S, +A]*, kde S je pouze typ stavu, který se uvnitř mění a A přidává možnost vrácení meta informace ze změny stavu. Monád je možné vytvořit pomocí kombinátorů, které skládají výslednou funkci změny stavu, je tedy zase možné vytvářet složitější kompozice z menších funkcí. Příkladem takového kombinátoru je funkce *modify*, jejíž implementace vypadá následovně:

```
def modify[S](f: S => S): State[S, Unit] = State(s => {
 val r = f(s);
 (r, ())
})
```

Tato funkce přijímá jako argument funkci bez side-effect, která provádí trans-

formaci původního stavu. Návratový typ je state monád, který tuto transformaci abstrahuje. Vytvořený monád zase pouze abstrahuje výpočet, monád ještě nemá žádnou informaci o počátečním stavu. Díky tomu je možné vytvářet složitější kompozice před samotným vyhodnocením. Pro spuštění výpočtu se musí zavolat funkce *run*, které se jako argument teprve předá počáteční stav.

```
import scalaz.State
import State._

object StateMonad extends App {
 def inc(num: Int): State[List[Int], Unit] = {
  modify { (elems: List[Int]) => {
   elems.map(_ + num)
  }}
 }

 override def main(args: Array[String]): Unit = {
  val nums = List(1, 2, 3, 4)
  inc(1) run nums
 }
}
```

Zde je ukázán spustitelný fragment kódu. Funkce *inc* vytváří state monád, který inkrementuje všechny prvky číselného seznamu o hodnotu předanou v parametru. Ve funkci *main* je ukázáno konkrétní použití.

Jako motivační příklad, kdy je vhodné použití měnného stavu bylo zmíněno generování unikátního identifikátoru. Pro řešení takového problému, kdy je nutné modifikovat stav dokud se nesplní nějaká podmínka, je vhodné použití funkce *whileM_*. Ta má dva argumenty, první vyhodnocuje, jestli je stav validní a druhý provádí transformaci. Transformace se provádí cyklicky dokud stav není vyhodnocen jako validní.

3.7 Free Monád

Free monád je návrhový vzor, který je spíše strukturální a patří mezi více pokročilé. Jeho hlavní účel je oddělení interpreteru od algebry programu. Pomocí algebry je poté možné monadickou kompozicí vytvořit deklarativně celý program bez znalosti interpreteru (tedy implementace). Až při vyhodnocení se vloží konkrétní interpreter, který tento program spustí. Tento přístup je podobný tomu, jak fungují kompilátory. Nejdříve se vytvoří **abstraktní syntaktický strom** (dále AST), který je reprezentovaný čistě daty. Až poté se vloží interpretery, které s AST pracují[2].

Opět zde není popsána vlastní implementace, místo toho je pro vysvětlení použita již existující implementace z knihovny scalaz. Zjednodušená signatura

vzoru je *Free[S, A]*, kde:

Free reprezentuje program. Tedy fragment AST, který je možný skládat s jinými fragmenty.

S reprezentuje jazyk, který je nutné vydefinovat. Zahrnuje elementární operace reprezentované pomocí ADT.

A reprezentuje návratový typ programu.

Free monád pracuje na opravdu vysoké úrovni abstrakce. Při kompozici se vlastně provádí kompozice programů, u kterých známe pouze jazyk a návratový typ.

Použití je zde vysvětleno na příkladu. Pomocí free monádu je vytvořena implementace zjednodušeného repozitáře pro správu knih v knihovnickém informačním systému. Tento repozitář podporuje operace vyhledávání, ukládání a mazání. Vše pracuje pouze s entitou knihy, která je implementována pomocí ADT:

```
case class Book(no: String, name: String, author: String)
```

Dále je potřeba vydefinovat jazyk, tedy pomocí ADT vydefinovat operace, které repozitář poskytuje:

```
sealed trait BookRepositoryF[+A]
case class Query(no: String) extends BookRepositoryF[Book]
case class Store(book: Book) extends BookRepositoryF[Book]
case class Delete(book: Book) extends BookRepositoryF[Unit]
```

Zde *BookRepositoryF[+A]* vyjadřuje obecnou operaci nad repozitářem s návratovou hodnotou A a tvoří jazyk programů.

```
object free {
 type BookRepo[A] = Free[BookRepositoryF, A]
}

import free._
trait BookRepository {
 def query(no: String): BookRepo[Book] = {
  Free.liftF(Query(no))
 }
 def store(book: Book): BookRepo[Book] = {
  Free.liftF(Store(book))
 }
 def delete(book: Book): BookRepo[Unit] = {
  Free.liftF(Delete(book))
 }

 def update(no: String, f: Book => Book): BookRepo[Book] = {
```

```
  for {
   b <- query(no)
   u <- store(f(b))
  } yield u
 }

 def delete(no: String): BookRepo[Unit] = {
  for {
   b <- query(no)
   _ <- delete(b)
  } yield ()
 }
}
```

Pro zpřehlednění je v objektu *free* nadefinovaný typový alias. Ten vyjadřuje typ free monádu, takže program, který využívá jazyk *BookRepositoryF* a má generický parametr *A*, který představuje návratovou hodnotu. Trait *BookRepository* již definuje funkce, které vrací konkrétní programy. Ty je možné komponovat dohromady. Použití funkce *Free.liftF* pouze zabaluje svůj argument do free monádu. Jak lze vidět například u funkce *update*, již zde je možné provádět kompozici programů, aniž by zde byla informace o tom, jak se programy interpretují.

Už chybí pouze konkrétní interpreter. Těch může být pro každý program samozřejmě více. Zde je důležité hlavně pokrýt celý jazyk free monádu. Interpreter je vždy vázaný na daný jazyk a pro správnou funkčnost musí pokrýt všechny elementární operace jazyka, tedy zde všechny potomky *BookRepositoryF*.

Ukázána je jednoduchá implementace repozitáře pomocí mutabilní mapy. Je to zjednodušení, které určitě není vhodné použít pro produkci, ale je vhodné například pro testování. Interpreter musí implementovat metodu, která projde postavený program (tedy postavený AST) a zařídí jeho obsluhu. Zde je konkrétní implementace s ukázkou použití:

```
trait BookRepositoryInterpreter {
 def apply[A](action: BookRepo[A]): Task[A]
}

case class BookRepositoryMapInterpreter() extends BookRepositoryInterpreter {
 val table: MMap[String, Book] = MMap.empty[String, Book]

 val step: BookRepositoryF ~> Task = new (BookRepositoryF ~> Task) {
  override def apply[A](fa: BookRepositoryF[A]): Task[A] = fa match {
   case Query(no) =>
    table.get(no)
     .map(a => now(a))
     .getOrElse(fail(new RuntimeException(s"There is no book with no $no")))
   case Store(book) => {
```

```
    table.put(book.no, book)
    now(book)
   }
   case Delete(book) => {
    now(table.remove(book.no)).void
   }
  }
 }

 def apply[A](action: BookRepo[A]): Task[A] = action.foldMap(step)
}

object LibraryFree extends App {
 override def main(args: Array[String]): Unit = {
  import BookRepository._

  val composite = for {
   _ <- store(Book("123", "Pan Prstenu", "Tolkien"))
   x <- query("123")
   _ <- delete("123")
  } yield x

  val task = BookRepositoryMapInterpreter().apply(composite)
 }
}
```

Každý interpreter musí implementovat funkci, která prochází celý vytvořený strom příkazů. Každý jeden příkaz je obsloužen funkcí *step*, která podle typu operace vykoná příslušnou obsluhu. V ukázce jsou konstrukty specifické pro scalaz jako je návratový typ interpretace, ale jejich vysvětlení není nutné pro obecné pochopení vzoru.

V ukázce použití je vidět, že je možné programy dále snadno skládat a vytvářet složitější programy. Nejdříve se vytvoří kompozice operací (AST) a až poté se přiřadí inerpreter, který provede obsluhu. Není zde nikde řečeno, že se výsledek musí spustit pouze v jednom interpreteru. Je možné například vytvořit interpreter pro logování nebo auditní logování a výsledný program spustit ve více interpreterech. Tím je možné docílit podobné funkčnosti jako u AOP.

Free monád dává sílu vytvoření DSL (domain specific language), který je možné používat nezávisle na tom, jak se bude interpretovat. Pomocí DSL lze vytvářet komplexní programy na vysoké úrovni abstrakce. To umožňuje implementaci funkčních požadavků aplikace, která je úplně nezávislé na technikáliích (například typ databáze) a již zde lze ověřit smysluplnost návrhu typovou kontrolou kompilátoru[9].

V ukázce je příklad, který reprezentuje pouze repozitář a přístup k němu. U rozsáhlejších aplikací je vhodné aplikace dekomponovat do více vrstev. Otáz-

kou je, jak skloubit více vrstev aplikace a použití free monádu.

Je důležité si uvědomit, kde přidaná hodnota free monádu (tedy nezávislost programu na interpretaci) dává smysl. V drtivé většině aplikací to je na nejnižší datové vrstvě. V produkci je možné používat relační databázi, testy spouštět nad in-memory databází. Přechod na odlišnou databázi znamená pouze nahrazení interpreteru.

Na druhou stranu, taková vlastnost nedává smysl na vrstvě s business logikou. Ta zahrnuje logiku, která je obecně platná pro danou doménu problému. Tato pravidla je nutné důkladně protestovat, protože na nich závisí správnost aplikace, ale pokaždé jsou stejná. Nedává tedy smysl více interpreterů.

Jedna z možností, jak přistupovat k vícevrstvé architektuře aplikace s použitím free monádu je, že se vytvoří vrstva pro repozitář pomocí free monádu, stejně jako je to v ukázce. Vrstva s business logikou se vytvoří jako samostatný modul, který je mixin kompozicí repozitářů. Je tedy možné uvnitř volat elementární funkce repozitářů, které jsou tvořeny free monády, a nad nimi vytvářet složitější operace s business logikou. Aplikace poté pracuje s modulem z vrstvy business logiky, ale jazyk programu tvoří pouze operace z datové vrstvy. Interpreter programu tedy pokrývá pouze jazyk datové vrstvy[2].

3.8 Shrnutí

V kapitole jsou popsány základní funkcionální návrhové vzory. Návrhový vzor je **řešení** obecného **problému** v určitém **kontextu**. Narozdíl od vzorů z objektového světa, funkcionální vzory jsou více generické a je možné implementovat abstraktní **řešení**, kterému se pouze vloží konkrétní **kontext**.

Funkcionální návrhové vzory jsou více matematické„ většinou vycházejí z **teorie kategorií**. Matematické struktury, jako je monoid nebo monád, mají jasně definované axiomy, které musí splňovat. Tyto matematické struktury jsou využívány jako návrhové vzory. Axiomy dávají jasný návod, jak vzor testovat.

Používání vzorů přináší:

- Řešení opakujících se problémů.
- Zlepšení možností kompozice, většina vzorů abstrahuje konkrétní výpočet a poskytuje velké množství kombinátorů.
- Jednotnost a lepší čitelnost kódu.
- Jasný návod jak testovat určité části kódu.

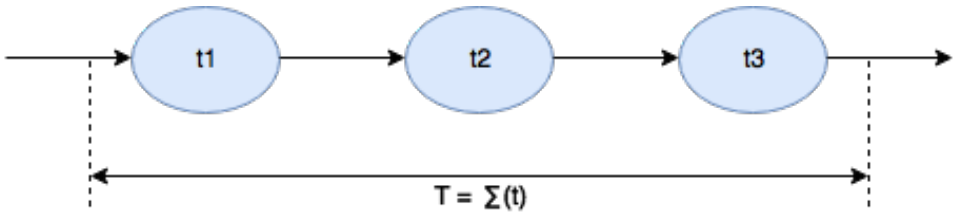
Existuje mnoho knihoven, které implementují funkcionální návrhové vzory. Příkladem je knihovna **scalaz** nebo **cats**.

Kapitola 4

Reaktivní programování

Reaktivní programování má za cíl zlepšit uživatelský zážitek z používání aplikace. Cílem je zlepšit **responsibilitu**, **resilienci** a **elasticitu** aplikace. Reaktivní programování se začíná v posledních letech čím dál více rozšiřovat, kvůli nutnosti udržet minimální dobu odezvy i u rozsáhlých aplikací, které využívá velké množství uživatelů. Základem reaktivního programování je asynchronní komunikace a velké možnosti škálování[10].

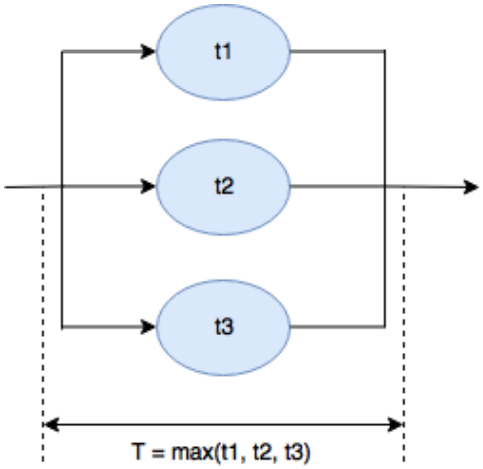
Responsivita představuje dobu odezvy systému. Aplikace, kde je dlouhá doba odezvy rychle odradí uživatele a stává se nepoužitelnou. Dlouhá doba odezvy je většinou dána množstvím přístupů k externím systémům, jako je například čtení z databáze. Na obrázku 4.1 je znázorněn sekvenční přístup k volání metod v aplikaci. Doba odezvy je potom rovna součtu dob odezvy všech těchto volání. Reaktivní aplikace tyto přístupy realizují pomocí asyn-



Obrázek 4.1: Sekvenční volání metod. Doba odezvy je rovna součtu všech latencí.

chronní komunikace a doba odezvy je tedy doba nejdelšího dotazu externího systému. Reaktivní přístup založený na asynchronní neblokující komunikaci je znázorněn na obrázku 4.2.

Resiliencí je myšlena doba odezvy a dopady na uživatele, pokud dojde k chybě. Vysokou resilienci je možné zajistit replikacemi, izolací funkcionalit uvnitř systémů a delegacemi. Je důležité, aby každá komponenta systému



Obrázek 4.2: Asynchronní volání metod. Doba odezvy je velikost nejdelší latence.

byla dostatečně izolována od zbytku aplikace. Pokud potom v některé komponentě nastane chyba, tak to nezasáhne zbytek aplikace. Obnovení komponenty s chybou bývá delegováno na jinou externí komponentu. U kritických komponent, kde je důležitá vysoká dostupnost, je možné vytvářet replikace, které jsou využity v době výpadku. Chyba by dále neměla mít negativní dopad na uživatele[10].

Elasticita je určena schopností systému se přizpůsobit nárazovým přílivům většího množství uživatelů. Například u e-shopů se očekává, že v období před Vánocemi dochází k většímu množství nákupů. Dobré elasticity lze dosáhnout vysokou možností horizontálního škálování systému[10].

V této části je popsáno, jak reaktivity dosáhnout pomocí monádů a reaktivních streamů.

4.1 Future

Future je ve Scale implementace monádu, která abstrahuje latenci. Výpočet, zabalený ve Future, se provádí asynchronně a neblokuje hlavní vlákno. Jelikož je Future monád, tak implementuje metody *map* a *flatMap*, díky kterým lze použít v for-comprehension. Další vlastnosti future je popsáno na následující ukázce:

```
import scala.concurrent.ExecutionContext.Implicits.global

def sleep(time: Long): Future[Unit] = Future {
 Thread.sleep(time)
}
```

```
val time = 1000

val start = System.currentTimeMillis()
val res = for {
 _ <- sleep(time)
 _ <- sleep(time)
 _ <- sleep(time)
 _ <- sleep(time)
} yield ()

res.onComplete({
 case Success(_) =>
  println(s"Time = ${System.currentTimeMillis() - start}")
 case Failure(_) => println(s"Failed!")
})
```

Ukázka začíná importem implicitní hodnoty **ExecutionContext**, která je nutná pro práci s Future. Tato defaultní hodnota využívá work-stealing thread pool, která používá stejný počet vláken jako je počet dostupných procesorů.

Dále je zde funkce *sleep*, která pouze uspí vlákno na čas daný parametrem. Zde je důležitý návratový typ, kterým je *Future[Unit]*, což je asynchronní volání, které nic nevrací. V těle funkce je pouze vytváření Future pomocí funkce *apply*, která má rozhraní:

```
def apply[T](body: =>T)(implicit executor: ExecutionContext): Future[T]
```

Již zde je potřeba, aby v aplikaci byla nastavena implicitní hodnota pro ExecutionContext. *apply* spustí asynchronní výpočet a zabalí ho do Future.

Dále je ukázána kompozice více Future dohromady pomocí for-comprehension, jejíž návratový typ je opět Future a tedy ani zde se neblokuje hlavní vlákno.

Výsledek výpočtu se zpracovává přidáním callbacku, který výsledek zpracuje. Callback se přidává pomocí funkce *onComplete*, jejíž rozhraní vypadá následovně:

```
def onComplete[U](f: Try[T] => U)(implicit executor: ExecutionContext): Unit
```

Jako parametr se předává funkce typu *Try[T] => U*, kde *U* je zde jenom proto, aby byla akceptována libovolná funkce, její výsledek se ale zahazuje. Zajímavější je vstupní parametr funkce - *Try[T]*. Try je monád, který abstrahuje možnost chyby, takže je vhodné ošetřit možné chyby při výpočtu. Callback se provolá, až když se dokončí výpočet, zase tedy neblokuje hlavní vlákno.

V ukázce se tedy vytvoří Future, který se skládá ze 4 asynchronních volání, které pouze uspí vlákno na 1000ms. V callbacku se vypíše doba trvání výpočtu. Ošetřuje se i pád volání, ale to je pouze ilustrativní, reálně to v ukázce nemůže nastat.

Spuštění tohoto programu skončí velmi rychle a nic se nevypíše. To je

proto, že po zavěšení callbacku hlavní vlákno již nic nevykonává a může tedy skončit. Tím skončí i celý program. Všechny rozpracované výpočty probíhají ve vedlejších vláknech a ty zaniknou s hlavním vláknem. Pro výpis z callbacku je nutné udržet hlavní vlákno naživu déle než trvá výpočet, takže například uspáním.

Future je možné využít za účelem zlepšení reaktivity aplikace tak, že všechny akce, které můžou potenciálně dlouho trvat, se zabalí do Future. Mezi takové akce patří přístup k databázi, volání externích systémů, anebo složité výpočty. Všechny tyto akce probíhají asynchronně a blokují hlavní vlákno. Výsledek bude využit až když doběhne výpočet.

4.2 Reaktivní streamy

Na enterprise aplikace lze nahlížet jako na proud dat, nad kterým se provádí operace. Uživatel vyvolá událost, ta většinou vygeneruje nějaký dotaz na backend. Tam se provedou validace, transformace a operace se promítne do databáze. Je to pouze proud dat od uživatele do databáze, který se různě transformuje. Naopak pokud si uživatel pouze prohlíží nějaké informace, tak je to proud dat z databáze směrem k uživateli. Přesně takhle jsou modelovány aplikace pomocí reaktivních streamů.

Stream v kontextu programování je obecně proud dat. Lze si ho představit jako orientovaný grav začínající u producenta, který generuje data. Z producenta graf vede přes libovolné množství uzlů, které stream zpracovávají, do konzumenta. Konzument může například ukládat data do databáze, nebo je zobrazit uživateli. Reaktivní streamy jsou založeny na asynchronním zpracovávání streamů.

Obrázek 4.3: Orientovaný graf reprezentující stream dat.

V této práci je popsáno použití streamů z frameworku **Akka Streams**, proto je využíván slovník tohoto frameworku. Základní prvky jsou[11]:

Source producent dat. Zde začíná graf. Jeho rozhraní je *Source[+Out, +Mat]*, kde *Out* je typ generovaných dat.

Sink konzument dat. Zde graf končí. Rozhraní je *Sink[+In, +Mat]*, kde *In* je typ příchozích dat.

Flow je uzel, který provádí zpracování. *Flow[-In, +Out, +Mat]* přijímá data typu *In* a generuje data typu *Out*.

RunnableGraph tvoří spustitelný graf, ve kterém už je jasně daná jeho topologie *Source -> Flow* -> Sink*.

Generický parametr *Mat* určuje typ, který uzel publikuje do externích systému. Může se jednat například o posílání emailů nebo logování statistik. Pokud uzel nic nepublikuje, tak je možné použít typ *NotUsed*.

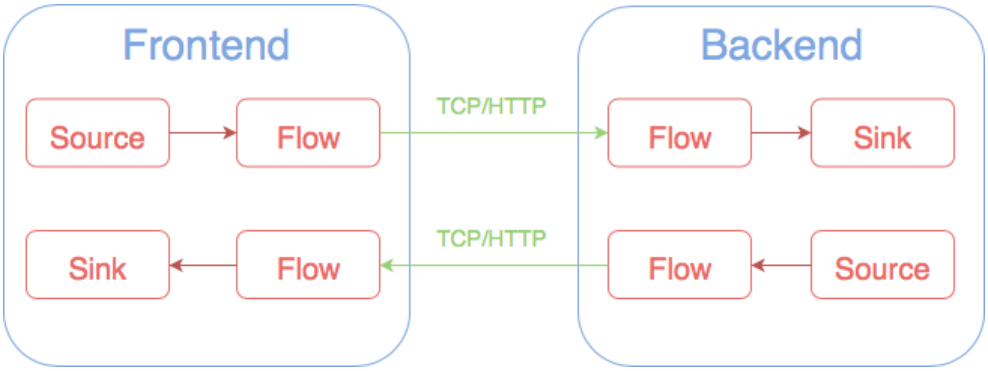
Jednoduchý příklad vytvoření grafu a spuštění streamu je ilustrován na následující ukázce:

```
import scala.concurrent.ExecutionContext.Implicits.global

implicit val system = ActorSystem("reactive-streams")
implicit val materializer = ActorMaterializer()

val source: Source[Int, NotUsed] = Source(1 to 10)

val sink: Sink[String, Future[Done]] = Sink.foreach(println)

val flow: Flow[Int, String, NotUsed] = Flow[Int].map(_.toString)

val graph: RunnableGraph[Future[Done]] =
  source.via(flow).toMat(sink)(Keep.right)

graph.run()
```

Streamy pracují s Future, takže je nutné mít definovanou implicitní hodnotu pro ExecutionContext. Běh streamu dále vyžaduje **Materializer**. Materializer je fabrika, která vytváří výpočetní jednotky pro stream. V ukázce je použit aktorový systém z Akka, který funguje na principu asynchronních aktorů. Zbytek je přímočarý, Source vygeneruje stream čísel od 1 do 10, flow provádí konverzi na String a sink každý prvek vypíše. Postavení grafu znamená vytvoření topologie a *run* spustí asynchornní a neblokující vyhodnocování.

Enterprise aplikace se skládají z frontendu a backendu, kde backend může být realizován jako monolit nebo microservices. Komunikace mezi komponentami bývá obvykle synchronní a blokující pomocí REST nebo SOAP rozhraní. Reaktivní streamy lze ale použít i na neblokující síťovou komunikaci. Akka streams nabízí možnost využití TCP protokolu. Další a možná používanější možnost je použití frameworku Akka Http, který umožňuje streamování pomocí HTTP protokolu uvnitř web socketu. Komunikace mezi komponentami je znázorněna na obrázku 4.4.

Největší výzvou pro implementaci reaktivních streamů je zpětná propagace. Může nastat situace, kdy je producent rychlý a nějaký uzel nestíhá obsluhovat příchozí data. Pokud nedojde k regulaci, tak dojde k přehlcení a může to skončit pádem aplikace (např. vyčerpáním paměti). Na obrázcích byl znázorněn pouze jeden směr proudu dat, ale to pouze pro jednoduchost. Reálně je nutné, aby se zpětně propagovala minimálně informace o zahlcení



Obrázek 4.4: Komunikace frontend-backend pomocí reaktivních streamů.

a kolik dat zvládá uzel zpracovat. To se nazývá **back-pressure**, tedy zpětná propagace. Většina již hotových implementací (např. zmíněná Akka Streams) již řeší zpětnou propagaci. Podrobnější informace lze nalést ve vyčerpávající dokumentaci[11].

4.3 Shrnutí

Reaktivní programování má za cíl zlepšit uživatelský zážitek z používání aplikace. Je založeno na asynchronní komunikaci a tedy vysoké míře paralelizace. Funkcionální programování minimalizuje měnný stav, čímž zjednodušuje paralelizaci, a proto je vhodné pro implementaci reaktivní aplikace.

V této části je ukázáno, jak je možné jednoduše dosáhnout vyšší reaktivity pouze pomocí standardní knihovny jazyku Scala s využitím **Future**. Future je implementace návrhového vzoru monád, která spouští výpočet asynchronně. Na future lze navěsit **callback**, který se spustí po doběhnutí výpočtu.

Další způsob, jak docílit reaktivity je pomocí reaktivních streamů. Stream je reprezentován orientovaným grafem, který znázorňuje proud dat. Producent generuje data. Ty dále prochází přes libovolný počet uzlů, kde každý uzel provádí zpracování dat. Proud dat končí u konzumenenta. Reaktivní streamy jsou neblokující a asynchronní. Mezi nejpoužívanější implementace patří **Akka Streams**.

Existuje ještě mnoho způsobů, jak lze dosáhnout reaktivity. Future patří mezi nejjednodušší a reaktivní streamy patří mezi dnes nejpropagovanější metody. Další možný způsob je **aktorový model**. Ten je založen na asynchronním a neblokujícím odesílání zpráv mezi aktory. Aktor je abstrakce synchronní výpočetní jednotky, která s okolním světem komunikuje pouze pomocí posílání zpráv. Vnitřní implementace aktora tedy může zahrnovat měnitelný stav, protože je jisté, že k němu nemá možnost přistoupit jiné vlákno.

Kapitola 5

Persistence

Tato kapitola se zabývá přístupem k persistenci dat. Práce je zaměřena na funkcionální principy a bylo toho hodně řečeno o tom, že jedním z principů je minimalizace mutabilního stavu. Většina aplikací má ale nějaký vývoj v čase a objekty, se kterými se pracuje, svůj stav v čase mění. Navíc je nutné mít k aktuálnímu stavu rychlý přístup a připravit aplikaci pro obnovení z pádu. Proto je nutné data uchovávat v persistentním uložišti.

Při návrhu aplikace většina programátorů hrozně rychle zabředává do modelování doménových objektů, tedy statického obsahu aplikace, který representuje jejich stav. Tento stav je poté uchováván databázi, nejčastěji relační, a nad ním se vystaví **CRUD** operace.

Pomocí CRUD operací lze měnit stav objektů. Přitom operace update a delete jsou dost destruktivní, jejich vykonání úplně zahodí všechny předchozí události, což může způsobovat problémy. Může dojít k chybě, nebo je potřeba vrátit operaci. V případě mazání dokonce objekt úplně zaniká a historie k němu tedy neexistuje. Tyto problémy je samozřejmě možné řešit. K objektům se přidávají značky, které reprezentují životní cyklus objektu (mazání je tedy změna značky a ne reálné mazání), vytváří se důkladný log všech akcí, z kterého je možné dohledat historické změny a případně obnovit stav do určitého okamžiku v historii.

V reálném světě to takhle ale nefunguje, objekty v reálném světě neexistují bezdůvodně v aktuálním stavu. V reálném světě se v čase odehrávají události, které mění stav objektu. Aktuální stav je poté aplikace všech událostí v historii. Přesně takhle funguje **event-sourcing**. To je odlišný přístup k modelování dat. V persistentním úložišti se neuchovává aktuální stav objektu, který je měnný, ale uchovávají se neměnné události v čase. Svět se tedy nemodeluje pomocí statických objektů, ale pomocí dynamických akcí, které svou aplikací generují tyto statické objekty.

Tento přístup k persistenci již nepotřebuje přídavné mechanismy, pro obnovu dat a uchování historie. Již z definice tohle všechno umí sám o sobě. Mezi další výhody tohoto přístupu patří škálovatelnost. Není zde žádný měnný stav,

události jsou sami o sobě neměnné.

V této části je popsán event-sourcing a proč se jeho použití hodí ve funkcionální aplikaci. Dále je popsán model **CQRS** a jeho použití v kombinaci s event-sourcing.

5.1 CQRS

Command Query Responsibility Seqregation je model, ve kterém je striktně oddělen zápis (command) a čtení (query). To umožňuje:

- škálování,
- definici různých agregátů pro stejná data.

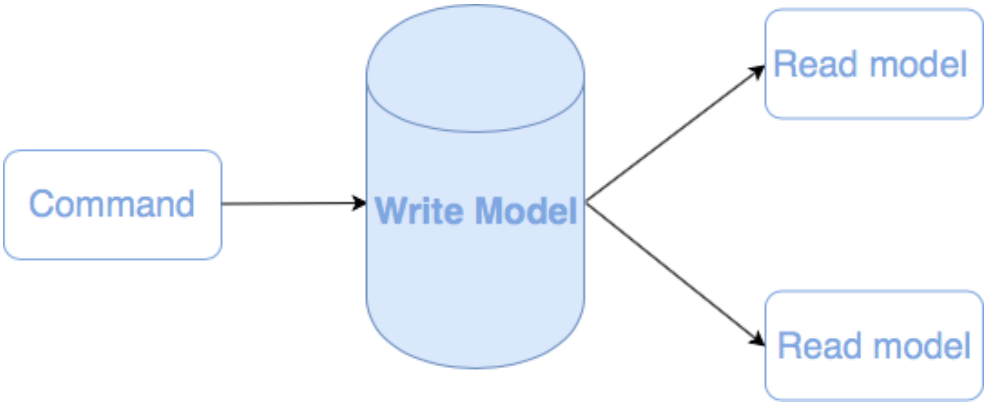
Druhý bod není možná úplně zřejmý. V CQRS vzniká persistentní stav na základě příchozích příkazů (command) v oddělené databázi pro zápis. Tato databáze se nazývá **write model** a ten existuje právě jeden. Write model se modeluje tak, aby databáze byla co nejspolehlivější, není zde žádoucí optimalizovat databázi pro dotazování. Při použití s CRUD modelem je možné například přidat atributy životního cyklu objektu, nebo podpůrné prvky pro uchování historie. Write model je autoritativní zdroj pravdy[2].

CQRS striktně odděluje čtení od zápisu. Pro zápis slouží příkazy, které zapisují do write modelu. Čtení mají na starost dotazy (query), které čtou data ze striktně odděleného **read modelu**. Write model může být realizovaný například pomocí relační databáze a read model pomocí grafové.

Další důležitou vlastností je, že je možné pro jeden write model vytvořit více read modelů. Jak bylo řečeno v části o DDD, tak správně navržený agregát by měl být popsán jazykem daného kontextu a tedy jeho signatura by měla být čitelná pro každého, kdo rozumí danému kontextu. Ten stejný model ale může vystupovat ve více kontextech a v každém je vhodná odlišná signatura. To CQRS umožňuje pomocí vytvoření více read modelů nad jedním write modelem (viz obrázek 5.1).

Read model obsluhuje dotazy. Proto je nutné modelovat agregáty tak, aby vyhovovali požadovaným dotazům. Je tedy důležité vytvořit read model tak, aby byl vhodný pro charakter dotazů nezávisle na tom, jak vypadá write model (jak bylo řečeno, je možné využít i úplně odlišný typ databáze).

Oddělené obslužné machanizmy pro zápis a čtení zlepšují možnosti škálování, ale hlavně přináší možnost vytvoření write modelu, který obsahuje prvky vhodné pro persistentní stav a modelování životního cyklu doménových objektů. Naproti tomu read model obsahuje pouze data potřebná pro daný kontext, který tento read model využívá.



Obrázek 5.1: V databázi je uložen write model, který je měněn pomocí příchozích příkazů. Nad jedním write modelem může být vystavěno více read modelů.

5.2 Event-sourcing

Event-sourcing je přístup k persistenci, ve kterém se svět modeluje pomocí událostí, které mohou reálně nastat. V databázi se uchovávají všechny události, které jsou neměnné a doménové objekty vznikají aplikací všech těchto historických událostí.

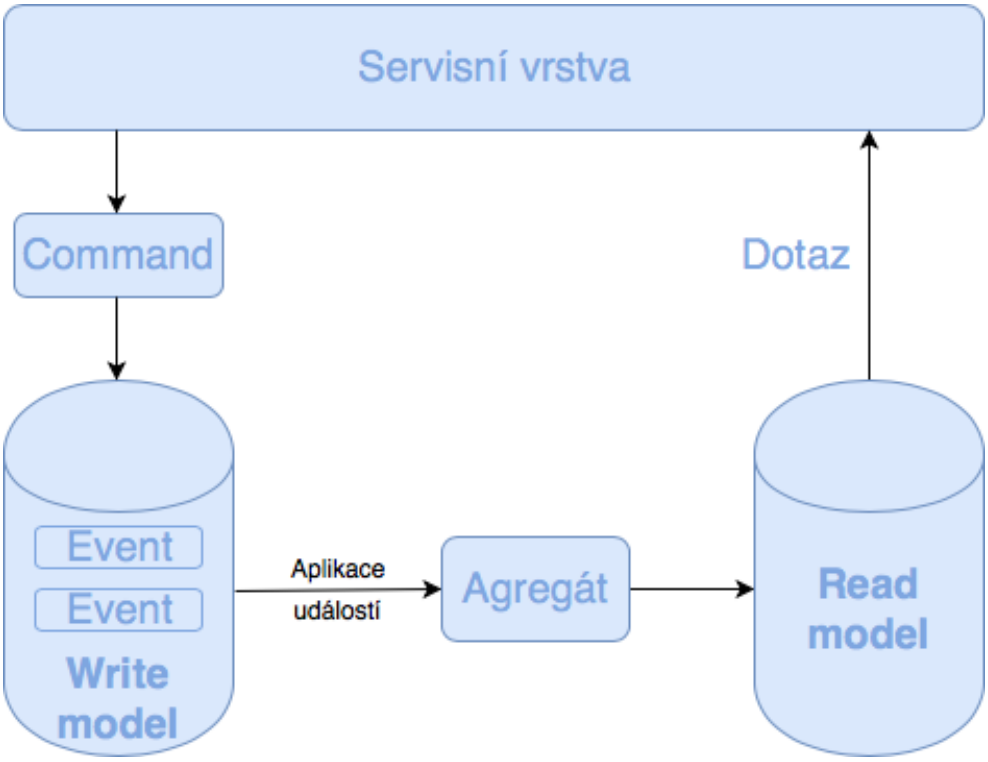
Tento přístup je daleko více funkcionální, než konvenční CRUD model. Pracuje se s imutabilními událostmi. Neexistují zde destruktivní operace update nebo delete, pouze se komplementárně přidávají nové neměnné události. Navíc v databázi nejsou uchovávány agregáty (s výjimkou optimalizací), ale události, které jsou vlastně reprezentací funkce, jejichž aplikace formuje agregát.

To, na rozdíl od CRUD modelu, vytváří zpětně obnovitelný model do libovolného okamžiku v historii bez podpůrných a složitých mechanismů. Navíc zde neexistuje žádný sdílený měnný stav, takže je možná paralelizace.

Pokud by se při každém čtení měli aplikovat všechny události v historii, tak čtení probíhá v lineárním čase. S rostoucím počtem událostí se zpomaluje databáze. Proto se pro čtení běžně vytváří snapshoty aktuálního stavu agregátů, které jsou pak čteny v konstantním čase.

Vytváření snapshotů není přímo součástí principu event-sourcing, je to pouze optimalizace. Snapshot je vlastně vymodelovaný agregát, který je vhodný pro čtení, ale úplně neodpovídá tomu, jak jsou data uchovávána. Je tady určitá analogie s CQRS. Zdrojem pravdy je posloupnost událostí a čte se snapshot, který komplementárně vzniká přidáváním nových událostí. Proto se nabízí využít tyto principy společně. Schéma datové vrsty pomocí principů CQRS a event-sourcing je znázorněno na obrázku 5.2.

Je důležité si uvědomit, že příkaz a událost jsou dvě odlišné věci. Per-



Obrázek 5.2: Použití event-sourcing v kombinaci s CQRS.

sistentní vrstva obdrží příkaz a nad ním spustí obslužný mechanismus. Zde je vhodné místo pro validace a je možné zde volat i externí systémy (například pro posílání informačních emailů). Pokud vše skončí úspěšně, tak se teprve vytvoří událost, která se uloží do databáze. Proto je vhodné pojmenovat příklazy slovesem v přítomném čase, čímž je zdůrazněno, že se daný příkaz předává k vykonání. Naopak událost je něco, co již v minulosti proběhlo. Pro pojmenování je vhodné sloveso v minulém čase.

Při čtení, tedy aplikací všech událostí v historii, není chtěné vykonat stejné obslužné mechanismy jako u příchozího příkazu. Není nutné spouštět validace, těmi události prošli již při vytváření. Už vůbec není zádoucí volání externích systémů.

Pro implementaci event-sourcing lze použít již vytvořené frameworky. Mezi nejpoužívanější patří **Akka Persistence**, který pracuje s aktorovým modelem. Framework je koncipován pro použití event-sourcing v rámci CQRS. Rozhraní pro čtení dat je implementováno reaktivním streamem, který umožňuje snadnou synchronizaci dat mezi read a write modelem. Více informací lze dohledat ve vyčerpávající dokumentaci[12].

5.3 Shrnutí

Persistence se v aplikaci stará o uchovávání dat. Pro enterprise aplikace je persistence klíčová část a její kvalita může zásadně poznamenat kvalitu celé aplikace. Databáze by měla obsahovat v každém okamžiku konzistentní data. Když se databáze dostane do nekonzistentního stavu, tak by mělo být možné jednoduše obnovit data do posledního konzistentního stavu. U rozsáhlých aplikací, které využívá mnoho uživatelů, je také důležitá vysoká rychlost zápisu a čtení.

Nejpoužívanější přístup k persistenci je s využitím relační databáze a CRUD modelu dat. Tento přístup má hodně blízko k objektovému programování, záznam v databázi představuje objekt v programu a v čase se mění jeho stav. Takový model je kvůli sdílenému stavu obtížné škálovat. Pro obnovu dat je nutné důkladné logování všech provedených transakcí.

V této kapitole je ukázán odlišný přístup k persistenci dat, event-sourcing, který má blíže k funkcionálnímu programování. Do databáze se ukládají neměnné události, jejichž aplikace vytváří stav. Takovou databázi je jednodušší škálovat a pro obnovu dat do konzistentního stavu není potřeba žádný externí systém - databáze je sama o sobě log. Nevýhodou je čtení, které znamená aplikaci všech událostí v historii. To lze vyřešit pomocí CQRS modelu.

CQRS je přístup k persistenci dat, kde je striktně oddělen write model, do kterého se zapisují data a je to jediný zdroj pravdy a read model, který obsluhuje dotazy a je synchronizován s write modelem. Nad jedním write modelem může být více read modelů.

Použití event-sourcingu v kombinaci s CQRS tedy znamená vydefinování událostí jako statických dat a příkazů, které spouští obslužné mechanismy a generují událost. Pokud se k tomu přidá modelování událostí jako ADT a obalení do monádu v obsluze příkazu, tak je to velmi podobné návrhovému vzoru **free monád**.

Kapitola 6

Praktická část

V této části je popsána implementace jednoduché aplikace v jazyku Scala, která ilustruje použití popsaných funkcionálních principů v kontextu vývoje podnikových aplikací. Součástí popisu je popis použitých důležitých frameworků a poznatky z jejich použití.

Aplikace má za cíl správu uživatelů. U každého uživatele se uchovává jméno, emailová adresa a heslo. Aplikace umožňuje vytváření a editaci uživatelů ve webové aplikaci.

Implementace pouze ilustruje popsané principy a proto je zvolena jednoduchá aplikace. Rozšíření by znamenalo vytvoření dalších modulů, které by vypadali obdobně.

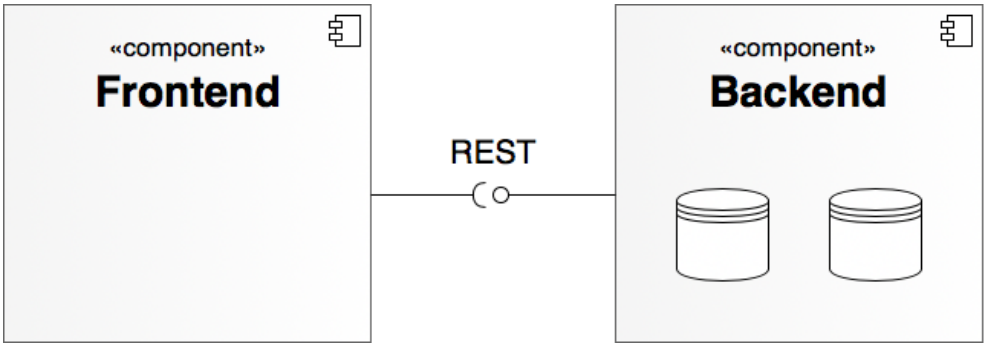
Součástí přiloženého CD jsou spustitelné aplikace s návodem, jak lze aplikaci spustit. Dále obsahuje zdrojové kódy s popisem, jak lze kód kompilovat a vytvořit opět spustitelný program.

6.1 Fyzické členění aplikace

Současná implementace se skládá z jednoho backendu a webové klientské aplikace. Komunikace je realizována pomocí REST webových služeb. Backend přistupuje k databázi a vystavuje REST rozhraní, které umožňuje manipulaci dat.

6.2 Backend

Backend představuje serverovou část s business logikou, která přistupuje k databázi a vystavuje REST rozhraní pro klientské aplikace. Aplikace je kompilována jako jar archiv a může být spuštěna v libovolném aplikačním serveru nebo kontejneru. Základní adresářová struktura aplikace je znázorněna na obrázku 6.2.



Obrázek 6.1: Fyzické členění aplikace.

- `api` ..... zde se nachází vystavěné rozhraní
- `domain` ..... zde se nachází doménové objekty
- `program` ..... zde se nachází programy, které vytváří AST z free monádů
- `repository` ..... zde se nachází algebra pro repozitáře
  - `handler` ..... zde se nachází interpretery algeber repozitářů
- `service` ..... zde se nachází moduly slučující více algeber
  - `utils` ..... zde se nachází podpůrné algebry
    - `handler` ..... zde se nachází interpretery podpůrných algeber

Obrázek 6.2: Adresářová struktura backendu.

6.2.1 Architektura - free monád

Architektura aplikace je řešena pomocí návrhového vzoru **Free monád** z frameworku **Freestyle**. Freestyle využívá makra, díky kterým je kód zbaven konstruktů, které jsou nutné pro operace nad free monády, ale nesouvisí s business logikou aplikace.

Návrhový vzor free monád umožňuje vytvoření DSL, z kterého se vytváří programy pouze jako statická data (AST). Tyto programy jsou poté spuštěny interpreterem, který prochází vytvořený AST. Program se vytváří kompozicí free monádů, které tvoří základní stavební jednotku AST. Ve frameworku Freestyle se free monády umisťují do *trait*, který má anotaci *@free* a nazývají se free algebry. V ukázkové aplikaci se nachází 3 free algebry:

UserCommandRepo zahrnuje write operace nad databází.

UserViewRepo zahrnuje read operace nad databází.

IdGenerator generuje unikátní ID.

Rozdělení repozitáře na read a write je zde kvůli větší robustnosti. Takto je možné vytvořit persistentní vrstvu pomocí event-sourcingu a CQRS. Toto

dělení k tomu vede, ale není to svazující. Oba interpretery můžou klidně přistupovat k jedné relační databázi s CRUD modelem dat.

V adresáři **service** se nachází moduly, které vytváří kompozici free algeber a tvoří nad nimi komplexnější operace. Moduly jsou zase *trait* a jsou anotovány *@module*. Free algebry, které modul využívá jsou přidány jako deklarace konstant. Kompozice free algeber je v knihovnách, které nevyužívají makra docela obtížné. Je nutné vytvořit tzv. coproduct a ten se v knihovně scalaz vůbec nenachází. Freestyle vytváří coproduct pomocí maker a kód je tedy stručný a zbaven komplexity, která nesouvisí s business doménou. Aplikace se aktuálně skládá z jednoho modulu - *UserService*. V něm se nachází veškeré operace nad uživateli. Operace provádějící zápis jsou důkladně logovány.

Adresář **program** obsahuje singletony, které pomocí DSL vytváří programy nad modulem. Vytvoření programu znamená postavení celého AST, který je možné interpretovat. Zde se nachází pouze jeden objekt - *UserProgram* - ve kterém se nachází funkce vytvářející programy nad uživateli.

Programy jsou využívány v adresáři *api*, kde se nachází implementace vystavěných API backendu. Teprve zde se interpretují programy.

6.2.1.1 Poznatky

Použití free monádu určitě patří mezi nejzajímavější části celé práce. Tento strukturální vzor umožňuje vytvoření jednoduše rozšiřitelného DSL, z kterého se vytváří spustitelné programy. To vše bez jakékoliv znalosti, jak se program bude interpretovat. Je možné vytvářet programy na vysoké úrovni abstrakce, které splňují všechny funkční požadavky, a až poté implementovat interpretery, které představují technickou stránku programu. Aplikace používající free monád má také vysokou míru SoC (Separation of Concerns).

Pro implementaci byl zvolen framework Freestyle, kvůli jeho jednoduchosti. Pomocí maker umožňuje vytvářet složité coprodukty free algeber a podstatně zjednodušuje implementaci interpreterů.

V teoretické části bylo ukázáno použití free monádu bez maker. Interpreter poté vypadá tak, že jedna funkce obsluhuje všechny typy operací. Je tedy nutný pattern matching přes všechny typy operací. To je takové úzké hrdlo. Pokud by algebra definovala hodně operací, tak se velmi rychle stane kód nepřehledný. Interpreter ve Freestyle vypadá daleko líp. Obsluha každého typu operace je znamená implementaci jedné funkce.

Makra mají ale i negativní dopady. Aplikace byla vyvíjena v IntelliJ IDEA Ultimate, což je nejspíš nejlepší IDE pro jazyk Scala. Ani IDEA se ale nedokázala vypořádat s makry. Hodně částí kódu hlásí warning nebo dokonce error, ale kompilace proběhne v pořádku (bez warningů a errorů). Je to důsledek různých značek, které si makra vytváří. To velmi znepříjemňuje vývoj.

Další nevýhodou jsou implicitní parametry a konverze, které má v sobě framework. Pokud se nepovede kompilace kódu, tak to nemusí znamenat přímo chybu programátora, ale zapomenutý import s nadefinovaným implicitním pa-

rametrem. Ten IDE samozřejmě nežádá, protože implicitní parametr využívá nějaká funkce uvnitř frameworku. Po importu je označen IDE jako nepoužitý. Automatické optimalizace importů tedy nejsou možné.

6.2.2 Persistence

Persistentní vrstva je řešena pomocí event-sourcingu v kombinaci s CQRS. Jako databáze pro write i read model je použita relační databáze PostgreSQL. Aplikace tedy pro svůj běh vyžaduje spuštěný a správně nastavený PostgreSQL.

Aplikace defaultně pracuje s následující konfuguracemi PostrgeSQL:

Port: 5432

Uživatel: gethido

Heslo: gethido

Název databáze: gethido

Konfigurace lze měnit v souboru **src/main/resources/application.conf**.

6.2.2.1 Write model

Write model je implementován jako event-sourcing uložiště a je to jediný zdroj pravdy. V databázi se nachází dvě tabulky - **journal** a **snapshot**. V journal se uchovávají všechny události a jsou označeny podpůrnými parametry. Snapshot slouží pro rychlou obnovu v případě restartování serveru.

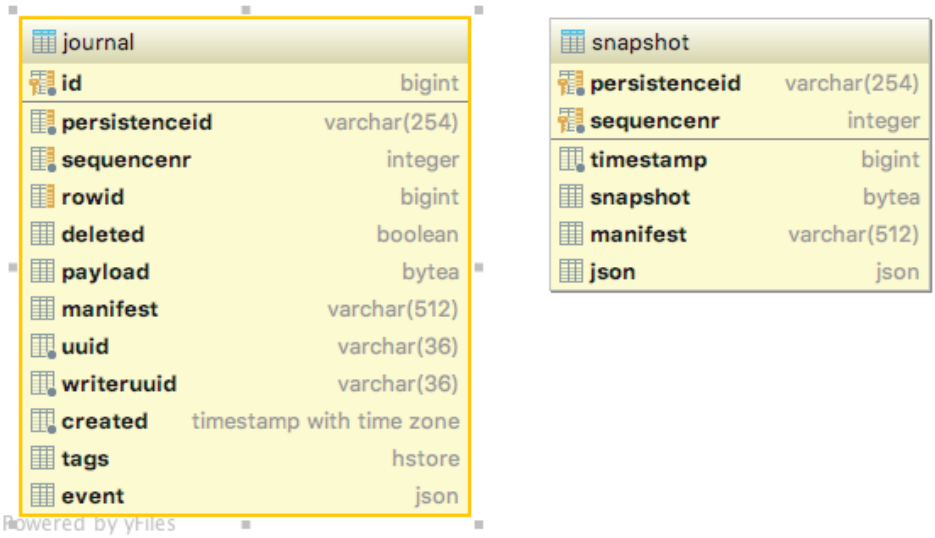
Aplikace pracuje s následujícími událostmi:

```
case class UserCreated(id: Long, name: String, email: String,
  password: String)
case class EmailUpdated(id: Long, email: String)
case class NameChanged(id: Long, name: String)
case class PasswordChanged(id: Long, password: String)
```

Události jsou modelovány jako ADT a reprezentují již proběhlé akce nad agregáty.

Implementace využívá framework Akka Persistence. Ten pracuje s aktorovým modelem. Pro každý agregát se vytváří tzv. persistentní aktor, potomek *PersistentActor* z frameworku. Ten je označen unikátním identifikátorem *persistenceId*. Aktor si drží aktuální stav agregátů (snapshot) a musí implementovat minimálně 2 funkce:

- receiveCommand
- receiveRecover



Obrázek 6.3: Schéma databáze pro write model.

receiveCommand obsluhuje všechny příchozí události, které se ukládají do databáze. Zde je vhodné místo pro validace. Pokud je událost validní, tak ji aktor aplikuje na svůj aktuální stav a ukládá ji do databáze. Dála může provádět různé podpůrné akce jako ukládání snapshotu, logování nebo posílání notifikačních emailů.

receiveRecover pouze obnovuje svůj stav z databáze. Funkce je tedy volána pouze při spouštění. Zde již nejsou nutné validace a podpůrné akce jako logování nebo posílání emailů už vůbec ne.

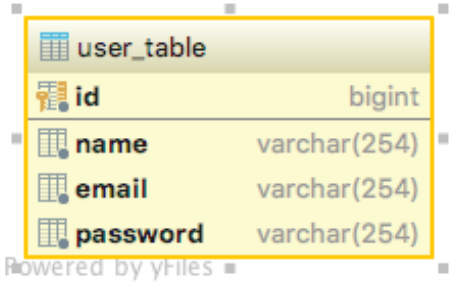
Aktuální snapshot si aktor drží kvůli validacím příchozím příkazů. Pokud přijde událost, která provádí update uživatele, tak aktor musí být schopný rychle zjistit, jestli takový uživatel vůbec existuje.

Pro čtení z journalu slouží *ReadJournal* rozhraní. To umožňuje spuštění reaktivního streamu, který obsluhuje příchozí události. U Akka Persistence se očekává, že persistentní aktor je použit v rámci CQRS a reaktivní stream je vhodný pro synchronizaci mezi read a write modelem. Aktuální řešení spouští stream po dokončení obnovy persistentního aktora.

6.2.2.2 Read model

Read model je řešen jako klasická relační databáze s jedinou tabulkou *user_table*. Pro práci s databází je použit framework pro funkcionálně relační mapování **Slick**. Ten umožňuje vytváření dotazů, které se podobají klasické práci s kolekcemi ve Scale.

Zápis do databáze provádí pouze synchronizační reaktivní stream, který propaguje události uložené do write modelu.



Obrázek 6.4: Schéma databáze pro write model.

6.2.2.3 Poznatky

Použití CQRS a ES je potřeba zvážit. Přínosy jsou znatelné až u rozsáhlejších aplikací, které obsluhují spousty požadavků a případné nekonzistence můžou mít znatelné dopady. Pro triviální aplikace je použití zbytečně komplikované a snižuje čitelnost kódu. Read model a write model je poměrně hodně provázaný a změna atributu u agregátu nejspíš vyžaduje změnu v obou modelech.

Zajímavý je dopad na modelování dat a to zejména u write modelu. U relační databáze se rozsáhlý agragát promítne do několika tabulek, které jsou propojené. Update je potom velmi komplikovaný, protože se musí řešit způsob propsání dat do propojených tabulek. U event-sourcingu tento problém úplně odpadá. Každý update nebo vytváření znamená pouze uložení poměrně jednoduché události do append-only úložiště.

Mohlo by se zdát, že problém s propojenými tabulkami se pouze odsune do read modelu. Ten ale slouží pouze k dotazování a to většinou není potřeba číst celý rozsáhlý agregát. V read modelu lze uchovávat pouze podčásti agregátu, které jsou optimalizované pro čtení.

6.2.3 API

Rozhraní backendu je implementováno pomocí REST webových služeb. Zde je využit framework Akka Http, který kromě REST také umožňuje vytvoření streamového rozhraní.

Backend definuje následující rozhraní:

GET /user/{id} vrací uživatele s daným ID.

GET /user/all vrací všechny uživatele.

GET /user/name/{name} vrací uživatele s daným jménem.

POST /user/email aktualizuje email uživatele.

POST /user/name aktualizuje jméno uživatele.

POST /user/password aktualizuje heslo uživatele.

POST /user/create vytvoří nového uživatele.

Podporovaný *content-type* je *application/json*.

Rozhraní pro aktualizaci uživatele očekává v těle dotazu JSON objekt ve formátu:

```
{
 "id": "string",
 "value": "string"
}
```

Rozhraní pro vytvoření nového uživatele očekává v těle dotazu JSON objekt ve formátu:

```
{
 "name": "string",
 "email": "string",
 "password": "string"
}
```

GET služby vrací uživatele nebo kolekci uživatelů ve formátu:

```
{
 "id": "string",
 "name": "string",
 "email": "string",
 "password": "string"
}
```

Jako identifikátor uživatele v json objektech je schválně použit String. Některé jazyky (např. JavaScript) nemají v základu podporu pro Long, proto je zvolen robustnější String, který se na backendu převede na Long.

6.3 Frontend

Klientská aplikace vytváří rozhraní pro interakci s uživatelem pomocí webové aplikace běžící v prohlížeči. Implementační jazyk je Scala, ale s využitím kompilátoru Scala.js. Kompilace tedy vytváří javascriptové soubory, které je možné vložit do statické HTML stránky a aplikace může běžet například v apache.

Jako hlavní framework pro tvoření UI je použit **scalajs-react**, který zabaluje ReactJS pro použití ve Scale.js. React je navržen pro snadný vývoj reaktivních UI. Tvorba UI znamená tvoření komponent, které je možné snadno skládat. Reaktivity je docíleno asynchronními callbacky, v kterých se nachází logika a volání externích systémů.

Aplikace se skládá ze 3 obrazovek:

- přehled všech uživatelů,
- editace existujícího uživatele,
- vytvoření nového uživatele.

Každá obrazovka je implementována jako komponenta.

Komponenta má 2 základní atributy:

- Props,
- State

Props reprezentuje vstupní parametry komponenty, které řídí, jak se má vykreslit. Jejich hodnota by se neměla promítnout ve vykreslené komponentě. Určují jak se má komponenta zobrazit, ale neurčují její obsah. Jejich stav je v životním cyklu komponenty neměnný.

State na druhou stranu určuje vykreslený obsah. Stav je implementovaný pomocí **State monádu** a v době životního cyklu komponenty může být modifikován. Jediný přímý přístup ke stavu je ale možný pouze při vykreslování komponenty, kde se navážou atributy stavu komponenty na zobrazené položky. Po vykreslení komponenty je přístup ke stavu možný pouze přes asynchronní **callbacky**, které můžou měnit stav a ten se reaktivně propisuje na navázané zobrazované prvky komponenty. Volání externích systémů také patří do callbacku.

Komunikace se serverovou částí je implementována pomocí AJAX uvnitř callbacků, takže probíhá asynchronně a je neblokující.

6.3.1 Poznatky

Použití Scala.js místo klasického JavaScriptu přináší typovou kontrolu v compile time a je možné využít velké množství Scala knihoven. React je již v základu tvořen hodně funkcionálně a reaktivně a v kombinaci s jazykem Scala to jde skvěle dohromady.

Jednou z nevýhod je zpětná kompatibilita frameworků. Existuje velké množství frameworků využívajících Scala.js a rychlost vydávání releasů se liší. Při používání více frameworků může snadno vzniknout konflikt mezi závislostmi.

Další nevýhoda je dostupnost informačních zdrojů a ukázkových aplikací ve frameworku scalajs-react. Vydavatel sice poskytuje stručnou dokumentaci a ukázkové příklady, ale vše je velmi jednoduché a k reálně použitelné aplikaci to má daleko.

Závěr

Cílem práce bylo ukázat, jak lze vytvářet rozsáhlé podnikové aplikace s použitím funkcionálních principů za účelem zlepšení a zvýšení kvality implementace pomocí funkcionálních principů.

Bylo ukázáno jak pomocí funkcionálních principů a možností jazyka Scala snížit chybovost, zlepšit testovanost aplikace díky referenční transparentnosti funkcí, jak umožnit škálovatelnost a přepoužitelnost kódu. Vše bylo popsáno teoreticky v rešeržní části a ukázáno na jednoduché aplikaci v praktické části práce.

Použitím principů byla vytvořena webová aplikace, která má nízkou dobu odezvy při uživatelské akci díky asynchronní a neblokující komunikaci. Architektura serverové části je řešena návrhovým vzorem free monád, který zvyšuje SoC aplikace a umožňuje snadnou rozšiřitelnost aplikace díky vytvořenému DSL modulu.

Persistentní vrstva byla vytvořena pomocí event-sourcingu a CQRS. Event-sourcing umožňuje návrat stavu aplikace do libovolného okamžiku v historii, takže je jednoduché obnovit data do konzistentního stavu, pokud nastane chyba. CQRS umožňuje oddělit read model, který je optimalizovaný pro dotazování a obsahuje pouze užitečná data pro určitý kontext.

Serverová aplikace byla psána tak, aby bylo možné horizontální škálování. V případě potřeby se může spustit na více serverech, je tedy možné jednoduše pracovat s přetížením systému.

Klientská část je tvořena jednoduchou reaktivní webovou aplikací. Veškerá komunikace a logika je vykonávána pomocí asynchronních a neblokujících callbacků, takže hlavní vlákno, které obsluhuje uživatelskou interakci, není těmito dlouhotrvajícími operacemi ovlivněno.

Výsledek práce by mohl zvednout povědomí o funkcionálním paradigmatu. Ukázat možnosti, jak lze psát i rozsáhlý a udržovatelný funkcionální kód v týmu vývojářů. Obecné funkcionální principy, jako je minimalizace mutabilního stavu a referenční transparence, můžou být použity i v objektovém programování, čímž lze navýšit kvalitu aplikace.

Práce by mohla do budoucna posloužit jako teoretický podklad pro další praktické rozpracování. Zajímavé a přínosné by mohlo být např. aplikování a praktické porovnání funkcionálního programování a objektového programování na tutéž aplikaci. Hlubší prozkoumání persistence pomocí event-sourcing a CQRS na clusteru s využitím NoSQL databází by rovněž mohlo být dále rozpracováno. Přínosné by bylo vytvoření rozsáhlé aplikace a na ní prozkoumat možnosti reaktivní komunikace pomocí streamů. Zajímavé by také mohlo být využití streamů v rámci gRPC protokolu.

Literatura

[1] 21 Shocking Project Management Statistics That Cost Business Owners Millions Each Year. *Mavenlink Blog [online]*, 2017, [cit. 2017-11-12]. Available from: `http://blog.mavenlink.com/21-shocking-project-management-statistics-that-explain-why-projects-continue-to-fail`

[2] Ghosh, D. *Functional and Reactive Domain Modeling*. NY: Manning Publications Co, 2016, ISBN 9781617292248.

[3] Eric, E. *Domain-driven design: tackling complexity in the heart of software*. New Jersey, United States: Pearson Education (US), 2003, ISBN 0321125215.

[4] Software design pattern. *Wikipedia [online]*, 2018, [cit. 2018-7-3]. Available from: `https://doc.akka.io/docs/akka/2.5.4/scala/stream/stream-introduction.html`

[5] Monoids, semigroups, and friends. *ploeh blog [online]*, 2018, [cit. 2018-9-3]. Available from: `http://blog.ploeh.dk/2017/10/05/monoids-semigroups-and-friends/`

[6] Monoids. *ploeh blog [online]*, 2018, [cit. 2018-9-3]. Available from: `http://blog.ploeh.dk/2017/10/06/monoids/`

[7] Semigroups. *ploeh blog [online]*, 2018, [cit. 2018-9-3]. Available from: `http://blog.ploeh.dk/2017/11/27/semigroups/`

[8] Demystifying the Monad in Scala. *Medium [online]*, 2015, [cit. 2018-27-03]. Available from: `https://medium.com/@sinisalouc/demystifying-the-monad-in-scala-cc716bb6f534`

[9] The free monad in (almost) real life. *BEYOND THE LINES [online]*, 2018, [cit. 2018-11-3]. Available from: `https:`

//www.beyondthelines.net/programming/the-free-monad-in-almost-real-life/

[10] The Reactive Manifesto. *The Reactive Manifesto [online]*, 2018, [cit. 2018-15-4]. Available from: https://www.reactivemanifesto.org

[11] Streams. *Akka Documentation [online]*, 2017, [cit. 2017-11-12]. Available from: https://doc.akka.io/docs/akka/2.5.4/scala/stream/stream-introduction.html

[12] Persistence. *Akka Documentation [online]*, 2017, [cit. 2017-11-12]. Available from: https://doc.akka.io/docs/akka/2.5.4/scala/persistence.html

Příloha **A**

Seznam použitých zkratek

IT Informační Technologie.

DDD Domain-Driven Design.

ADT Algebraic Data Type.

AST Abstract Syntax Tree.

DSL Domain Specific Language.

SoC Separation of Concerns.

REST Representational State Transfer.

AJAX Asynchronous JavaScript And XML.

CRUD Create, Read, Update, Delete.

CRQS Command Query Responsibility Seqregation.

JVM Java Virtual Machine.

gRPC Google's Remote Procedure Calls.

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `exe`
  - `server` ... adresář se spustitelnou formou implementace serverové části
  - `client` ... adresář se spustitelnou formou implementace klientské části
- `src`
  - `server` ..... zdrojové kódy implementace serverové části
  - `client` ..... zdrojové kódy implementace klientské části
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF