FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Redesign backendu služby sdílení vozidel Uniqway
Student:	Petr Kostelník
Vedoucí:	Ing. Filip Ravas
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Úkolem bakalářské práce je zavést změny do backendové části systému carsharingové služby Uniqway, které pomocí rozdělení jednotlivých logických částí přispějí k lepší produktivitě nových členů projektu.

* Na základě předcházejících závěrečných prací věnujících se změně architektury backendu Uniqway na mikroslužby identifikujte části systému vhodné k rozdělení.
* Navrhněte způsob rozdělení jednotlivých částí, které nevyžadují komplexní změny v infrastruktuře a nezvýší celkovou složitost backendu.
* Navrhnuté změny implementujte s důrazem na zachování existující funkcionality systému.
* Vhodně zvoleným způsobem vyhodnoťte účinnost zavedených změn.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 22. ledna 2022 v Praze.

Bakalářská práce

REDESIGN BACKEDNU SLUŽBY SDÍLENÍ VOZIDEL UNIQWAY

Petr Kostelník

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Filip Ravas
12. května 2022

Obsah

Poděkování vi

Prohlášení vii

Abstrakt viii

Seznam zkratek ix

1 Úvod 1

2 Cíl práce 3

3 Literární rešerše 5
3.1 Základní pojmy . . . 5
3.2 Monolitická architektura . . . 5
3.2.1 Softwarová architektura . . . 5
3.2.2 Definice . . . 6
3.2.3 Porovnání s architekturou mikroslužeb . . . 6
3.2.4 Vlastnosti . . . 6
3.3 Dekompozice . . . 7
3.4 Refactoring . . . 8
3.4.1 Definice . . . 8
3.4.2 Proč refaktorovat . . . 8
3.4.3 Proces . . . 8
3.4.4 Možné obtíže . . . 8
3.4.5 Doporučené postupy . . . 8
3.4.6 Technický dluh . . . 9
3.5 Návrhové vzory . . . 9
3.5.1 Definice . . . 9
3.5.2 Popis . . . 9
3.5.3 Dělení . . . 10

4 Analýza 11
4.1 Infastruktura Uniqway systému . . . 11
4.2 Serverová aplikace . . . 11
4.2.1 Balíčky . . . 12
4.2.2 Technologie . . . 13
4.3 Identifikace logických částí . . . 13
4.3.1 Obchod s odměnami . . . 14
4.4 Překážky pro rozdělení systému . . . 15
4.4.1 Velká provázanost . . . 15
4.4.2 Malá soudržnost . . . 15
4.4.3 Degradující modularizace . . . 16

5 Návrh **17**
5.1 Požadavky na návrh . . . 17
5.2 Řešení nízké soudržnosti . . . 18
5.2.1 Balíčková struktura . . . 18
5.3 Řešení velké provázanosti . . . 19
5.4 Řešení degradace modularizace . . . 20

6 Implementace **23**
6.1 Příprava . . . 23
6.1.1 Vytvoření balíčkové struktury . . . 23
6.1.2 Přesun tříd . . . 23
6.2 Vytvoření fasády modulu . . . 24
6.2.1 Vyhledání vazeb a přidání fasády . . . 24
6.2.2 Přidání metod do fasády . . . 24
6.2.3 Přesměrování volání na fasádu . . . 24
6.2.4 Odstranění původních služeb . . . 24
6.3 Vytvoření fasád komponent . . . 24
6.4 Propojení fasád komponent a modulu . . . 25
6.5 Vyhodnocení změn . . . 25

7 Závěr **27**

Obsah přiloženého média **31**

Seznam obrázků

4.1 Schéma systému Uniqway 12

5.1 Balíčková struktura komponenty 19
5.2 Souhrn navržených změn 21

Seznam tabulek

Seznam výpisů kódu

Chtěl bych poděkovat svému vedoucímu Ing. Filipu Ravasovi za jeho ochotu, připomínky a rady při tvorbě této bakalářské práce. Také děkuji své rodině a přítelkyni za veškerou podporu během mého celého studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při 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, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona a to na dobu určitou do skončení trvání ochrany dle Smlouvy.

Nakládání s předloženou prací se řídí Smlouvou o spolupráci uzavřenou v návaznosti na spolupráci mezi Českým vysokým učením technickým v Praze a společností ŠKODA AUTO a.s. a Smart City Lab s.r.o na výzkumném projektu „CarSharing pro vysokoškolské studenty", uveřejněné v registru smluv na adrese https://smlouvy.gov.cz/smlouva/5973503. Jsem vázán Smlouvou o zachování mlčenlivosti, že nezpřístupním třetí osobě důvěrné informace, které jsem při své práci na Projektu získal.

V Praze dne 12. května 2022 .....

Abstrakt

Práce se zabývá navržením změn v monolitické architektuře serverové aplikaci systému Uniqway, které ho rozdělí do jednotlivých logických částí a zlepší tak vysokou provázanost a malou strukturalizaci systému. V návrhu se při dekompozice systému využívá návrhových vzorů, především pak Fasády. Během implementace se pro zavedení změn využívá refaktoring. Zavedené změny zvýší čitelnost kódu a umožní novým členům lépe se v systému zorientovat.

Klíčová slova modularizace serverové aplikace, dekompozice systému, návrhové vzory, monolitická architektura, Uniqway, Java, redesign serverové aplikace

Abstract

The work deals with the proposed changes in the monolithic architecture of the server application of the Uniqway system, which divides it into individual logical parts and thus improves the high interconnection and low structuring of the system. For designing of the decomposition of the system are used design patterns, especially Facade. During implementation is used refactoring to implement the changes. The changes introduced will increase the readability of the code and allow new members to better orient themselves in the system.

Keywords server application modularization, system decomposition, design patterns, monolithic architecture, Uniqway, Java, server application redesign

Seznam zkratek

API	Application Programming Interface
CRUD	Create Read Update Delete
DAO	Data Access Object
DDD	Domain-Driven Design
DTO	Data Transfer Object
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
MVC	Model-View-Controller
ORM	Object-relation Mapping
REST	REpresentational State Transfer

Kapitola 1
Úvod

Sdílená ekonomika je v dnešní době odvětví s perspektivní budoucností. Sdílení zboží není pro lidstvo žádnou novinkou, nikdy předtím však nebylo možné ho uskutečnit tak efektivně a na dobré úrovni jako dnes. Podstatný faktor v tom hraje právě vyspělá technologie, která nám poskytuje řadu možností, jak sdílenou ekonomiku vylepšit a přispět tak k růstu tohoto podnikání.

Perspektivnost se zde ukrývá mimo jiné též v širokém záběru, který nám sdílená ekonomika nabízí. Je totiž možné nabízet a poptávat téměř cokoliv, od ubytování, přes finance až právě po sdílení vozidel, tzv. carsharingu. Ve sdílení vozidel spatřuje řada lidí výhodu větší ekologičnosti a finanční výhodnosti, která pak zaujme převážně škálu mladých lidí, jenž nedisponující vysokými finančními prostředky. Často se pak jedná konkrétně o studenty. Ti nemívají problém s moderními technologii, a naopak je pro ně přitažlivý způsob digitálních transakcí a využití skrze moderní aplikace, které lze stáhnout do mobilních telefonů. Sdílení lze pak sjednat jednoduše, v co nejkratším čase a bez nutnosti osobního kontaktu.

Další výhodou může být jednoduchost a bezstarostnost využití půjčeného vozidla. Není nutné se starat o zákonné či havarijní pojištění, dálniční známku, servisní prohlídku nebo výměnu zimních či letních pneumatik. Pro řidiče, kteří auto využijí jen příležitostně, by koupě byla zbytečná a nevýhodná. Carsharing dále může pomoci zredukovat velké množství automobilů ve městech, společně s problémem nedostupnosti parkovacích míst.

Mezi služby, které nabízí sdílení aut patří právě i Uniqway, [1] vytvořená studenty z ČVUT v Praze, ČZU v Praze a VŠE v Praze pro studenty a zaměstnance všech vysokých škol. Jedná se o první carsharing provozovaný studenty, čímž je tento projekt unikátní.

Nejdůležitější částí systému Uniqway je serverová aplikace starající se o celou jeho logiku. Tato část je postavena na monolitické architektuře, což znamená, že veškerý kód se vyskytuje v jedné jediné kódové základně. Se zvětšující se velikostí této aplikace se zvyšuje komplexita systému, čímž se stává složitější přidávání nových funkcí a je především náročné pro nově příchozí vývojáře se v tomto systému zorientovat.

Na rozdělení systému již pracovali Bc. Petr Prouza a Bc. Štěpán Severa v diplomových pracích [2][3] se společným cílem transformovat monolitickou architekturu na mikroslužby. Po dokončení jejich prací došlo k vyhodnocení, že přechod na tuto achritekturu není pro systém Uniqway vhodný. Důvody byly například složitost mikroslužeb pro zde pracující studenty, kteří s tím ve většině případů nemají žádné zkušenosti, či velikost rozsahu mikroslužeb, který by byl finančně i personálně náročný.

Proto bylo navrženo nové téma, které se bude ve stávající monolitické architektuře zabývat vhodným rozdělením jednotlivých logických částí serverové aplikace, které zpřehlední systém a přispěje tak k lepší produktivitě nových členů.

Toto téma jsem si zvolil, protože mě zaujal projekt pracující s vozidly v reálném světě. Nejednalo se tak pouze o program, jehož výsledek nelze spatřit v každodenním životě. Zároveň se

mi líbilo, že se jedná o studentský projekt, a tak komunikace s ostatními členy týmu je bližší a srozumitelnější.

Tato bakalářská práce se zabývá problematikou monolitické architektury serverové aplikace, konkrétněji její modularizací a strukturalizací, která zpřehlední systém pro nové členy týmu Uniqway. V teoretické části jsou představeny důležité pojmy jako je monolitická architektura, dekompozice, refaktoring a návrhové vzory. Dále se práce zabývá analýzou stávajícího řešení systému a poté v praktické části návrhem, jak serverovou aplikaci co nejvhodněji rozdělit a upravit bez nutnosti komplexních změn a ovlivnění stávající funkcionality. Na to pak navazuje implementační část, kde je popsáno zavedení navržených změn do systému, a nakonec vyhodnocení účinnosti těchto změn.

Kapitola 2
Cíl práce

Cílem této bakalářské práce je zavést změny do backendové části systému carsharingové služby Uniqway, které pomocí rozdělení jednotlivých logických částí přispějí k lepší produktivitě nových členů projektu.

- Na základě předcházejících závěrečných prací, věnujících se změně architektury backendové části služby Uniqway na mikroslužby, identifikovat části systému vhodné k rozdělení.
- Navrhnout způsob rozdělení jednotlivých částí, které nevyžadují komplexní změny v infrastruktuře a nezvýšení celkové složitosti backendu.
- Navrhnuté změny implementovat s důrazem na zachování existující funkcionality systému.
- Vhodně zvoleným způsobem vyhodnotit účinnost zavedených změn.

Kapitola 3
Literární rešerše

V této kapitole budou pospány všechny důležité pojmy a techniky, které pomohou k pochoponí problému, kterým se tato bakalářská práce zabývá a dále pomůžou při vytváření návrhu změn pro stávající serverou aplikaci systému Uniqway a následné implementaci.

3.1 Základní pojmy

V této podkapitole jsou vysvětleny pojmy, které jsou zmíněny v bakalářské práci, jak v rámci literární rešerše, tak analýzy, a přehled o nich slouží k lepšímu porozumění textu.

API Application programming interface, je množina příkazů, funkcí, protokolů a objektů, která poskytuje rozhraní skrze které umožňuje vzájemnou komunikaci různých systémů jakými jsou například server a mobilní aplikace. [4]

CRUD Create (vytvořit), Read (číst), Update (aktualizovat), and Delete (odstranit) (CRUD) jsou 4 základní funkce, které by modely měly minimálně být schopné provádět nad svými zdroji. Když se vytváří rozhraní API, je žádoucí, aby modely poskytovaly všechny tyto základní typy funkcí. [5]

HTTP HyperText Transfer Protocol je protokol určený k přenosu informací mezi síťovými zařízeními. [6]

MVC Model-View-Controller je návrhový vzor pro aplikace zahrnující tří vzájemně propojené části. MVC se skládá z částí Model, který se stará o data, View, který má na starosti vizualizaci těchto dat a controller, který obsahuje procesy obsluhující vstup přijatý z vnější a provádí odpovídající aktualizaci modelu a view. [7]

REST REpresentational State Transfer je architektonický styl pro poskytování standardů mezi počítačovými systémy na webu, které systémům usnadňuje vzájemnou komunikaci. Systémy kompatibilní s REST, jsou často nazývané RESTful systémy a charakterizují se bezestavostí a rozdělením zodpovědnosti mezi klienta a server. [8]

3.2 Monolitická architektura

3.2.1 Softwarová architektura

Softwarovou architekturu lze jednoduše definovat jako organizaci systému. Tato organizace zahrnuje všechny její komponenty, způsob jejich vzájemné interakce, prostředí, ve kterém fungují, a

principy používané k návrhu softwaru. V mnoha případech může také zahrnovat způsob vývoje softwaru v budoucnu. Softwarová architektura v softwarovém inženýrství pomáhá odhalit strukturu systému a zároveň skrýt některé detaily implementace. [9]

3.2.2 Definice

Pokud všechny funkce projektu existují v jediné kódové základně, pak tuto aplikaci nazýváme monolitickou aplikací. Aplikaci navrhujeme v různých vrstvách, jako je prezentace, služba a persistence, a poté nasazujeme tuto kódovou základnu jako jeden soubor. [10]

3.2.3 Porovnání s architekturou mikroslužeb

Architektonický styl mikroslužeb je přístup k vývoji jediné aplikace jako sady malých služeb, z nichž každá běží ve svém vlastním procesu a komunikuje s lehkými mechanismy, často s API zdroji HTTP. Tyto služby jsou postaveny na podnikových možnostech a lze je nezávisle nasadit pomocí plně automatizovaných zaváděcích strojů. Existuje naprosté minimum centralizované správy těchto služeb, které mohou být napsány v různých programovacích jazycích a používat různé technologie ukládání dat. [11]

3.2.4 Vlastnosti

Monolitická architektura má své výhody i nevýhody. Zde jsou vybrány vlastnosti, které jsou významné pro tuto architekturu, pomáhají k pochopení jejího výběru pro serverovou aplikaci, a zároveň nastiňují problémy, které přináší postupně rozrůstající se monolit, jimž můžou vývojáři v budoucnu čelit nebo již čelí.

3.2.4.1 Výhody

Jednoduchost vývoje Monolitický přístup je standardním způsobem vytváření aplikací. Nejsou vyžadovány žádné další znalosti. Veškerý zdrojový kód je umístěn na jednom místě, tudíž ho lze rychleji pochopit. [12]

Jednoduchost ladění Proces ladění je jednoduchý, protože veškerý kód je umístěn na jednom místě. Lze snadno sledovat tok požadavku a najít problém. [12]

Jednoduchost při přijímání nových členů týmu Zdrojový kód je umístěn na jednom místě. Noví členové týmu mohou snadno odladit některé funkční toky a seznámit se s aplikací. [12]

Nízké náklady v raných fázích aplikace Všechen zdrojový kód je umístěn na jednom místě, zabalen v jediné jednotce k nasazení a nasazen. Nejsou tak zde žádné režijní náklady ani náklady na infrastrukturu. [12]

Z důvodu těchto výhod se monolitická architektura obvykle používá v raných fázích vývoje aplikací:

Hlavní funkcí aplikace je být zisková V důsledku toho je důležité rychle implementovat některá řešení návrhu pro ověření konceptu, aby se ověřila aplikace v reálném světě. Dále je také důležité přivést zákazníky do systému. Zlepšení lze uskutečnit v budoucnu. [12]

Nejasnost požadavků v raných fázích vývoje Je těžké vytvořit smysluplnou architekturu, pokud jsou požadavky nejasné. Skuteční zákazníci mohou definovat obchodní potřeby poté, co některé funkce již fungují. [12]

3.2.4.2 Nevýhody

Údržba Pokud je aplikace příliš rozsáhlá a složitá na to, aby ji bylo možné zcela pochopit, je náročné provádět změny rychle a správně. [13]

Velká provázanost kódu I přes jasnou strukturu služeb uvnitř aplikace, dochází velmi často postupně k větší provázanosti mezi moduly. Důsledkem toho je, že změna na jednom místě může mít neznámé účinky na neznámých místech. Takový kód se stává i náročnějším na porozumění, protože složité, propletené vztahy jsou obtížné na pochopení. [14]

Starší technologie Monolitické aplikace mají překážku v přijímání nových technologií. Vzhledem k tomu, že změny rámců nebo jazyků ovlivní celou aplikaci, je extrémně nákladné jak z hlediska času, tak financí takovou změnu provést. [13]

Nedostatek flexibility Kvůli monolitické architektuře jsme omezeni využitím technologií, které se používají uvnitř systému. Nelze tedy použít jiné nástroje pro vyřešení daného problému, i když jsou pro to optimálnější. [12]

Problémy s nasazením Sebemenší změna vyžaduje přemístění celého monolitu. [12]

V souhrnu je monolitická architektura optimální pro malé aplikace z důvodu rychlého vývoje, jednoduchosti testování, ladění a nákladů. Když se ale systém rozroste, můžou se objevit nežádoucí překážky. Některé z nich lze vyřešit refaktoringem. Tento proces bude popsán níže. [12]

3.3 Dekompozice

Jak v případě transformace monolitické architektury do architektury mikroslužeb, tak i u modularizace je nejprve potřeba si určit, jakým způsobem a na jaké části systém co nejlépe rozdělit. Mezi hlavní vlastnosti, které mají výsledné jednotlivé části, patří [15]:

Vysoká soudružnost Každá část má na starosti malou sadu úzce souvisejících funkcí. [15]

Společné uzavření Věci, které se mění společně, mají být zabaleny dohromady, aby bylo zajištěno, že každá změna ovlivní pouze jeden modul. [15]

Systém lze dělit dle:

Sloves Každá část aplikace zodpovídá za činnost, kterou lze popsat slovesem. Například rezervovat vozidlo. [16]

Podstatných jmen Každá část aplikace zodpovídá za činnost, kterou lze popsat podstatným jménem. Například slovem rezervace. [16]

Podnikových schopností Definují části odpovídající podnikovým možnostem. To je něco, co podnik dělá, aby generoval hodnotu. Například za vytvoření rezervace odpovídá třída Reservation. [15]

Subdomén Definuje části odpovídající subdoménám DDD. DDD označuje problémový prostor aplikace – podnikání – jako doménu. Doména se skládá z více subdomén. Každá subdoména odpovídá jiné části podnikání. [17]

3.4 Refactoring

3.4.1 Definice

Refaktoring lze dle [18] definovat dvěma způsoby v závislosti na kontextu zda se jedná o podstatné jméno nebo sloveso.

▶ **Definice 3.1.** *(podstatné jméno) Změna provedená ve vnitřní struktuře softwaru za účelem jeho snazšího pochopení a levnější úpravy beze změny jeho pozorovatelného chování.*

▶ **Definice 3.2.** *(sloveso) restrukturalizovat software za použití řady refaktoringů beze změny jeho pozorovatelného chování.*

3.4.2 Proč refaktorovat

Refaktoring zlepšuje kód tím, že ho dle [19] dělá:

- Efektivnějším řešením závislostí a složitostí.
- Lépe udržitelným nebo opakovaně použitelným díky zvýšené efektivitě a čitelnosti.
- Čistším, takže je snadněji čitelný a pochopitelný.
- Snadněji opravitelným pro vývojáře a snazší na nalezení chyb nebo zranitelnosti v kódu.
- Lépe upravitelným. Úprava kódu se provádí beze změny funkcí samotného programu. Mnoho základních editačních prostředí podporuje jednoduché refaktoringy, jako je přejmenování funkce nebo proměnné v celé základně kódu.

3.4.3 Proces

Proces refaktorování obsahuje mnoho malých změn ve zdrojovém kódu programu. Jedním z přístupů k refaktoringu je například zlepšit strukturu zdrojového kódu v jednom bodě a poté systematicky rozšířit tytéž změny na všechny použitelné odkazy v rámci programu. Myšlenkový proces spočívá v tom, že všechny malé změny v souboru kódu zachovávající chování mají kumulativní účinek. Tyto změny udržují původní chování softwaru. [19]

3.4.4 Možné obtíže

S procesem jsou někdy spojené i obtíže mezi které patří [19] dle :

- Proces zabere více času, pokud je vývojový tým ve spěchu a refaktoring není plánovaný.
- Bez jasných cílů může refaktoring vést ke zpožděním a práci navíc.
- Refaktoring nemůže řešit softwarové chyby sám o sobě, protože je vytvořen za účelem vyčištění kódu a snížení jeho složitosti.

3.4.5 Doporučené postupy

Mezi osvědčené postupy pro refaktoring dle [19] patří:

Nejprve refaktoring Vývojáři by měli refaktorovat před přidáním aktualizací nebo nových funkcí do stávajícího kódu, aby se snížil technický dluh.

Refaktoring v malých krocích To poskytuje vývojářům zpětnou vazbu v rané fázi procesu, aby mohli najít možné chyby a také zahrnout obchodní požadavky.

Stanovení jasných cílů Vývojáři by měli určit rozsah a cíle projektu na začátku procesu refaktoringu kódu. To pomáhá vyhnout se zpožděním a práci navíc, protože refaktoring má být formou úklidu, nikoli příležitostí ke změně funkcí nebo vlastností.

Časté testování To pomáhá zajistit, aby refaktorované změny nepřinášely nové chyby.

Automatizování Automatizační nástroje usnadňují a urychlují refaktoring, čímž zvyšují efektivitu.

Oprava závad samostatně Refaktoring není určen k řešení softwarových nedostatků. Odstraňování problémů a ladění by se mělo provádět samostatně.

Pochopení kódu Prohlédnutí kódu je důležité pro porozumění jeho procesům, metodám, objektům, proměnným a dalším prvkům.

Refaktoring, oprava a pravidelná aktualizace Refaktoring generuje nejvyšší návratnost investic, protože může vyřešit významný problém, aniž by zabral příliš mnoho času a úsilí.

3.4.6 Technický dluh

„ *Technický dluh je metaforický koncept, který popisuje jeden z největších problémů při vývoji softwaru.*“ Nastává nejčastěji v momentě, kdy jsou programátoři pod časovým tlakem a je upřednostněno rychlejší vytvoření funkčnosti, která je požadována, před kvalitou a správnou strukturou kódu. Takto nekvalitní kód s sebou pak přináší řadu komplikací, mezi které patří údržba a rozšiřování o přidané funkce. Úpravy se mohou prodražit a je bráněno inovacím. To všechno může v nejhorším případě vést až k úplnému zastavení vývoje projektu. Platí zde stejné pravidlo jako u jiných dluhů, že pokud není technický dluh splácen průběžně, jeho nahromadění se projeví. [20]

3.5 Návrhové vzory

3.5.1 Definice

V softwarovém inženýrství je návrhový vzor obecným opakovatelným řešením běžně se vyskytujícího problému v návrhu softwaru. Návrhový vzor není hotový návrh, který lze převést přímo do kódu. Je to popis nebo šablona pro řešení problému, kterou je možné použít v mnoha různých situacích. [21]

3.5.2 Popis

Návrhové vzory mohou urychlit proces vývoje tím, že poskytnou otestovaná a ověřená vývojová paradigmata. Efektivní návrh softwaru vyžaduje zvážení problémů, které se mohou projevit až později v implementaci. Opětovné použití návrhových vzorů pomáhá předcházet jemným problémům, které mohou způsobit velké problémy, a zlepšuje čitelnost kódu pro kodéry a architekty obeznámené se vzory. [21]

Lidé často chápou, jak aplikovat určité techniky návrhu softwaru na určité problémy. Tyto techniky je obtížné aplikovat na širší škálu problémů. Návrhové vzory poskytují obecná řešení zdokumentovaná ve formátu, který nevyžaduje specifika spojená s konkrétním problémem. [21]

Vzory navíc umožňují vývojářům komunikovat pomocí dobře známých a srozumitelných názvů pro softwarové interakce. Běžné návrhové vzory lze postupem času vylepšovat, díky čemuž jsou robustnější než návrhy pro konkrétní případy. [21]

3.5.3 Dělení

Návrhové vzory dělíme do 3 kategorií podle funkcionalit, na které se zaměřují. Tyto kategorie jsou dále obecně popsány.[21]

Kreativní vzory Všechny tyto návrhové vzory jsou o instanci třídy. Tento vzor lze dále rozdělit na vzory vytváření tříd a vzory vytváření objektů. Zatímco vzory pro vytváření tříd využívají v procesu vytváření instancí efektivně dědičnost, vzory pro vytváření objektů využívají k dokončení práce efektivně delegování. [21]

Strukturální vzory Všechny tyto návrhové vzory jsou o složení třídy a objektu. Vzory při vytváření strukturních tříd využívají ke skládání rozhraní dědičnost. Strukturální objekty-vzory definují způsoby, jak skládat objekty pro získání nových funkcí. [21]

Vzorce chování Všechny tyto návrhové vzory jsou o komunikaci objektů třídy. Vzorce chování jsou vzorce, které se nejvíce konkrétně týkají komunikace mezi objekty. [21]

Kapitola 4
Analýza

Tato kapitola se nejprve věnuje popisu celkové infastruktury projektu Uniqway, z jakých komponent se projekt skládá a jejich význam, který zde zastávají. Dále se detailněji zaměří na serverovou část, ve které bude probíhat zbytek analýzy od technologií, přes rozdělení kódu do balíčku až po problémy, které se zde vyskytují.

4.1 Infastruktura Uniqway systému

Systém uniqway je tvořen několika komponentami, které spolu navzájem komunikují a vytváří tak plnohodnotnou fungující službu pro sdílení vozidel. Jádrem tohoho systému je serverová aplikace spolu s relační databází. Tato část bude popsána více v samostatné kapitole.

První z několika forntendových aplikací je správcovská webová aplikace, která stejně jako všechny později zmíněné komponenty, kominukuje se serverem. Tato webová aplikace slouží pro správce, kterým umožňuje pracovat s uloženými daty a spravovat vše od uživatelů, přes obchod s odměnami, až po vozový park.

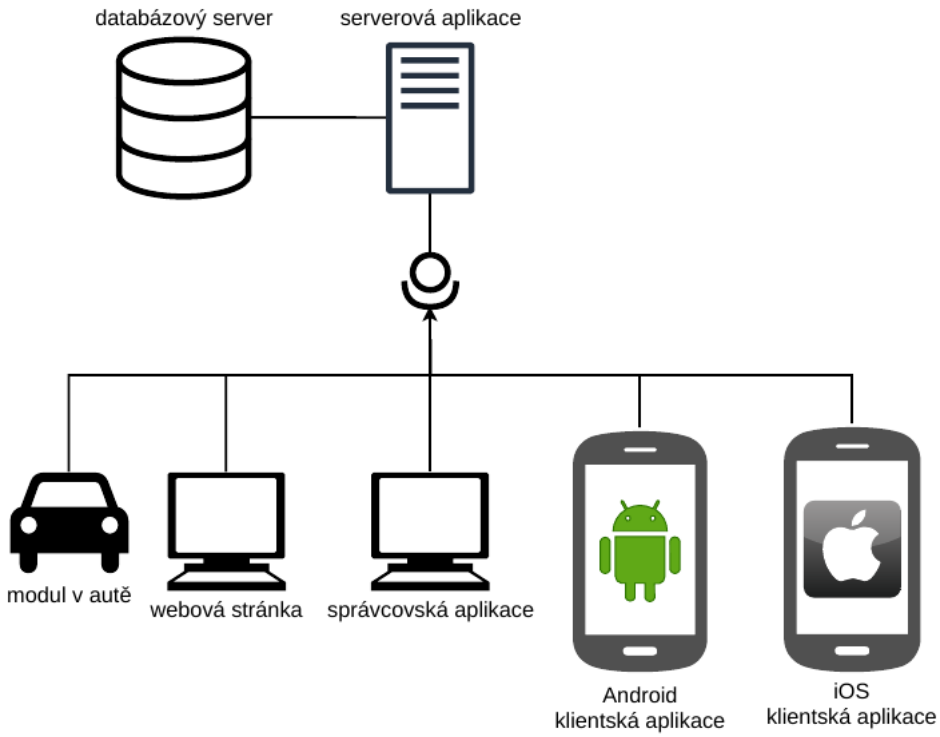
Naproti správcovské je zde uživatelská webová aplikace. Přes ní se mohou noví uživatelé zaregistrovat, prohlédnout si automobily, spočítat orientační cenu rezervace a zjistit další informace o službě a její nabídce.

Dalšími komponentami jsou klientské mobilní aplikace pro operační systémy Android a iOS. Opět se jedná o komponenty pro uživatele, kteří si mohou skrze tyto aplikace najít dostupná vozidla, vytvořit si na ně rezervaci, prohlédnout si, kde se nachází vyznačené parkovací zóny, odemknout a zamknout auto, nebo nakoupit Uniqway produkty v obchodě s odměnami.

Poslední součást infrastruktury je hardwarový modul v automobilu, který umožňuje odemknutí a zamknutí výše zmíněnými mobilními aplikacemi a zároveň posílá serveru důležitá data o stavu vozidla.

4.2 Serverová aplikace

Jak lze vidět na obrázku 4.1, serverová aplikace je středobodem systému. Zajišťuje totiž komunikaci mezi všemi komponentami projektu využitím REST API. Tato komponenta je postavena na monolitické architektuře a návrhovém vzoru MVC. Kód je členěn do balíčků dle předepsané formy frameworku Play. Vybrané balíčky a jejich obsah jsou popsány níže, stejně jako technologie využívané při vývoji.



Obrázek 4.1 Schéma systému Uniqway

[2]

4.2.1 Balíčky

actions Obsahuje třídy umožňující provedení akce nad vybranými požadavky ještě před jejich zpracováním, například autentizaci modulu či uživatele.

controllers Obsahuje třídy starající se o zpracování HTTP požadavků. Jejich metody jsou použity v API rozhraní.

dao Obsahuje třídy starající se o přístup k datům z databáze systému.

[mails] Obsahuje třídy starající se o vlastnosti emailů.

models Obsahuje modelové třídy, které představují entity v databázi. V tomto balíčku se nachází ještě podstatný balíček *dto*. Ten obsahuje třídy sloužící pro přenos dat objektů.

services Obsahuje třídy starající se o obchodní logiku.

utils Obsahuje pomocné třídy například pro formátování, filtrování nebo vyhazování výjimek.

views Obsahuje třídy definující HTML dokumenty.

4.2.1.1 Další dělení

Třídy v balíčkách se dále člení do balíčků dle oprávnění volajícího. Toto dělení se samozřejmě vyskytuje pouze tam, kde je ho třeba a v míře jaké je třeba, nikoliv všude a kompletně.

admin Třídy může volat pouze admin.

client Třídy může volat autentizovaný klient

cron Třídy automaticky volá cron.

demo Třídy může volat kdokoliv.

Mezi balíčky, kde je toto dělení nejvíce vidět a je významné, patří *controllers*, *dto* a *services*. *Dto* a *services* dále obsahují ještě balíček *global*. Ten ve stávajícím systému obsahuje třídy, které slouží jako nadtřída pro stejnojmenné třídy z balíčků zmíněných výše a shromažďuje jejich společné vlastnosti. V některých případech se třídy z *global* balíčku používají jako univerzální, většinou na místech, kde dělení dle oprávnění nemá význam.

Další dělení, které se zde částečně vyskytuje je dle funkcionality. Mezi takové balíčky patří například *car*, *user* nebo *shop*.

V systému není dělení popsané v této podkapitole striktně dodržováno a tak se v něm nachází na mnoha místech nesrovnalosti, které způsobují značnou nepřehlednost.

4.2.2 Technologie

Serverová aplikace je psána v jazyce Java a frameworku Play. Databáze se používá relační PostgreSQL a pro objektově relační mapování (ORM) pak Ebean ORM. Pro verzování systému se využívá společný Git repozitář, který také obsahuje část dokumentace systému.

Jak již bylo zmíněno, serverová aplikace komunikuje s ostatními komponentami systému Uniqway přes REST API. Dokumentace tohoto rozhraní se nachází v portálu Swagger a slouží pro frontendové vývojáře. Tato dokumentace ale nebyla tvořena od začátku projektu, začala se tvořit až v průběhu vývoje, a důsledkem toho se zde nenachází všechny zdroje.

K testování se potom využívá frameworku JUnit, který slouží pro psaní jednotkových testů. Dále jsou zde vytvořeny i API testy.

4.3 Identifikace logických částí

V této podkapitole jsou obecně popsány nejzásadnější funkcionality jednotlivých logických částí serverové aplikace, které byly identifikovány jako vhodný kandidát pro oddělní do modulu pomocí kombinací způsobů dekompozice popsaných v kapitole 3.3. U každého logického celku se pak nachází klíčová slova. Ta obsahují slova či slovní spojení, která se pojí s názvy tříd a jejich významu v systému. Tato slova pomáhají při návrhu rozdělení kódu v modulu do jednotlivých komponent jako tomu je pak v samostatné podkapitole věnující se obchodu s odměnami, na kterém se poté pracuje v praktické části v rámci implementace. Dále také pomáhají při vyhledávání úzce spjatých tříd a definovat lépe hranice modulů a jednotlivých komponent.

Auto Tato logická část je jedna z obsáhlejších. Zahrnuje veškeré funkce spojené s automobilem. Patří sem třídy zodpovědné za komunikaci s vozidlem přes modul, který tvoří jednu z hlavních komponent systému Uniqway, odemykání a zamykání, sbírání informací o stavu jako je třeba množství benzínu v nádrži, ke kterému se pojí jesště účtenky za tankování. Dále sem patří třídy popisující auto jako takové, tedy o jaký typ se jedná, vybavení vozu nebo typ motoru.

- Klíčová slova: auto, benzín, účtenka

Parkovací zóna Tato logická část se stará parkovací zóny, což jsou místa vyhrazená k zaparkování vozů. Mimo tyto zóny uživatel tedy nemůže jízdu ukončit.

- Klíčová slova: parkování, parkovací zóna

Cena Tato logická část se zabývá funkcionalitou od vytvoření ceníků až po samotný výpočet ceny rezervace nebo jen jejich dílčích částí. Uživatelé si mohou nechat vypočítat výslednou cenu za využití služeb.

- Klíčová slova: cena, ceník

Platba Tato logická část má na starosti verifikaci platebních karet a platby. Zde se jedná především o bezpečnost než rozsah.

- Klíčová slova: platba, verifikace, měna

Rezervace Tato logická část se nachází ve středu všeho dění, stejně jako serverová aplikace v systému Uniqway. Je využívána pro rezervaci vozidla uživatelem a jeho následné zapůjčení, včetně jízdy. Je zde tedy propojení mnoha entit, které je třeba korigovat. Také je zde funkcionalita starající se o data předchozích dnů rezervace nebo stavy rezervace pro analytické účely.

- Klíčová slova: rezervace, jízda, předchozí den, analýza statusu

Uživatel Tato logická část shromažďuje a spravuje informace o uživateli podobně jako první zmíněná část, která se stará o vozidlo. V tomto případě se jedná o identifikační informace, licence opravňující řízení vozidla, uživatelské role, profily, třídy týkající se zákazníka, který může nakupovat v obchodě s odměnami nebo třídy starající se o vytvořené skupiny uživatelů.

- Klíčová slova: uživatel, licence, role, skupina uživatelů

Autorizace a autentizace Tato logická část má na starosti validaci, přihlašování uživatelů do aplikací, což obnáší ověřování přihlašovacích údajů, pak také autentizace uživatelů a autorizaci přístupů ke zdrojům.

- Klíčová slova: autorizace, autentizace, validace

Oznámení Tato logická část slouží k posílání emailů, například poslání faktury, a tzv. push notifikací se kterými souvisí funkcionalita pro správu verzí aplikací. Podporuje i možnost ověření stavu tokenů.

- Klíčová slova: notifikace, email, verze, token

Faktura Tato logická část se stará o vytváření faktur po ukončení rezervace nebo objednávky v obchodu s odměnami.

- Klíčová slova: faktura

Obchod s odměnami Tato logická část se také řadí mezi rozsáhlejší, protože v sobě obsahuje funkcionalitu potřebnou pro celý provoz obchodu. Uživatel si zde může pořídit tématické předměty za malý doplatek a nasbírané body, které se získávají za využívání služby Uniqway. Zakoupený předmět si poté vyzvedne na výdejním místě.

- Klíčová slova: odměna, produkt, objednávka, sklad

4.3.1 Obchod s odměnami

Obchod tvořící modul lze pomocí klíčových slov rozdělit na 4 komponenty, které dohromady tvoří celou funkcionalitu obchodu. Základ každé komponenty tvoří hlavní entita, podle které je daná komponenta následně pojmenována a dále se v nich vyskytují všechny úzce spjaté třídy. Po rozdělení obchodu a zvolení dalších klíčových slov, se už jednotlivé komponenty nedají dále dělit, aniž by nevznikaly části o jednotkách tříd. Zároveň máme kód dobře rozdělený na části s vysokou soudržností.

- Klíčová slova: odměna, produkt, objednávka, sklad

Odměna Má na starosti bodové odměny a aktivity, za které lze body získat. Každá aktivita je jinak odměňována a samotné aktivity se liší jak svým obsahem, tak časovým obdobím, po které platí. Příkladem takové aktivity může být počet ujetých kilometrů nebo už jen samotné vytvoření objednávky.

- Klíčová slova: odměna, aktivita

Produkt Má na starosti všechny produkty nabízené v obchodu. K produktům jsou přiřazeny kategorie, obrázek, aby si mohl zákazník produkt prohlédnout ve své aplikaci, a dále vlastnosti a parametry, které přidávají další informativní popis.

- Klíčová slova: produkt, obrázek, kategorie, atribut, vlastnost

Sklad Má na starosti skladování. Stará se o evidenci produktů na výdejních místech a počet jaký jich byl přijat či odebrán.

- Klíčová slova: sklad

Objednávka Má na starosti tvorbu objednávky zákazníkem a její správu jako je třeba měnící se stav objednávky od vytvoření, přes zpracovávání až po dokončení. Dále komponenta eviduje výdejní místa, kde si zákazník může vyzvednout svůj produkt.

- Klíčová slova: objednávka, výdejní místo

4.4 Překážky pro rozdělení systému

Většina překážek, které byly nalezeny, se dají označit za důsledek rychlého vývoje monolitického systému bez pravidelného refaktoringu, který by napomohl kód udržet více čitelný a čistší. Docházelo tak k nárustu technického dluhu, který se postupem času hůře vymazává, způsobuje potíže při dalším vývoji a pro nové členy je obtížnější se v systému zorientovat. V této kapitole jsou popsány překážky, které se ve stávajícím systému nachází a je potřeba zohlednit jejich řešení při navrhování změn vedoucích k rozdělení kódové základny, aby byly výsledné moduly skutečně užitečné, a aby místo zpřehlednění a usnadnění vývoje nebylo docíleno opačného jevu.

4.4.1 Velká provázanost

Pokud se v systému vyskytuje velká provázanost, projevuje se to tím, že při změně provedené v jedné části, se může objevit nežádoucí efekt v části jiné. Je-li navíc systém větší, pak už může být obtížné toto nežádoucí chování předvídat. A při snaze opravit nově vzniklý problém, se může efekt změny projevit zase na jiném místě a tímto se lze dostat do stavu, kdy přidání změn bude velmi obtížné a v případě nezkušených vývojářů neznalých systému až nemožné. Je tedy žádoucí, aby provázanost tříd mezi moduly byla co nejmenší. V systému se aktuálně nachází velká provázanost především mezi třídami balíčku *models* a třídami balíčku *services*. V případě balíčku *models* je provázanost způsobena propojením databázových tabulek, které tyto třídy reprezentují. U tříd z balíčku *services* je provázanost způsobena proměnnými typu jiných konkrétních tříd a voláním jejich metod.

4.4.2 Malá soudržnost

Pokud se v systému vyskytuje malá soudržnost, pak to znamená, že část kódu, která obsahuje nějakou funkcionalitu, je rozprostřena do široce v kódové základně a tím je vývojáři zkomplikováno rozpoznat spolu související třídy a je nucen přeskakovat mezi různými částmi kódové

základny. Ve stávajícím systému se nachází velká soudržnost v rámci hlavních balíčků. Například balíček *services* obsahuje pouze třídy, které mají na starosti obchodní logiku a balíčky *models* zase reprezentaci entit v databázi. To může být ze začátku vývoje dobré, ale jen dokud se systém nerozroste do velikosti v jaké se nyní Uniqway nachází. V ten moment se stává taková soudržnost již na škodu, je obtížné se v tom zorientovat a při přidání funkce, musí vývojář skákat mezi balíčky a hledat konkrétní třídy. Je tedy na pováženou tuto soudržnost vytvořit mezi třídami, které spolu více úzce souvisí.

4.4.3 Degradující modularizace

Pokud se v systému vyskytuje degradace modularizace, pak to znamená, že není například dodržováno dělení tříd do odpovídajících modulů či balíčků, záleží, na jaké úrovni se třída nachází, nebo může být narušena nízká provázanost vytvářením nových vazeb na konkrétní třídy z jiných modulů. Ve stávajícím systému tato degradace začíná na úrovni dělení popsané v kapitole 4.2.1.1. Tedy dělení tříd do balíčků dle oprávnění volajícího a dle functionality. Problém je, že se zde nenachází žádná struktura nebo systém, který by tomu zabraňovaly nebo to alespoň omezovaly. Výsledkem tak jsou například nezařazené třídy do balíčků dle oprávnění volajícího. Samotné třídy neobsahují v názvu informaci o tomto zařazení, které se tak musí následně zjišťovat z širšího kontextu využití této třídy. Při větším počtu takto nezařazených tříd, může původní dělení do balíčků začít ztrácet smysl. Celé to samozřejmě komplikuje novým členům zorientovat se v systému.

Kapitola 5
Návrh

V této kapitole je popsán postup při vytváření návrhu změn v systému, které budou následně implementovány. Návrh se zde zaměřuje na problémy popsané v analýze, které úzce souvisí s rozdělením systému na jednotlivé logické části. Ačkoliv se analýza a implementace zaměřují na obchod s odměnami, je výsledný návrh více obecný, a to z důvodu, že navržené změny na této části musí být aplikovatelné v celém systému zcela stejně nebo jen s drobnými úpravami pro zachování jednotného systému.

5.1 Požadavky na návrh

Před samotným návrhem je vhodné definovat, jaké vlastnosti se od něho požadují a následně se zaměřit na problémy, jejichž vyřešení povede ke chtěnému výsledku. Mezi požadavky patří:

Vhodné rozdělení logických částí Při vhodném rozdělení není potřeba po přidání menších funkcionalit do systému zvažovat nové přerozdělení logický částí.

Nekomplexní změny Změny nevyžadují složitější postupy při implementaci, které by zasáhly do velké části systému.

Zachování funkcionality Funkcionalita systému zůstane zcela stejná jako před implementací navržených změn.

Nezvýšení celkové složitosti Změny nezvýší celkovou složitost serverové aplikace, které by tak zapříčinily opačný efekt pro nové členy týmu než je cílem této bakalářské práce.

Zvýšení produktivity nových členů Systém, který je lépe čitelný a snazší na pochopení, umožnňuje novým členům rychleji se zorientovat v systému a tím tak zvyšovat jejich produktivitu.

Tzv. high level přístup Návrh se nezaměřuje na detailnější implementaci tříd, ale na komunikaci mezi většími částmi systému, jímž jsou třídy součástí.

Výše vypsaných změn se bude návrh celou dobu držet. Především se musí dát pozor, aby návrh neobsahoval komplexní změny a nezvyšoval celkovou složitost, ale zároveň byl dostatečně efektivní na to, aby novým členům pomohl k lepšímu a rychlejšímu pochopení systému.

Překážky popsané v kapitole 4.4 byly úmyslně vybrány, protože poukazují přesně na problematiku systému, která novým vývojářům ztěžuje zaučení. Návrh se tedy věnuje jejich vyřešení, čímž se docílí, že systém je po zavedení změn odolnější vůči degradující modularizaci, má nové rozdělení splňující vysokou soudržnost a sníženou provázanost oproti stávajícímu systému. Postup řešení překážek je následující:

Nízká soudržnost Výsledkem vyřešení nízké soudržnosti je navržení nového rozdělení identifikovaných částí z kapitoly 4.3 do modulu s novou balíčkovou strukturou.

Velká provázanost Výsledkem vyřešení velké provázanosti je odstranění závislosti kódu mimo modul na konkrétních třídách, které se v modulu nachází.

Degradující modularizace Výsledkem vyřešení degradující modularizace je struktura v modulu, která pomáhá udržet zapouzdření jak celého modulu, tak jeho jednotlivých částí.

5.2 Řešení nízké soudržnosti

Řešením nízké soudržnosti a zároveň i cílem bakalářské práce je rozdělení vhodných jednotlivých logických částí systému do samostatných modulů, které budou zastřešovat společnou funckionalitu. Jak bylo naznačeno v analýze u tohoto problému, nevýhody stávajícího dělení již značně převyšují jeho výhody, kvůli velkému množství entit, a proto by bylo vhodné vytvořit vysokou soudržnost přerozdělením kódové základny do modulů spojující vhodné entity a funkcionalitu.

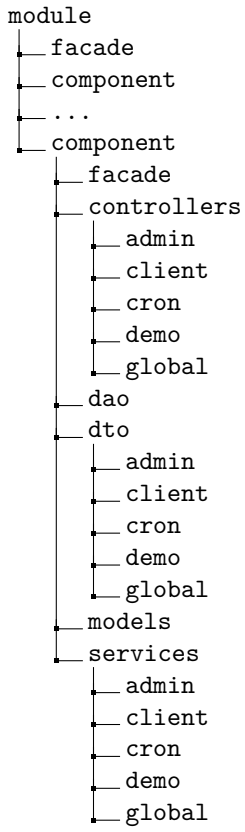
Obchod s odměnami tvoří přesně takovou část, a proto je dobrým kandidátem na oddělení do samostatného modulu. V analýze už je rovnou i pomyslné detailnější rozdělení vnitřku modulu na jednotlivé komponenty, do kterých jsou třídy patříci do daného modulu dále děleny. Modul v tomto případě tedy zastává funkcionalitu obchodu s odměnami a komponenty uvnitř, pak s vysokou soudržností dělí třídy na konkrétnější funkcionality, jako je komponenta starající se o sklad, další o odměny, pak o objednávku a poslední o produkty. Je potřeba zmínit, že část kódu, která se připravuje k oddělení je potřeba dobře pochopit, jak z hlediska kódu, tak procesů, aby byla skutečně rozdělením vytvořena vysoká soudržnost a její výhody.

5.2.1 Balíčková struktura

Nyní je zatím navrženo rozdělení vybraných tříd do modulu a jeho konkrétní komponenty, ale ještě je potřeba definovat proč a jak bude vypadat dělení uvnitř. Zde se nabízí jako nejlepší řešení zachovat stávající dělení. Význam a funkce přesouvaných tříd se nijak nemění a dané pojmenování balíčků se může označit za standart, kterému rozumí často i nezkušení vývojáři, kteří absolvovali určité univerzitní předměty.

Základní balíčková struktura komponenty se tedy skládá z balíčků určené pro třídy, které standartní entita používá. Struktura bude obsahovat balíčky *controllers*, *dao*, *dto*, *models*, *services*. Balíček *dto* zde nebude již schovaný v balíčku *models*, ale bude na stejné úrovni jako zbylé, jelikož to působí přehledněji a při takovémto množství balíčků si to lze dovolit, aniž by se vytvářela nepřehledná struktura. Zachováno zůstane i další dělení popsané v kapitole 4.2.1.1. Důvod je stejný jako u předchozího dělení, přesunem třídy se nijak nemění oprávnění ani jeho funkcionalita.

Existují třídy, které ani do jednoho dělení nepatří. Pro ty se samozřejmě přidá do struktury odpovídající balíček. Základní balíčková struktura je vyobrazena na obrázku 5.1 spolu s finálními úpravami celého návrhu, proto není totožná s tímto návrhem.



Obrázek 5.1 Balíčková struktura komponenty

5.3 Řešení velké provázanosti

S velkou provázaností je to ve velkém monolitickém systému bez komplexních změn náročnější, proto se jí nelze najednou zcela zbavit, ale lze ji částečně snížit ve vybraných částech. Jak bylo popsáno v kapitole 4.4.1, největší provázanost se nachází mezi třídami balíčku *models* a *services*. S modely se nedá v tomto případě nic dělat, jelikož by to znamenalo zásah do databáze. To by vyžadovalo komplexní změnu, které by musela předcházet podrobná analýza. Bez komplexních změn se ale dá snížit provázanost mezi třídami z balíčku *services*. Jejich provázanost tkví v proměnných, které jsou typu jiných tříd. Po rozdělení do jednotlivých modulů při řešení soudržnosti se provázanost nijak neovlivní, ale jsou metodiky známé jako návrhové vzory, které slouží k řešení daných problémů a jeden z těchto vzorů nabízí elegantní řešení, jak využít přesunuté třídy do modulu a schovat jejich detaily před zbylými třídami mimo modul. Tím řešením je návrhový vzor Fasáda.

Fasáda je třída, která poskytuje jednoduché rozhraní ke složitému subsystému, který obsahuje mnoho pohyblivých částí. Fasáda může poskytovat omezenou funkčnost ve srovnání s přímou prací se subsystémem. Zahrnuje však pouze ty funkce, na kterých klientům skutečně záleží. Mít fasádu je užitečné, když je potřeba integrovat svou aplikaci se sofistikovanou knihovnou, která má desítky funkcí, ale potřebujete jen malý kousek její funkčnosti. [22]

Návrhový vzor fasáda by se strukturou a chováním dal přirovnat k bráně, kterou vytvářeli pro vybrané mikroslužby v rámci diplomových prací [2][3]. Tato fasáda slouží jako vstupní bod do daného modulu, který posílá požadavky zvenčí dále na konkrétnější třídy nacházející

se v modulu, ale to už je zasílateli požadavku skryto. Kód nacházející se mimo modul tak obsahuje pouze odkaz na tuto fasádu a nikoliv na konkrétní třídy. Proto, aby fasáda mohla fungovat je potřeba v ní inicializovat všechny potřebné třídy, které se vyskytují mimo modul a vytvořit metody, které slouží k předávání požadavků z venčí temto třídám.

Navržení fasády není zcela přímočaré. Jedním z problémů je dělení balíčků dle oprávnění volajícího. Protože je systém v balíčku *services* dělený podle tohoto oprávnění na 5 různých balíčků, včetně tříd z balíčku *global*, umožnuje to ve všech mít stejně pojmenované třídy, a tak při inicializaci byť jen dvou z těchto tříd ve fasádě nastane konflikt ve jménech. Jedním z řešením je přejmenování tříd v celém systému přidáním názvu odpovídajícího bálíčku oprávnění, do kterých třídy patří, do jejich jména. To by fungovalo, ale ve výsledku se stane třída tvořící fasádu snadno přetížená a nepřehledná po vložení všech potřebných tříd. Proto se zvolí varianta, ve které je fasáda tvořena více třídami a každá z nich má na starosti jeden typ oprávnění. Z toho vyplývá, že celá fasáda dohromady může obsahovat až 5 tříd podle nutnosti. Každá z těchto tříd může volat naprosto odlišné metody, tudíž není možné vytvořit společný interface nebo abstraktní třídu, které by obsahovaly společné vlastnosti. Do nově navržené balíčkové struktury tak přibude ještě jeden balíček na úroveň komponent pojmenovaný facade obsahující třídy fasády.

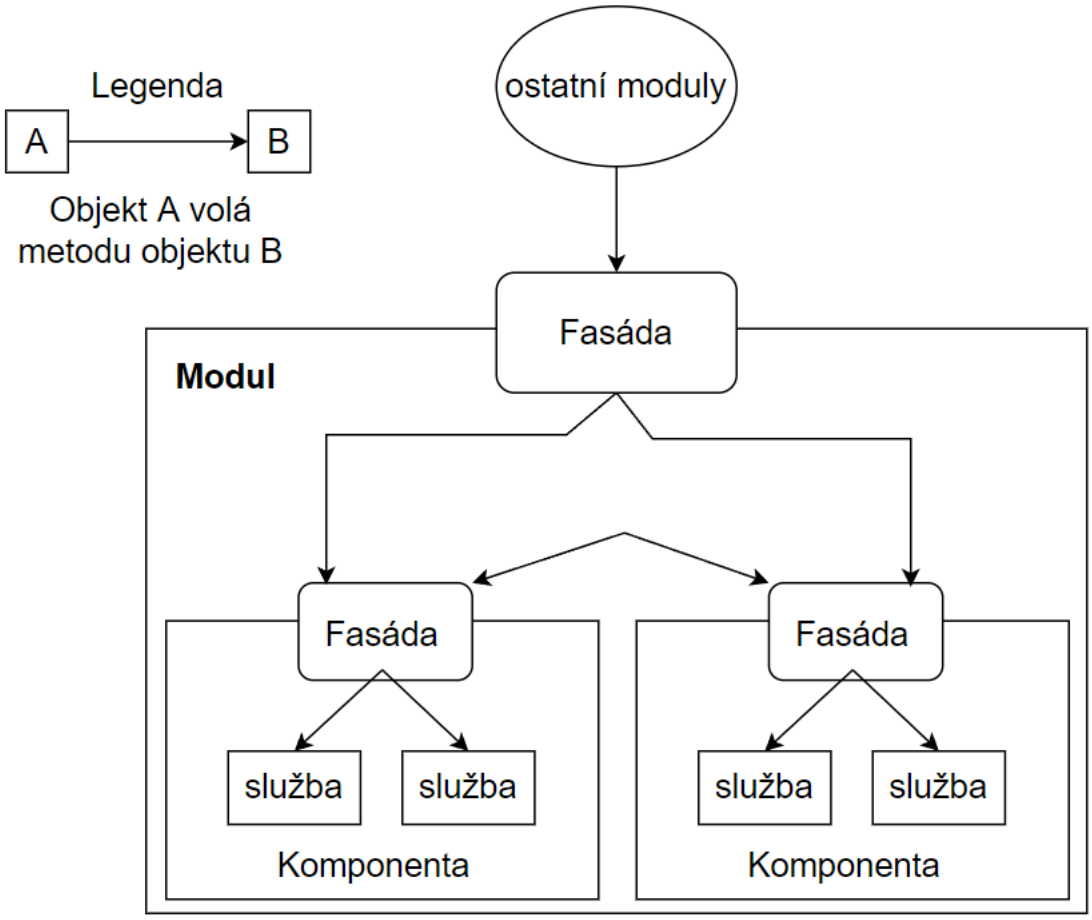
Dalším problémem je volání metod. Stejně jako třídy mají stejná jména, mají je i metody. Tentokrát už to není problém tříd jedné entity, ale naopak několika různých, protože služby jedné entity máme již oddělené jednotlivě do tříd fasády. V tomto případě je lepší využít detailnějšího pojmenování, které v okolním kódu přesněji popisuje, na co je metoda zaměřená. Jména metod ve fasádě jsou tvořena převzetím jména metody, na které dale odkazují a k tomu je přidán název entity se kterou pracují.

Po vyřešení potížích, které bránily fasádě ve využití, je tedy navrhnut modul, jeho vnitřní rozdělení a nyní i způsob jak snížit závislost okolních tříd na konkrétních třídách v modulu.

5.4 Řešení degradace modularizace

Řešením problému degradující modularizace, jak je zmíněno v 4.4.3, je nějaká struktura nebo systém, který by nutil vývojáře více dodržovat zvolená dělení a tím udržoval jednotný systém. Jde tedy o to, jak zavést změnu, která následně říká, která třída kam patří, a omezit využívání konkrétních tříd a vytváření tak další provázanosti.

Řešením se ukazuje být opět fasáda, ale tentokrát o úroveň níže v modulu, vytvořením fasády pro každou komponentu modulu zvlášť. Tím dojde k jejich zapečetění a získají se výhody jako je další snížení provázanosti tříd, protože i třídy uvnitř komponent budou navzájem skryty třídám zbylých komponent a soudržnost komponent více zastřešena pod svou vlastní fasádu. Grafické znázornění celého návhru se všemi navrženými změnamije zobrazeno na obrázku 5.2.



Obrázek 5.2 Souhrn navržených změn

Kapitola 6
Implementace

V této kapitole je popsán postup implementace, který zároveň slouží jako šablona pro dekompozici zbylého kódu, protože se tato bakalářská práce omezuje pouze na část kódu a to obchodu s odměnami. Proces, který se při tom využije se nazývá refaktroing, který je blíže popsaný v kapitole 3.4. Hlavním důvodem je, že není žádoucí upravovat vnější funkcionalitu, ale pouze vnitřní strukturu a na to se právě refaktoring užívá. Mezi postupy refaktoringu patří postupování po malých krocích a po každém dokončeném úseku kód otestovat, aby se ověřilo nenarušení stávající funkcionality. K tomu se využijí testy, které již pro serverovou aplikaci existují, tedy jednotkové a API testy.

6.1 Příprava

Než se začne samotnou tvorbou fasády, je potřeba si kód nejprve připravit. Všechny třídy, které budou zastřešeny do jednoho modulu, se přesunou mimo stávající balíčky a rozdělí se do navržené balíčkové struktury. Nejprve přesunou třídy související s jednou komponentou, nezáleží kterou, pak s další a takto postupujeme, dokud nejsou přesunuty všechny třídy.

6.1.1 Vytvoření balíčkové struktury

Prvním krokem implementace je vytvoření balíčku pro nový modul a dále do něj vytvoření navržené balíčkové struktury, tedy prázdných balíčků, do kterých budeme postupně přesouvat vybranné třídy, navržené pro komponenty .

6.1.2 Přesun tříd

Po vytvoření balíčků se začne s přesunem samotných tříd. Zde se využije výhod používaného IDE. Při přesunu třídy je potřeba upravit všechny soubory, které danou třídu obsahují a změnit její cestu v importu. Tuto práci automaticky vyřeší právě zmíněné IDE. Žádné třídy kromě kontrolerů, by neměly vyžadovat další změny v kódu.

Jak bylo dříve zmíněno, serverová aplikace využívá rozhraní API, které volá metody kontrolerů. Volaná metoda je zde identifikována cestou, která se přesunem tříd také změnila, ale na rozdíl od cest importů, není editorem automaticky změněna, a proto musí být tato úprava provedena ručně.

Po přesunu tříd a úpravě cest, je potřeba první otestování systému, že vše stále plně funkční. V úvodním refaktoringu docházelo především ke změně v API rozhraní, proto je potřeba pozornost

nejvíce zaměřit na API testy. Po úspěšném průchodů všech testů, se může pokračovat v přesunu dalších částí modulu.

6.2 Vytvoření fasády modulu

Po úvodní přípravě je na řadě fasáda modulu, která izoluje uvnitř nacházející se třídy od zbytku kódu. Prvním krokem je vytvoření tříd tvořící fasadu do příslušného balíčku.

6.2.1 Vyhledání vazeb a přidání fasády

Dalším krokem je vyhledání všech tříd, které mají vazbu na třídy v nově vytvářeném modulu. Využije se při tom pomocné funkce editoru, která umí veškeré hledané vazby nalézt. I zde, stejně jako ve zbylé implementaci, se doporučuje postupovat jednotlivě po komponentách.

Do těchto tříd se inicializují třídy fasády, které odpovídají balíčkům, ze kterých nahrazované služby pochází. Po vložení fasádových tříd do konstruktoru, může být vyžadována úprava testů, z důvodu chybějících parametrů, které byly přidány při inicializaci fasádových tříd. Po úpravě se systém v rámci refaktoringu opět otestuje.

6.2.2 Přidání metod do fasády

Další fází je přidání metod do fasády, které budou volány zvenčí a volání dále přesměrují na metody konkrétních služeb, které jsou nahrazeny v kódu mimo modul. Je proto nejprve potřeba tyto služby inicializovat v konstruktoru. Metody ve fasádě mají stejné vstupní parametry, návratovou hodnotu i stejnou viditelnost, ale konkrétnější jméno popisující jeho funkci.

6.2.3 Přesměrování volání na fasádu

V tomto bodě je připraveno vše pro přesměrování volání metod z původních služeb na třídy fasády. Opět se musí dát pozor, aby původní služba byla nahrazena správnou třídou fasády a metoda vyměněna za nově vytvořenou metodu fasády. Po dokončení přesměrování je velmi důležité systém znovu otestovat, tentokrát aby došlo ke kontrole, že volání metod přes fasádu je zcela funkční.

6.2.4 Odstranění původních služeb

Po úspěšném otestování je na řadě odstranění již nevyužívaných služeb kódu mimo modul a dojde tak k odstranění přímé závislosti na konkrétní třídu. Během odstraňování se mění počet parametrů v konstruktoru, proto je, stejně jako u inicializace fasádových tříd, potřeba následně upravit související testy a ověřit správnost provedení jejich spuštěním.

6.3 Vytvoření fasád komponent

Postup při tvorbě fasád komponenty probíhá identicky jako při tvorbě fasády pro modul, pouze je potřeba si zde uvědomit co se od čeho odděluje. Roli modulu, který se odděloval od zbytku kódu, si teď postupně vystřídají jednotlivé komponenty a kód, od kterého modul bude odpojován, bude tvořen zbylými komponentami. Jako kontrola, že se v předchozích částech implementace postupovalo správně, jelikož byla odstraněna závislost zbytku kódu na třídách v modulu, se v kroku, při kterém se vyhledávají závislosti, objeví pouze třídy vyskytující se ve stejné komponentě jako zkoumaná služba, v jiné komponentě ze stejného modulu nebo ve vytvořené fasádě modulu. Pokud by se taková závislost našla, lze jí jednoduše odstranit stejným postupem jakým

se odstraňovaly všechny ostatní závislosti. Během celé tvorby fasád komponent se opět testuje po každém uceleném kroku.

6.4 Propojení fasád komponent a modulu

I ve třetí fázi se jedná o velmi podobný postup jako v přechozích dvou fázích. V tomto kroku se odstraní závislosti služeb komponent na fasádě modulu, které tam tak byly pouze dočasně, ale v rámci refaktoringu umožnily postup rozdělit na více menších částí, které se lépe kontrolovaly a mohly se průběžně testovat. V této fázi se nahradí služby používané ve fasádě modulu, třídami tvořící fasády komponent a to stejným způsobem jako tomu bylo při nahrazování služeb třídami fasád v kapitole Vytvoření fasády vnějšího modulu.

Po dokončení závěrečné části implementace se systém naposledy otestuje a výsledkem je modul oddělující zvolenou logickou část systému.

6.5 Vyhodnocení změn

Jako součást zhodnocení účinnosti změn by bylo ideální mít nově příchozí členy, se kterými se provede testování a zhodnocení výsledků práce na systému před zavedenými změnami, a po jejich zavedení. Nicméně takové možnosti nejsou a účinnost musí být vyhodnocena za pomoci dat, které se získají převážně za pomocí IDE a jejich funkcí které nám nabízí. Zhodnocení účinnosti se rozdělí na 3 části, kde se každá z nich zaměří na výsledek řešení jedné z překážek z kapitoly a jejich prospěšnost.

Nízká soudržnost Původní rozdělení kódové základny do balíčků mělo za následek, že se zde nacházely třídy pokrývající alespoň 30 funkcionalit, které spolu úzce nesouvisely. Toto číslo je odvozeno z identifikování logických částí v kapitole 4.3. Nově vytvořený balíček reprezentující modul obchodu s odměnami má tyto funkcionality 4, které jsou ještě navíc rozděleny do jednotlivých komponent, kde každá komponenta má na starosti právě 1 funkcionalitu. Nový člen týmu má nyní možnost prouzkoumat část kódové základny zabývající se vybranou funkcinalitou na jednom místě, kde se nachází omezené množství tříd, které je lépe vstřebatelné.

Vysoká provázanost Vysoká provázanost tříd z balíčků *services* vytvářela velkou nepřehlednost a v některých případech nežádoucí jevy při aplikování změn v systému. Původní počet tříd spadající do modulu obchodu s odměnami mající provázaní se zbylým systémem byl 4. Nyní s vytvořenou fasádou modulu je to 0. Veškeré závislosti jsou přesměrovány na fasádu. Stejné výsledky jsou i mezi komponentami uvnitř modulu, které jsou také schovány za fasádu. Není zde žádná provázanost mezi třídami odlišných komponent.

Degradující modularizace Původní struktura nijak nebránila mít všechny třídy z balíčku *services*, pouze v tom daném balíčku bez dalšího dělení, které by alespoň seskupovalo spolu související třídy, a především tato struktura nijak neomezovala vytváření závislostí tříd mezi sebou. Nyní jsou i spolu související třídy v komponentách zastřešeny fasádou a při potřebě volání metod mezi třídami z odlišných komponent nedochází k vytváření další závislosti mezi konkrétními třídami, ale pouze na třídy fasád. Tím se zabrání postupnému provazování jednotlivých tříd mezi komponentami a výsledkem je přehlednější struktura s určitou hierarchií.

Kapitola 7
Závěr

Cílem této práce bylo zavést změny do backendové části systému Uniqway, které pomocí rozdělení do jednotlivých logických částí přispějí k lepší produktivitě nových členů projektu.

Zadání bakalářské práce je splněno a naplnění jednotlivých cílů je blíže popsáno níže. Prvním z cílů této práce byla identifikace logických částí systému vhodné k rozdělení. Výsledek tohoto úkolu se nachází v kapitole věnující se analýze, konkrétně pak v kapitole 4.3. Dalším cílem bylo navrhnout způsob jakým se jednotlivé části oddělí bez vyžádání komplexních změn a navýšení celkové složitosti backendu. Tento návrh je popsán v kapitole 5. Dále bylo úkolem navrhnuté změny implementovat, tomu se věnuje kapitola 6. Posledním cílem bylo vhodným způsobem vyhodnotit účinnost zavedených změn. Toto vyhodnocení se nachází v kapitole 6.5.

Seznámení a práce s jakýmkoliv rozsáhlým systémem je náročná, obzvláště pokud je potřeba vyznat se téměř v celé jeho šíři, a ne pouze ve vybraných místech. Z toho důvodu nebyl redesign backendu kompletně dokončen. V rámci této práce vznikla analýza a návrh jak oddělit jednotlivé části do modulů. Důraz byl kladen především na zjednodušení systémové struktury pro nové členy. Součástí toho bylo vytvořit návrh pro zavedení změn, které budou vytvářet v modulech téměř identickou strukturu a tím podpořit jednotný systém, v němž se lépe orientuje. Pro důkaz účinnosti a prospěšnosti vytvořeného návrhu byl implementován modul obchodu s odměnami, na jehož základě proběhlo vyhodnocení zavedených změn.

V práci byl detailně popsán postup, jak zanalyzovat dílčí části kódové základny na úroveň jednotlivých komponent, ze kterých se skládá výsledný modul. Dále je zde popsáno, jak tento modul oddělit od zbylého systému a vytvořit v něm potřebnou strukturu, čímž získá požadované vlastnosti. Celková modularizace systému, který je neustále ve vývoji, není snadná, ale pokud se bude postupovat zde navrženým postupem a změny bude koordinovaně zaváděny do systému služby Uniqway, výsledkem bude přehlednější systém, ve kterém bude snadnější pracovat, zavádět nové funkce a přijímat nové členy týmu.

Bibliografie

1. První český univerzitní carsharing — Uniqway [online]. [Cit. 2022-04-27]. Dostupné z: https://uniqway.cz.
2. PROUZA, Petr. *Transformace systému z monolitické architektury do architektury mikroslužeb*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.
3. SEVERA, Štěpán. *Transformace systému z monolitické architektury do architektury mikroslužeb*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.
4. API (Application Program Interface) Definition. *Techterms* [online]. [Cit. 2022-04-27]. Dostupné z: https://techterms.com/definition/api.
5. What is CRUD? *Code Academy* [online]. [Cit. 2022-04-27]. Dostupné z: https://www.codecademy.com/article/what-is-crud.
6. What is HTTP? *Cloudflare Learning* [online]. [Cit. 2022-04-27]. Dostupné z: https://www.cloudflare.com/learning/ddos/glossary/hypertext-transfer-protocol-http/.
7. MVC (Model-View-Controller) Definition. *Techterms* [online]. [Cit. 2022-04-27]. Dostupné z: https://techterms.com/definition/mvc.
8. What is REST? *Code Academy* [online]. [Cit. 2022-04-27]. Dostupné z: https://www.codecademy.com/article/what-is-rest.
9. What Is Software Architecture? *Cast Software glossary* [online]. [Cit. 2022-04-27]. Dostupné z: https://www.castsoftware.com/glossary/what-is-software-architecture-tools-design-definition-explanation-best.
10. Monolithic vs Microservices architecture. *GeeksForGeeks* [online]. [Cit. 2022-04-27]. Dostupné z: https://geeksforgeeks.org/monolithic-vs-microservices-architecture/.
11. FOWLER, Martin. Microservices. *Martin Fowler articles* [online]. [Cit. 2022-04-27]. Dostupné z: https://martinfowler.com/articles/microservices.html#footnote-etymology.
12. How To Understand Monolithic Architecture. *Datamify* [online]. 2021 [cit. 2022-04-27]. Dostupné z: https://datamify.com/architecture/how-to-understand-monolithic-architecture.
13. RICHARDSON, Chris. Monolithic vs. Microservices Architecture. *Microservices* [online]. 2021 [cit. 2022-04-27]. Dostupné z: https://microservices.io/patterns/monolithic.html.

14. HODGES, Nick. Coupling and Cohesion. *Better Programming* [online]. 2020 [cit. 2022-04-27]. Dostupné z: https://betterprogramming.pub/coupling-and-cohesion-75aa16bc9adf.
15. RICHARDSON, Chris. Pattern: Monolithic Architecture. *Microservices* [online]. 2021 [cit. 2022-04-27]. Dostupné z: https://microservices.io/patterns/decomposition/decompose-by-business-capability.html.
16. Microservices:Architecture. *Cinish Medium.com* [online]. 2018 [cit. 2022-04-27]. Dostupné z: https://cinish.medium.com/microservices-architecture-5da90504f92a.
17. RICHARDSON, Chris. Pattern: Decompose by subdomain. *Microservices* [online]. 2021 [cit. 2022-04-27]. Dostupné z: https://microservices.io/patterns/decomposition/decompose-by-subdomain.html.
18. FOWLER, Martin; BECK, Kent. *Refactoring: Improving the Design of Existing Code*. Boston: Addison-Wesley, 2019. ISBN 978-0134757599.
19. FOWLER, Martin. Refactoring [online]. [Cit. 2022-04-27]. Dostupné z: https://refactoring.com/.
20. ŠTRÁFELDA, Jan. Technický dluh [online]. [Cit. 2022-04-27]. Dostupné z: https://www.strafelda.cz/technicky-dluh.
21. Design Patterns [online]. [Cit. 2022-04-27]. Dostupné z: https://sourcemaking.com/design_patterns.
22. Refactoring and Design Patterns [online]. [Cit. 2022-04-27]. Dostupné z: https://refactoring.guru/design-patterns/facade.

Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu média
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `zadani.pdf` ..... zadání práce ve formátu PDF