Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Tvorba aplikací v Ruby on Rails

Lukáš Pospíšil

! ! !

Místo této strany vložíte zadání diplomové práce.

(Do jedné vazby originál a do druhé kopii)

! ! !

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Tvorba aplikací v Ruby on Rails" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 31.3.2014 _____

Poděkování

Rád bych touto cestou poděkoval svému vedoucímu práce doc. Ing. Vojtěchu Merunkovi, Ph.D. za poskytnuté rady a nasměrování v průběhu tvorby této práce. Velké díky též patří mé přítelkyni a rodičům za podporu v průběhu celého studia.

Tvorba aplikací v Ruby on Rails

Souhrn

Tato diplomová práce se zaměřuje na srovnání dvou webových aplikací. První, starší, aplikace je napsána v programovacím jazyce PHP. Druhá aplikace je vytvořená v Ruby on Rails jako součást projektu této diplomové práce. V první, rešeršní, části práce je představen jazyk Ruby a webový framework Ruby on Rails. Tato část též popisuje metody pro měření složitosti softwaru a základy jazyka UML. V druhé, projektové, části je popsána původní PHP aplikace a dále se tato část zaměřuje na tvorbu vlastní Rails aplikace. Na příkladech je prakticky ukázán postup tvorby Rails aplikace včetně nastavení prostředí serveru, na které aplikace běží. V závěru této části je vypočtena softwarová složitost obou řešení. Následně jsou oba výsledky porovnány a na jejich základě jsou zodpovězeny položené otázky.

Klíčová slova: Ruby, Ruby on Rails, webová aplikace, objektové programování, měření složitosti softwaru, HTML5, CSS3, Bootstrap, JavaScript, Nginx, Linux, PostgreSQL, míra LOC, metoda funčních jednic

Application development in Ruby on Rails

Summary

This diploma thesis focuses on comparison of two web applications. First application is older and written in PHP programming language. Second application is created in Ruby on Rails framework as a part of this diploma project. In the first research part of the thesis Ruby programming language and Ruby on Rails framework are introduced. The first research part also covers software estimation methods and UML language basics. Second, thus project part of the thesis describes original PHP application and then covers Rails application development by using real world examples. Setup of server environment is also included. Software sizing calculations for both solutions are presented at the end of project part. Then both results are compared and conclusions are used to answer initial questions.

Keywords: Ruby, Ruby on Rails, web application, object oriented programming, web 2.0, software sizing, Bootstrap, JavaScript, Nginx, Linux, PostgreSQL, LOC method, Function Point Analysis

Obsah

1 Úvod **10**
2 Cíl práce **11**
3 Metodika **12**
3.1 Struktura práce 12
I. LITERÁRNÍ REŠERŠE **13**
4 Přehled řešené problematiky **14**
4.1 Ruby 14
4.1.1 Historie jazyka Ruby 15
4.1.2 Základní syntaxe 17
4.1.2.1 Úvod do syntaxe jazyka Ruby 17
4.1.2.2 Proměnné 19
4.1.2.3 Datové typy 22
4.1.2.4 Příkazy řízení běhu 25
4.1.3 OOP v Ruby 33
4.1.3.1 Třídy a jejich metody 33
4.1.3.2 Dědičnost, mixin a polymorfismus 36
4.2 Ruby on Rails 39
4.2.1 Základní vlastnosti Rails 39
4.2.2 Historie 40
4.2.3 Architektura Rails 41
4.2.3.1 MVC 41
4.2.3.2 Convention over Configuration 43
4.2.3.3 Struktura Rails aplikace 43
4.2.3.4 Gems a Gemfile 45
4.2.3.5 Templates, partials a helpers soubory 47
4.2.3.6 Další technologie obsažené v Rails 49
4.3 UML 52
4.3.1 Class diagram 52
4.4 Složitost softwaru 52
4.4.1 LOC 53
4.4.2 Metoda funkčních jednic 53
II. PROJEKT **55**
5 Vlastní řešení **56**
5.1 Kniha jízd – původní verze 56
5.2 Rails aplikace 58
5.2.1 Příprava prostředí pro Rails 59
5.2.1.1 Nastavení webového a databázového serveru 60

5.2.1.2 Instalace jazyka Ruby 61
5.2.1.3 Instalace Ruby on Rails 62
5.2.2 Vytvoření rails aplikace 62
5.2.2.1 Nastavení databáze 63
5.2.2.2 Další nastavení 63
5.2.3 Scaffolding 64
5.2.4 Bootstrap CSS3 framework 67
5.3 Výpočet složitosti obou řešení 69
5.3.1 Výpočet LOC 69
5.3.1.1 Původní řešení v PHP 69
5.3.1.2 Rails řešení 70
5.3.2 Výpočet FP 70
5.3.2.1 Výpočet FP pro původní řešení 71
5.3.2.2 Výpočet FP pro řešení v Rails 71
6 Zhodnocení výsledků a doporučení 72
7 Závěr 73
8 Použitá literatura 74
III. PŘÍLOHY 78
9 Přílohy 79

Seznam obrázků

Obr. 1 Trend vydávání nových Ruby gems podle [MOSES, 2013] 17
Obr. 2 Třída Numeric podle [FLANAGAN, MATSUMOTO, 2009, s. 42] 23
Obr. 3 Struktura třídy Exception podle [TALIM, 2013] 31
Obr. 4 Interakce jednotlivých vrstev MVC podle [AZAD, 2007] 42
Obr. 5 Původní PHP aplikace - tankování 57
Obr. 6 Původní PHP aplikace - report 58
Obr. 7 Class diagram modelů Rails aplikace 59
Obr. 8 Výsledná Rails aplikace za použití Bootstrapu 68
Obr. 9 Výsledný počet LOC pro Rails řešení 70

Seznam tabulek

Tab. 1 TOP 10 programovacích jazyků na GitHub podle [BARD, 2013] . 14
Tab. 2 Koeficienty pro jednotlivé jazyky podle [JONES, 2013] 54
Tab. 3 Výpočet LOC pro původní řešení v PHP 69

1 Úvod

V posledních letech si webové technologie opět získávají značnou pozornost. Ať už se jedná například o nový standard HTML5, responsivní design pro zvýšení použitelnosti webu i na mobilních zařízeních, AJAX pro vyšší interakci webu s uživatelem. Ve všech případech jde o technologie, které posunuly web na novou úroveň a po letech ustrnulého vývoje HTML 4 se situace začala měnit. Objevily se také velké webové aplikace typu Facebook či Twitter, které tyto technologie využívají. Tyto weby se řadí do nové generace webu, který je označený jako web 2.0. Web 2.0 přináší nový pohled na webové aplikace, kdy oproti dřívější době byl web zaměřen na prezentaci informací uživatelům. U webu 2.0 se toto změnilo a uživatel si často vytváří obsah sám či minimálně aplikace reaguje na jeho požadavky a poskytuje požadovaný obsah.

Vývoj webových technologií jde v posledních letech strmě nahoru a webové stránky naprogramované před deseti lety již mají často svou životnost za sebou. Většinou jsou nahrazovány evolucí stejné aplikace jen s modernějším vzhledem, ovšem dnešní doba umožňuje využít velké množství předpřipravených kusů zdrojového kódu a vlastním kódem tuto funkcionalitu pouze propojit. Je možné tedy místo pouhé evoluce vytvořit původní web za pomoci moderních webových technologií, moderních postupů a modulární funkcionality. Tyto postupy cílí především na zjednodušení práce programátora a zjednodušení případného dalšího vývoje webu. Na trhu existuje celá řada webových frameworků, které cílí na podporu moderních webových technologií, rychlý vývoj webových aplikací a tedy i vysokou produktivitu programátora. Mezi populární webové frameworky s tímto zaměřením lze řadit i Ruby on Rails, který bude v této diplomové práci představen. Je založen na programovacím jazyku Ruby, který obsahuje řadu zajímavých vlastností použitelných nejen pro zjednodušení vývoje softwaru a měl by být oproti běžněji používanému jazyku PHP spíše výhodou. V této diplomové práci bude původní webová aplikace nahrazena řešením v Ruby on Rails a po jejím vytvoření dojde k zhodnocení výsledného a původního řešení. Zhodnocení obou řešení proběhne na základě měr složitosti software, které určují též pracnost obou řešení. Předpokladem je, že nové řešení v Ruby on Rails bude ve výsledku méně složité, tedy méně pracné než původní řešení.

2 Cíl práce

Hlavním cílem této diplomové práce je porovnání dvou řešení jedné webové aplikace. Původního řešení napsaného v jazyce PHP bez použití objektového přístupu k programování a nového řešení v Ruby on Rails, kde je naopak použit pouze objektový přístup. Toto srovnání má potvrdit či vyvrátit předpoklad uvedený v úvodu této práce. Předpokládá se, že u řešení vytvořeného v Ruby on Rails frameworku bude výsledná složitost nižší než u původního řešení v PHP.

Dalšími cíli diplomové práce jsou cíle vedlejší, které vyznikly na základě hlavního cíle. Prvním vedlejším cílem je představit programovací jazyk Ruby a popsat vlastnosti, na kterých webový framework Ruby on Rails staví. Dalším cílem je popsat framework Ruby on Rails a technologie které nabízí a podporuje. Jelikož hlavním cílem této diplomové práce je srovnání dvou webových aplikací, jedné starší a druhé nově vytvořené pomocí Ruby on Rails, je cílem této práce také popsat a na příkladech ukázat postup tvorby Ruby on Rails aplikace a nastínit kroky nutné pro přípravu technického zázemí pro provozování vytvořené Rails aplikace.

3 Metodika

Metodika užitá v diplomové práci je založená na studiu odborné literatury, programování a aplikaci metod pro porovnání složitosti softwaru.

Práce autora se sestávala z následujících kroků:

- Studium odborné literatury
- Naprogramování nové verze webové aplikace v Ruby on Rails
- Porovnání obou řešení s užitím metod a měr pro odhad složitosti softwaru
- Zhodnocení výsledků a porovnání obou řešení

3.1 Struktura práce

První část diplomové práce se sestává z literární rešerše. Cílem této části je seznámení s programovacím jazykem Ruby a také s problematikou tvorby webových aplikací ve frameworku Ruby on Rails, který na jazyku Ruby staví. Na konci této rešeršní části je věnován prostor jazyku UML pro vizualizaci informačních systémů a přiblížena je zde též problematika složitosti softwaru s ohledem na použité metody porovnání složitosti obou řešení. Tato rešeržní část práce je rozdělena na čtyři podkapitoly (Ruby, Ruby on Rails, UML a Složitost software), které se samostatně věnují výše uvedeným tématům.

Ve druhé části diplomové práce, která se věnuje vlastnímu projektu, je představena původní PHP aplikace a stanoveny požadavky na novou aplikaci v Ruby on Rails. Dále je v této části popsána instalace Ruby a Ruby on Rails na operačním systému Ubuntu Linux a také postup a příprava prostředí pro provoz Rails aplikace. Detailně je v této části popsán postup tvorby výsledné Rails aplikace. Na konci této části je vlastní výpočet měr složitosti software podle postupu popsaného v rešeršní části. Šestá kapitola se pak věnuje zhodnocení vypočtených měr složitosti softwaru a autorovým poznatkům získaným při práci na projektové Rails aplikaci. Sedmá kapitola, závěr, pak obsahuje shrnutí diplomové práce a interpretaci výsledků.

I. Literární rešerše

4 Přehled řešené problematiky

Tato kapitola obsahuje podklady a předpoklady, ze kterých autor vycházel při tvorbě projektové Ruby on Rails aplikace a je rozdělena na několik tématicky oddělených podkapitol. Následující podkapitola je věnována úvodu do programovacího jazyka Ruby.

4.1 Ruby

Pokud bychom chtěli Ruby představit jednou větou, tak jde o moderní, víceúčelový, plně objektový programovací jazyk. Ruby podporuje více programátorských paradigmat včetně funkcionálního, imperativního a objektového. Ruby se řadí mezi interpretované jazyky podobně jako Perl, Python, PHP a používá dynamické typování a automatickou správu paměti (garbage collector).

Pokud srovnáme popularitu Ruby s ostatními podobně zaměřenými programovacími jazyky jako je Perl, Python a PHP, tak Ruby z tohoto srovnání vychází nejhůře. Nižší popularita Ruby je viditelné zejména na menším počtu pracovních nabídek pro Ruby vývojáře, ale i tak se Ruby v posledních letech dostává stále více pozornosti.

Pozice	Jazyk	Počet vytvořených repozitářů za r. 2013
1	JavaScript	264131
2	Ruby	218812
3	Java	157618
4	PHP	114384
5	Python	95002
6	C++	78327
7	C	67706
8	Objective-C	36344
9	C#	32170
10	Shell	28561

Tab. 1 TOP 10 programovacích jazyků na GitHub podle [BARD, 2013]

Jak ukazuje analýza Adama Barda, [BARD, 2013], patří Ruby na populárním vývojářském webu GitHub mezi nejpopulárnější jazyky již druhým rokem za sebou. Samozřejmě, že tento výsledek srovnávající pouze počet vytvořených repozitařů nelze považovat za jako kompletní analýzu oblíbenosti jednotlivých jazyků, ale i tak lze konstatovat, že Ruby momentálně patří mezi oblíbené programovací jazyky. Nemalou zásluhu na této oblíbenosti má také webový

framework Ruby on Rails, který je, jak již název napovídá, celý napsán v Ruby. Webovému frameworku Ruby on Rails se podrobně věnuje kapitola 4.2. Ruby on Rails.

V následujících podkapitolách budou přiblíženy jednotlivé aspekty a součásti jazyka Ruby, a to především s ohledem na použité postupy při tvorbě projektové aplikace v páté kapitole s názvem „Vlastní řešení“. Jelikož tato práce si neklade za cíl být učebnicí jazyka Ruby a rozsah diplomové práce ani neumožňuje vše do detailu obsáhnout, je vhodné pro kompletní přehled syntaxe jazyka Ruby využít některou z citovaných knižních publikací.

4.1.1 Historie jazyka Ruby

Historii jazyka Ruby se přehledně věnuje článek společnosti Gunner Technology [GUNNER TECHNOLOGY, 2011], který uvádí několik zajímavých postřehů. Historie programovacího jazyka Ruby začala v 90. letech v Japonsku. Tvůrce Ruby Yukihiro Matsumoto hledal jazyk podobný Perlu, ale plně podporující objektové programování. Při svém hledání ovšem nebyl úspěšný a postupně dospěl k názoru, že bude nutné vytvořit svůj vlastní programovací jazyk. Jeho cílem bylo vytvořit programovací jazyk, který by byl univerzálnější než Perl a zároveň více objektově orientovaný než Python. Pro svůj nový jazyk si vzal za vzor také některé další programovací jazyky, a to především: Lisp, Smalltalk, Eifell. Do Ruby přidal některé jejich dobré vlastnosti a v řadě případů i převzal jejich syntaxi.

Prvotní vývoj jazyka Ruby trval několik let, než byla v roce 1995 vydána první veřejná verze - 0.95. Tyto první verze byly ovšem používané téměř výhradně programátory v Japonsku, zejména proto, že celá dokumentace Ruby byla v té době v japonštině. Od verze 1.3 z roku 1999 se situace začala pomalu měnit. Tato verze přinesla přeloženou dokumentaci do angličtiny, a tak začal Ruby pomalu pronikat do povědomí programátorů i mimo Japonsko. Oproti jiným programovacím jazykům tyto rané verze již obsahovaly většinu objektových vlastností, které známe dnes – třídy, objekty, dědičnost, zapouzdření. Toto tvzení dokazuje například článek Dalibora Šrámka [ŠRÁMEK D., 2002]. Ač je článek starší již více než deset let, tak při srovnání původní Ruby syntaxe s aktuální syntaxí Ruby řady 1.9. není mnoho zásadních rozdílů. Uvedené příklady Ruby kódu v tomto článku lze s minimálními úpravami spustit pod Ruby 1.9.3.

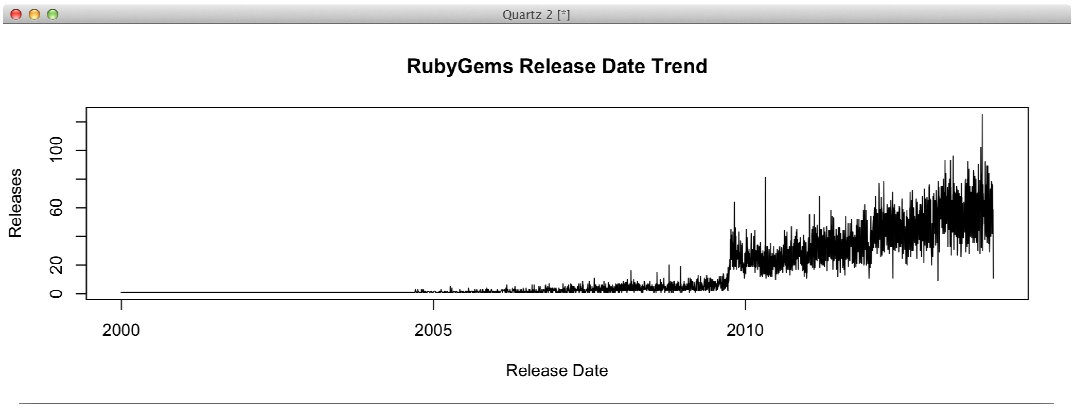
Této stránce historie jazyka Ruby je věnován rozhovor Bruce Stewarta [STEWART B., 2001] se samotným tvůrcem. Autor jazyka Ruby Matsumoto uvádí, že Ruby původně vyvíjel pouze pro sebe. Když ale Ruby v roce 1995 poprvé veřejně vydal, tak byl ohlasy a vřelým přijetím mezi programátorskou komunitou mile překvapen. Matsumoto při tvorbě jazyka Ruby cílil na to, aby programování v Ruby bylo zábavné a zároveň probíhalo zcela přirozeně.

Tedy aby programátor mohl jednoduše vyjádřit své myšlenky pomocí kódu a nemusel s jazykem bojovat. S tím souvisí tolik potřebná jednoduchá syntaxe. Výsledkem je programovací jazyk použitelný pro široké spektrum programátorů. Moderní programování totiž není o dokonalých algoritmech a tisících řádkách vlastního kódu, ale především o tom, jak jednotlivý programátor dokáže využít nabízené vlastnosti programovacího jazyka a jeho prostředí. Vlastní zdrojový kód má pak za úkol vhodně vše spojit ve fungující celek. Ruby k tomu využívá vlastní databázi modulů zvaných „gems“. Ruby „gems“ je věnována samostatná podkapitola v kapitole 4.2 Ruby on Rails.

Pokud se vrátíme zpět k popularitě Ruby. Tato popularita se neustále verzi od verze zvětšovala, i tak ale byl Ruby na začátku 21. století víceméně neznámý pro většinu programátorů. Obecně je autory věnující se Ruby uznáváno, že největší zlom v popularitě nastal v roce 2004, kdy byla vydána první verze Ruby on Rails. Tento fakt potvrdil i samotný autor Ruby Yukihiro Matsumoto v obsáhlém rozhovoru [ZHOU, 2012], kde do detailů rozebírá historii a budoucnost jazyka Ruby. Jen rok po vydání frameworku Ruby on Rails, tedy v roce 2005, si Ruby opět získal pozornost, a to díky společnosti Apple, která přidala Ruby a Ruby on Rails jako standardní součást vývojářského kitu k Mac OS X Leopard. Ruby on Rails se díky tomuto rozhodnutí dostalo tolik potřebné pozornosti ze strany vývojářské komunity a po velmi pozitivním přijetí Ruby on Rails začal i Ruby rapidně nabírat na popularitě.

Vývoj Ruby neustává ani v posledních letech. Dá se říci, že právě naopak. Vývojáři Ruby se snaží reagovat na podněty a Ruby neustále zlepšují. V posledních letech byla Ruby vyčítána malá škálovatelnost a pomalejší vykonávání jednotlivých skriptů a to je uváděno jako jeden z důvodů, proč Twitter začal Ruby nahrazovat jiným řešením. Tento problém je též zmíněn v rozhovoru s Y. Matsumotem [ZHOU, 2012], který toto neklade až tak za vinu Ruby, ale podle něj jde o problém použité architektury. Aktuální stabilní verze Ruby verze 2.1.0 je již několikáté vydání, kde se vývojáři soustředili především na rychlost provádění kódu, na optimalizaci nového virtuálního stroje YARV. Tato verze přináší také nový algoritmus pro „garbage collector“. Při porovnání posledních verzí Ruby na webu Briana Hempela [HEMPEL, 2014] je vidět, že se tato práce vyplácí a poslední vylepšení interpreteru Ruby významně přispěla k celkovému zrychlení. Je tedy vidět, že se Ruby neustále vyvíjí a pozornost neupadá ani po počátečním nadšení kolem Ruby on Rails. Malou ukázkou aktivity komunity kolem Ruby může být článek Johna Mose [MOSES, 2013], který na svém blogu provedl analýzu počtu vydání nových gems, rozšíření pro Ruby. Ná následujícím obrázku je vidět, že počet vydání nových verzí „gems“ se neustále zvyšuje. To jen dokazuje zvětšující se komunita přispěvatelů a uživatelů kolem jazyka Ruby.

Obr. 1 Trend vydávání nových Ruby gems podle [MOSES, 2013]



Nejen z tohoto důvodu je jazyk Ruby často řazen mezi jeden z nejdynamičtěji rostoucích skriptovacích jazyků na světě. Osobně se domnívám, že svou popularitou se může začít měřit s oblíbenými jazyky jako je Perl, Python a PHP.

4.1.2 Základní syntaxe

Tato kapitola se zaměřuje na základní syntaxi jazyka Ruby. Příklady Ruby kódu uvedené v této kapitole jsou vybrány s ohledem na konstrukce Ruby kódu použitého v projektové Rails aplikaci. Pro kompletní přehled o jazyku Ruby lze více než doporučit dvě publikace, ze kterých následující podkapitoly primárně vychází, a to The Ruby Programming Language [FLANAGAN, MATSUMOTO, 2008] a Beginning Ruby: From Novice to Professional [COOPER, 2009].

4.1.2.1 Úvod do syntaxe jazyka Ruby

Jak již bylo zmíněno v kapitole 4.1.1, Ruby od svého vzniku cílí na to, aby programování v něm bylo přirozené a zároveň zábavné. V této kapitole bude rozebráno, zda jde o pravdivé tvrzení.

Ruby je podobně jako Perl či PHP navržen tak, aby programátor mohl použít více způsobů k vyřešení dané úlohy a nakonec mohl vybrat tu nejvhodnější. Tímto se naopak liší od Pythonu, který se spíše snaží představovat doporučované způsoby řešení jednotlivých problémů. Syntaxe jazyka Ruby je ovlivněna programovacími jazyky Perl, Smalltalk, Python a v některých případech i jazykem Lisp. Jak zmiňuje Ruby vývojář Colin Steele ve svém článku „An Introduction to Ruby“, [Stewart, 2001]. Ruby je podle něho ze dvou čtvrtin Perl, z jedné Python a z jedné Smalltalk. I tak si osobně

myslím, že Ruby jde svou vlastní cestou. Ruby cílí na elegantní, dobře čitelný, a tudíž přehledný zdrojový kód. Možnost elegantně větvit jednotlivé kontrukce je z velké části dána plně objektovým návrhem. V Ruby je totiž vše objekt. Velmi dobrá čitelnost kódu naopak vděčí vhodně zvolenému názvosloví jednotlivých vnitřních příkazů, kdy celé kontrukce kódu mají často podobu jednoduchých anglických vět. Následující příklad z knihy Petera Coopera [Cooper, 2009] je toho důkazem:

```
10.times do
  puts "Hello, world!"
end
```

a což vytiskne na obrazovku desetkrát „Hello, world!“:

```
Hello, world!
Hello, world!
…
Hello, world!
=> 10
```

Z uvedeného příkladu lze poznat zjednodušení syntaxe oproti programovacím jazykům, kterými se tvůrci Ruby nechali inspirovat. Tam, kde Perl a PHP používají středník pro oddělení jednotlivých příkazů uvnitř bloku kódu a jednotlivé bloky uzavírají do složených závorek, používá Ruby pro oddělení jednotlivých příkazů znak odřádkování a blok ukončuje klíčovým slovem `end`. Je možné použít i středník a složené závorky, ale není to nutné. Ruby tak poskytuje více možností a pro programátory zvyklé na určitou syntaxi je pak programovaní v Ruby o to jednodušší. Použití středníku se ovšem hodí při použití krátkých programů zvaných „one-liners“, kde je celý program zapsán v jednom řádku. Tento způsob zápisu je znám především z programovacího jazyka Perl.

Příklad takového Ruby one-lineru [IRETON D., 2013]:
`ruby -i -pe "gsub /^# (export https{0,1}_proxy=)/, '\1'" ~/.zshenv`

Uvedený příklad lze pustit přímo z příkazové řádky operačního systému unixového typu a v tomto případě příkaz odkomentuje všechny řádky vyhovující danému pravidlu, resp. provede jejich nahrazení bez počátečního znaku #.

Elegantní syntaxe se nejlépe pozná u složitějších kontrukcí, tedy především při použití OOP.

```
User.find_by_email('me@email.cz').country = 'Czech Republic'
```

Výše uvedený přiklad je z Ruby on Rails a jeho cílem je najít v třídě User všechny uživatele (instance třídy User), kteří mají email me@email.cz a zároveň pochází z České republiky. Argumenty find_by_email a country jsou potom metody třídy User. Volané metody lze řetězit použitím znaku tečky. Dokonce je možné použít stejné metody víckrát za sebou. Následující dva příklady ukazují použití metody reverse spadající pod třídu String:

```
irb(main):001:0> puts 'hello'.reverse
olleh

irb(main):002:0> puts 'hello'.reverse.reverse
hello
```

Dobře navržená syntaxe je silnou stránkou jazyka Ruby. Následující kapitoly se zaměří na jednotlivé oblasti jazyka Ruby detailněji a příslušná syntaxe bude demonstrována na praktických příkladech.

4.1.2.2 Proměnné

V Ruby existuje několik druhů proměnných. Pokud nejde o konstanty, tak proměnné používají jmennou konvenci, kdy je název pouze malými písmeny a pří víceslovných názvech jsou jednotlivá slova oddělena podtržítkem.

Příklad víceslovné proměnné:
`promenna_s_vice_slovy_v_nazvu`

Pokud nejsou proměnné v Ruby inicializované, mají zpravidla hodnotu `nil`, a pokud je navíc interpret Ruby spuštěn s přepínačem `-w`, tak při zavolání neinicializované proměnné vypíše i varovaní.

V Ruby existuje následujcích pět základních typů proměnných, jak popisuje kniha The Ruby Programming Language [FLANAGAN, MATSUMOTO, 2008, s. 87]:

Lokální proměnné

Jsou definované a přístupné pouze v rámci bloku kódu, tedy nejčastěji v rámci metody. Při opuštění bloku se automaticky zruší a ve svém názvu nepoužívají žádný speciální znak.

```
numeric_variable = 10
```

Globální proměnné

Globální proměnné jsou přístupné v rámci celého Ruby programu. Od ostatních proměnných se odlišují počátečním znakem $. Tento systém byl autorem Ruby převzat z programovacího jazyku Perl, kde proměnné používají speciální úvodní znak podle svého typu.

Příklad použití globální proměnné:

```
$global_variable = 10
```

Instanční proměnné

Instanční proměnné jsou definované v rámci dané instance. Pro odlišení používají uvozující znak `@`, jak je vidět na následujícím příkladu:

```
@cust_id=id
```

Jelikož Ruby striktně dodržuje zapouzdření objektů, není možné k těmto proměnným přistupovat přímo mimo objekt. Pro přístup a zápis do instančních proměnných lze využít metody `attr_accessor`, `attr_reader` a `attr_writer`. Více se těmto metodám věnuje kapitola OOP v Ruby.

K instanční proměnné lze také přistupovat pomocí rezervovaného slova self, přičemž následující dva zápisy jsou ekvivalentní:

```
self.var = 19
@var = 19
```

Třídní proměnné

Tyto proměnné jsou definované v rámci jedné třídy a všech jejích instancí. Pro odlišení od ostatních proměnných používají uvozující dva znaky @@. Následující příklad zobrazuje, jak k takovým proměnným lze přistupovat skrze metody:

```
class SuperClass
  @@var1 = 10
```

```
  @@var2 = 20
  def self.out1; @@var1; end
  def self.out2; @@var2; end
end
```

Konstanty

Konstanty jsou speciální typ proměnné, určené pro neměnné hodnoty. Jejich hodnota změnit lze, akorát při pokusu o změnu vypíše interpret Ruby varování. Konstanty jsou označené úvodním velkým písmenem a třídy, které mají standardně v názvu velké písmeno, se také řadí mezi konstanty. Více se třídám věnuje kapitola 4.1.5 OOP v Ruby.

Příklad konstanty s hodnotou:
`Foobar = 10`

Ale jde i:
`FOOBAR = 10`

Speciální proměnné

Tyto proměnné mají v Ruby speciální význam. Jde o rezervovaná slova, tedy nelze vlastní proměnnou pojmenovat například nil.

nil
Vyjadřuje nic a funkce této proměnné je podobná proměnné null v Javě. Jde o jedinou instanci třídy NilClass. nil jako návratová hodnota se vrací například u nedefinované proměnné, pokud se k ní pokusíme přistoupit.

true
Vyjadřuje logickou pravdu. Jde o jedinou instanci třídy TrueClass.

false
Vyjadřuje logickou nepravdu. Jde o jedinou instanci třídy FalseClass.

self
Obsahuje odkaz na aktuální objekt. Používá se při OOP.

__FILE__
Obsahuje název zdrojového souboru.

__LINE__

Obsahuje aktuální řádku ze zdrojového souboru.

4.1.2.3 Datové typy

Ruby obsahuje následující základní datové typy, kterým se detailně věnuje kniha Davida Flanagana The Ruby Programming Language [FLANAGAN, MATSUMOTO, 2009, s. 41] a následující přehled z ní vychází:

Boolean

Reprezentuje logické hodnoty pravda (true) a nepravda (false). Více se jim věnuje předchozí kapitola s názvem Proměnné, jelikož `true` a `false` jsou speciální proměnné, i když jde samozřejmě také o objekty.

String

Textové řetězce v Ruby jsou automaticky instancemi třídy String. Pokud jsou uvozeny dvojitými uvozovkami, je umožněna substituce proměnných do textového řetězce. Pokud je textový řetězec uvozen jednoduchými uvozovkami, tak substituce není možná. Plnou podporu UTF-8 znaků přinesla až Ruby verze 1.9, do té doby bylo nutné explicitně specifikovat, zda jde o ASCII nebo o UTF-8 kódování.

Příklad na substituci:

```
hodnota = 10
puts "Výsledek : #{hodnota*60*60}";
```

V Ruby existuje i globálně přístupná metoda `to_s`, která převádí jednotlivé objekty na String. Příklad použití takovéto funkce:

```
cislo = 5
puts "Cislo je: #{cislo.to_s}"
```

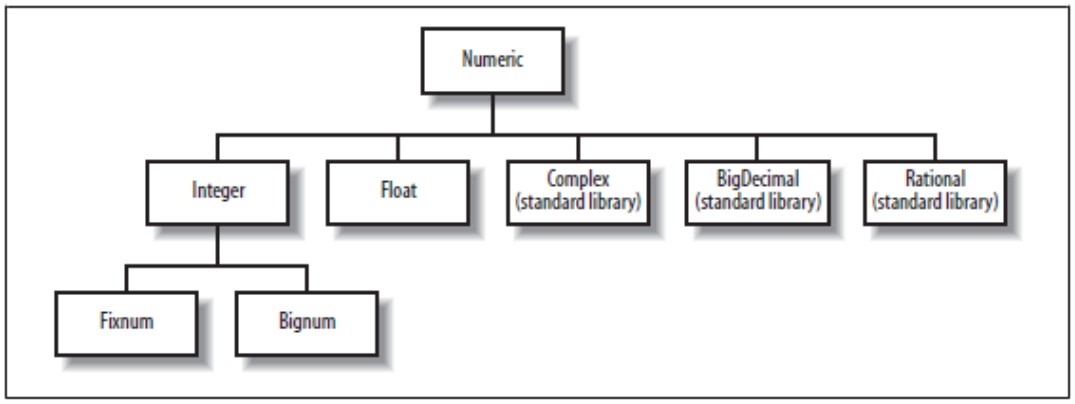
Spojování textových řetězců je podobné jako v Pythonu a používá se znak +, což je změna oproti Perlu a PHP, kde se pro spojování textových řetězců používá znak tečky.

Příklad na spojování řetězců:
`muj_text = “Prvni cast “ + “druha cast”`

Ruby v základu obsahuje velké množství metod pro práci s textovými řetězci. Kompletní dokumentaci třídy String včetně všech metod je možné nalézt na oficiálních stránkách jazyka Ruby.

Number

U čísel je situace trochu složitější, protože zde existuje více podtříd reprezentující jednotlivé číselné typy. Kompletní struktura třídy Numeric je vidět na následujícím obrázku.



Obr. 2 Třída Numeric podle [FLANAGAN, MATSUMOTO, 2009, s. 42]

Integer

Třída Integer reprezentuje všechny celočíselné hodnoty. V závislosti na použité architektuře (32 bitové či 64 bitové) jsou tyto celočíselné objekty instancí třídy `FixNum`, u 32 bitové architektury jde o čísla přibližně do velikosti 31bitů a u 64bitové architektury o čísla přibližně do velikosti 62bitů. Pokud je číslo větší než architektura dovoluje, stane se číslo automaticky instancí třídy `BigNum`. Tuto automatickou konverzi demonstruje následující příklad:

```
irb(main):001:0> 0.class
=> Fixnum

irb(main):002:0> 12222355345469749747949749469.class
=> Bignum
```

Float

Třída Float obsahuje čísla s pohyblivou desetinnou čárkou a metody pro práci s nimi. Číslo typu float se vytvoří automaticky při použití desetinné čárky v čísle:

```
irb(main):003:0> 0.0.class
=> Float
```

Ruby podporuje i další číselné typy, například komplexní čísla v třídě Complex, racionální čísla v třídě Racional atd. Tyto číselné typy ale nejsou hojně používané a pro jejich bližší popis doporučuji nahlédnout do oficiální Ruby dokumentace.

Array

Array neboli pole je uspořádaný datový typ obsahující kolekci různorodých hodnot (objektů). Jednotlivé hodnoty jsou uspořádány podle indexu (klíče) začínající nulou, jde tedy o instance třídy Integer. Další hodnota má pak index o jeden větší a tak dále. Pomocí indexů je také možné k hodnotám přistupovat.

```
Array[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
Key = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Value = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

pole = [  "fred", 10, 3.14, "This is a string", "last element", ]
pole.each do |i|
  puts i
end
```

Hash

Hash je podobný polím. Jako klíč ale k hodnotám používá objekty, tedy nejčastěji textové řetězce či lépe symboly. Ale jinak může jít o jakýkoliv objekt. A stejně tak hodnoty mohou být různého typu. Narozdíl od Perlu je hash v Ruby uspořádaný, Ruby si totiž interně pamatuje pořadí prvků, jak byly přidávány do hashe. V PHP 5 existuje datový typ array, který kombinuje funkcionalitou jak pole, tak i asociativní pole. V české literatuře se někdy hash označuje jako asociativní pole, a to právě z důvodu vazby klíč => hodnota.

Příklad použití hashe:

```
hsh = colors = { "red" => 0xf00, "green" => 0x0f0, "blue" =>
0x00f }
hsh.each do |key, value|
   print key, " is ", value, "\n"
```

```
end
```

Symbols

Symboly jsou speciální datový typ. Jde v podstatě o odlehčené textové řetězce a používají se jako identifikátory. V Rails jsou symboly použité jako odkazy na metody, proměnné či jako klíče v hashi.

Předchozí příklad by za použití symbolů vypadal takto:

```
hsh = colors = { :red => 0xf00, :green => 0x0f0, :blue => 0x00f }
```

S Ruby 1.9 přišla nová syntaxe pro hashe, která výrazně zjednodušila syntaxi a předchozí zápis je možné zapsat zkráceně takto:

```
hsh = colors = { red: 0xf00, green: 0x0f0, blue: 0x00f }
```

Tato syntaxe se v projektové Rails aplikaci hojně vyskytuje.

4.1.2.4 Příkazy řízení běhu

Tato kapitola popisuje konstrukce kódu Ruby, které slouží k řízení běhu programu. Jde především o podmínky, cykly, bloky, iterátory (iterators) a další.

Iterátory

Moderní objektové programovací jazyky používají pro řízení běhu místo klasických konstrukcí jako je `for`, `while` či `foreach` techniky zvané iterátor (iterator). Nejinak je tomu u Ruby. Podobnou funkcionalitu obsahuje třeba i Python. Iterátor je ve skutečnosti metoda, která umožňuje procházet instance (objekty) iterované proměnné.

Základním iterátorem je each, umí procházet objekty pole nebo hashe a je vždy asociovaný s blokem kódu:

```
(0...42).each do |n|
 puts n
end
```

Podobnou konstrukci lze ovšem napsat i za pomoci iterátoru times:

```
42.times do |n|
 puts n
end
```

A to samé lze též zapsat pomocí upto:

```
1.upto(42) do |n|
  puts n
end
```

Další iterátor map slouží obdobně jako u PHP či Perlu k změně obsahu datové struktury:

```
sites = sites.map do |s|
    "#{s}.tutsplus.com"
end
```

Step určuje krok mezi jednotlivými hodnotami rozsahu (range):

```
time1 = '2013-02-02'.to_datetime.to_i
time2 = '2013-02-02'.to_datetime.to_i
(time1..time2).step(1.hour) do |date|
  puts Time.at(date)
end
```

U tohoto příkladu se na chvíli zastavíme. Tento příklad ukazuje, jak je syntaxe Ruby efektivní a přitom jednoduchá. Pokud chcete řešit něco podobného v Perlu či PHP, není syntaxe rozhodně takto přímočará.
Existují však i další iterátory, které jsou více než podobné metodám pro práci s množinami v programovacím jazyce Smalltalk. Tyto metody a jejich použití přehledně shrnuje článek Matthewa Carriera [CARRIERE, 2008].

Mezi další, ale méně časté iterátory patří `collect`, který s blokem kódu asociovaný být nemusí. `collect` funguje obdobně jako `map`, tedy používá se k změně obsahu datové struktury. V následujícím příkladě vrací `collect` nové pole s druhou mocninou vstupních hodnot:

```
a = (0...42)
b = Array.new
b = a.collect {|n| n*n}
puts b
```

`select` slouží k výběru hodnot podle použitého pravidla. Vrácená datová struktura pak obsahuje všechny hodnoty, které na podmínku vrátili `true`:

```
a = [1,2,3,4]
b = a.select {|n| n > 2}
```

Výsledkem je pole b obsahující hodnoty 3 a 4.

`reject` funguje opačně něž `select` a vrací hodnoty, pro které pravidlo neplatí:

```
a = [1,2,3,4]
a.reject {|n| n > 2}
```

Výsledkem je pole b obsahující hodnoty 1 a 2.

Ruby obsahuje další iterátory jako `detect` a `inject`. V článku [CARRIERE, 2008] jsou oba vysvětleny včetně příkladu jejich použití.

Cykly

Ruby též podporuje i standardní kontrukce pro kontrolu běhu programu, jako je for, while, until a unless. Následující ukázky nejsou použité v projektové aplikaci této diplomové práce, ale pro úplnost jsou zde ve stručnosti uvedeny. Podrobně se cyklům věnuje kniha Davida Flanagana, The Ruby Programming Language [FLANAGAN, MATSUMOTO, 2009, s. 127]:

Syntaxe cyklu `while`, který opakuje blok kódu, dokud platí zadaná podmínka:

```
while $i <= $num  do
   puts("Inside the loop i = #$i" )
   $i +=1
end
```

Ukázka cyklu `until`, který funguje opačně než `while` a opakuje blok kódu, dokud podmínka nevrátí `true`:

```
until $i > $num  do
   puts("Inside the loop i = #$i" )
   $i +=1;
end
```

Příklad cyklu `for`:

```
for i in 0..5
  puts "Value of local variable is #{i}"
end
```

Ukázka cyklu `loop`, která funguje jako nekonečná smyčka, tedy pokud uvnitř bloku nedojde k jejímu ukončení pomocí příkazů `break`, `return` či `exit`:

```
i=0
```

```
loop do
  i+=1
  print "#{i} "
  break if i==10
end
```

Oproti tomu PHP neobsahuje takové množství cyklů a kromě standardních, jako je `while`, `for`, se často používá příkaz `foreach`, který slouží k procházení záznamů v poli (`array`) či k procházení objektů u PHP5. PHP5 byla první verze PHP, která přinesla podporu OOP, PHP4 objektově orientované programování nepodporuje. PHP5 také umožňuje definici vlastních iterátorů, ovšem vestavěné iterátory jako má Ruby neobsahuje.

Příklad použití cyklu `foreach` v PHP:

```
<?php
foreach (array(1, 2, 3, 4) as &$value) {
    $value = $value * 2;
}
?>
```

Bloky

Bloky, tak jak jsou vysvětleny Davidem Flanaganem [FLANAGAN, MATSUMOTO, 2009, s. 140] jsou kus kódu, který nemůže sám existovat a vždy se pojí s metodou. Jejich účelem je vykonat při iteraci určitou posloupnost akcí. Jelikož jde o velmi často používanou syntaxi, tak správné zvládnutí bloků patří k základům programování v Ruby. Jejich syntaxe je převzatá z programovacího jazyku Smalltalk. Mezi základní příkazy začínající blok patří `each`, `times`, `map`, `do`, `step`. Více příkladů lze najít v předchozí kapitole iterátory, ale pro úplnost je uveden následující příklad bloku s each bez použití `do`. V tomto případě je nutné použít složené závorky a ukončovací příkaz `end` se vynechává. Tento zápis používá na jednoduché kontrukce, které se nepřesáhnou jeden řádek.

Příklad bloku s `each`:

```
sites.each { |site|
    puts "#{site}.tutsplus.com"
}
```

Ruby též obsahuje kontrukce pro řízení běhu programu uvnitř bloku kódu. Mezi tyto konstrukce patří především `next`, `redo`, `exit` a `return`. Blíže se jim

věnuje kniha The Ruby Programming Language [FLANAGAN, MATSUMOTO, 2009, s. 146].

next
Pomocí tohoto příkazu dojde k přerušení aktuálně prováděné iterace a začně se provádět následující. Tento příkaz se často pojí s podmínkou `if`, která je vysvětlena v následující kapitole.

redo
Tento příkaz vyrestartuje cyklus či iterátor zpět na počáteční iteraci.

exit
Vykonáním příkazu `exit` se iterátor či cyklus ukončí.

return
Příkaz `return` vrací argument jako návratovou hodnotu.

Podmínky

Abychom měli přehled kompletní, tak se tato kapitola věnuje podmínkám v Ruby. Jelikož jde o běžnou funkcionalitu, tak se tato kapitola omezí pouze na uvedení příkladů syntaxe a krátký popis. Detailnější popis lze nalézt v knize Beginning Ruby od Petera Coopera [COOPER, 2009, s. 62], následující řádky z této knihy vycházejí.

if
`if` je omezující konstrukce, kde kód uvnitř této konstrukce je vykonán, pokud je podmínka splněna. Podmínka obsahuje tři typy příkazů `if, elsif, else` a je ukončena příkazem `end`.

Následuje příklad na kompletní `if-elsif-else` podmínku [TutorialsPoint, 2014]:

```
if var == 10
   print ‘‘Variable is 10″
elsif var == ‘‘20″
   print ‘‘Variable is 20″
else
   print ‘‘Variable is something else’’
end
```

unless

`unless` je opakem `if`, kód uvnitř podmínky `unless` se provede, pokud není podmínka splněna. Vhodné použití je například pro kontrolu nenastavených proměněnných. V Rails projektu je unless použit pro kontrolu, zda je uživatel přihlášen.

```
unless defined?
   puts "nedefinovano!
end
```

A příklad z Rails validující přihlášeného uživatele:

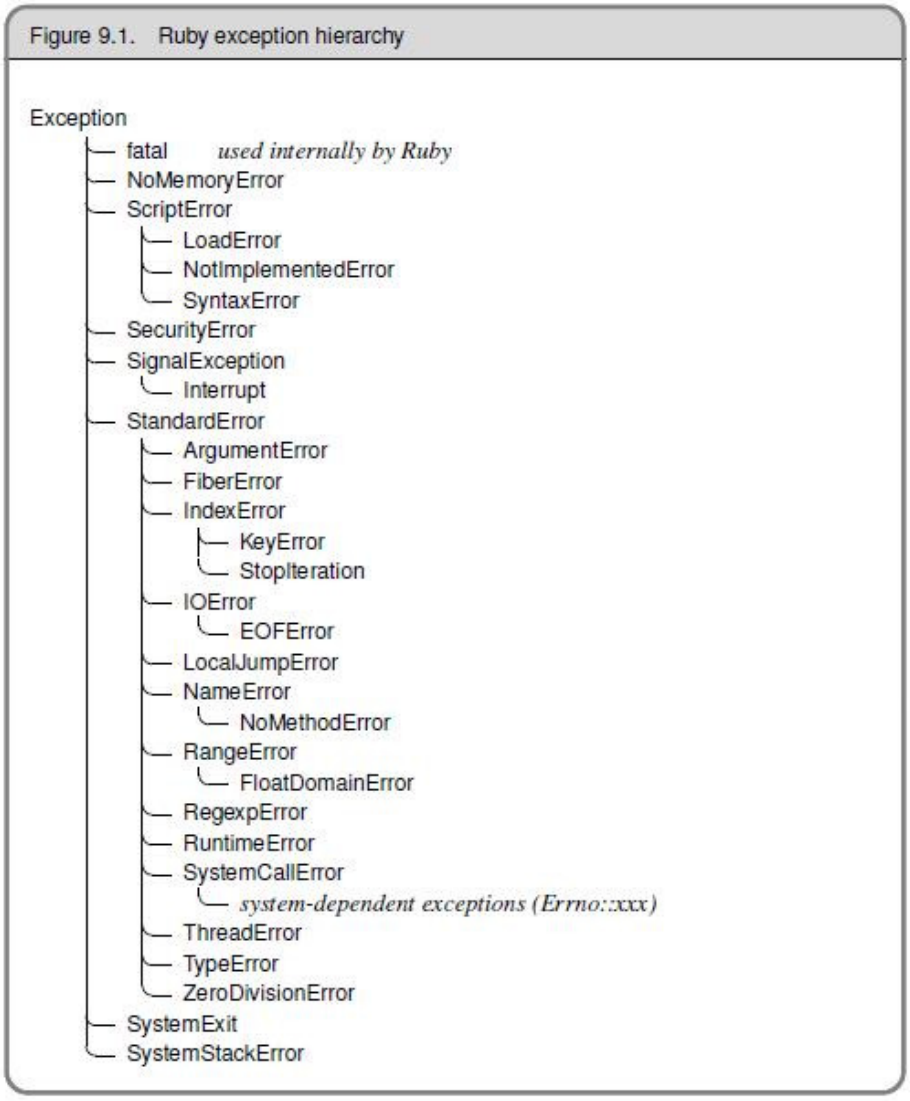
```
unless signed_in?
   store_location
   redirect_to signin_url, notice: "Prosím přihlaste se."
end
```

Ruby narozdíl od např. Pythonu obsahuje také příkaz case, jeho funkce je obdobná příkazu switch z Javy a PHP. Není ovšem v Rails aplikaci použit, a proto zde není jeho syntaxe uvedena. Podrobný popis příkazu case lze nalézt v knize The Ruby Programming Language [FLANAGAN, MATSUMOTO, 2009, s. 123].

Výjimky

Výjimky jsou jednou z dalších výhod OOP jazyků. Ruby systém výjimek obsahuje v standardní knihovně a obdobný systém odchytávání výjimek používá například jazyk Python, Java či C++. Například Perl podobný systém ve standardní knihovně neobsahuje a PHP podobnou funkcionalitu obsahuje až od verze 5.1.0 vydané v roce 2005.

Celý princip spočívá v tom, že pokud některý kus kódu zareaguje jinak, než se předpokládá, dojde vygenerování výjimky (objektu), která se buď na místě zpracuje (odchytí), a nebo neodchytí. V tomto případě putuje výjimka dál do nadřazené části kódu, a pokud ani zde nedojde ke zpracování výjimky, tak aplikace zhavaruje s vypsáním chybového hlášení. Standardně Ruby program neobsahuje žádný kód na obsluhu výjimek, tedy program při jakékoliv vygenerované výjimce zhavaruje. Proto je důležité na toto pamatovat a možné výjimky pečlivě ošetřit. Obecně doporučovaný postup je odchytávat co nejvíce konkrétní výjimky a to nejlépe v místě jejich vzniku.



Obr. 3 Struktura třídy Exception podle [TALIM, 2013]

Jak popisuje výše uvedený obrázek, mají výjimky v Ruby hierarchickou strukturu. Hlavní třída pro výjimky je třída `Exception` a jednotlivé typy výjimek jsou pak podtřídami této třídy. Standardně se odchytávají především výjimky z třídy `StandardError`, které přímo souvisí s vlastním kódem. Ostatní typy řeší jiné defektní stavy jako například nedostatek paměti pro další běh Ruby programu či ukončení běhu programu díky zachycení přerušení od uživatele (`ctrl + c`).

Následující příklad obsahuje doporučovaný způsob ošetření chyb v Ruby:

```
begin
```

```
        # kód, který vytvoří výjimku
rescue StandardError => standard_error
        # kód, kterým ošetříme standardni vyjimku
rescue OtherError => other_error
        # kód, který ošetří jinou výjimku
else
        # kód, který je spuštěn, pokud begin část nevrátí žádnou výjimku
ensure
        # tento kód je spuštěn vždy bez ohledu na nějaké výjimky
end
```

Lze odchytávat všechny výjimky, ale toto je špatný postup:

```
loop do
  begin
    sleep 1
    eval "djsakru3ju3498 += 5u84fior8u8t4ruyf8ihiure"
  rescue Exception
    puts "I refuse to fail or be stopped!"
  end
end
```

Pokud totiž použijeme ve vlastním programu něco podobného, stane se kód imunní na přerušení ze strany uživatele (`ctrl+c`), tedy imunní na výjimku z třídy `SignalException` a také zastavení skrze příkaz `kill` také neproběhne. Ruby ale obsahuje i možnost, jak definovat vlastní výjimky. Příkazem `raise` je pak možné vlastní výjimku vyvolat.

```
class MyCustomError < StandardError
  attr_reader :object

  def initialize(object)
    @object = object
  end
end

begin
  raise MyCustomError.new("an object"), "a message"
rescue Exception => e
  puts e.message # => "a message"
  puts e.object # => "an object"
end
```

Pro naše potřeby takto popsané výjimky stačí, ovšem logika výjimek je velmi obsáhlá, proto lze doporučit knihu The Ruby Programming Language [FLANAGAN, MATSUMOTO, 2008, s. 154], která se v kapitole 5.6 výjimkám detailně věnuje.

4.1.3 OOP v Ruby

Objektově orientované programování, či zkráceně OOP, si získalo v posledním desetiletí velkou popularitu a jde o celosvětově uznávaný a preferovaný přístup pro tvorbu softwaru většího rozsahu, než je pár desítek řádků kódu. Ruby, jakožto plně objektový programovací jazyk, se od svého vzniku snaží o kompletní podporu objektově orientovaného programování.

V Ruby se s ohledem na OOP vyskytují následující pojmy [ŠRÁMEK,2002]:

- Třída (class) – reprezentuje obecnou entitu, kategorii objektů.
- Instance – je konkrétním výskytem třídy.
- Objekt – synonymum termínu instance.
- Proměnná – součástí objektu mohou být proměnné udržující informace o jeho stavu.
- Metoda – funkce spojená s objektem a obvykle měnící jeho stav.
- Atribut – v Ruby označení pro proměnnou objektu, která je dostupná mimo tento objekt.
- Zpráva – objektu je možné zaslat zprávu, která obsahuje informaci o tom, co a jak má objekt provést. Zaslání zprávy objektu je většinou implementováno jako volání metody objektu s parametry.

4.1.3.1 Třídy a jejich metody

Základem OOP je třída, která definuje strukturu všech svých instancí a též definuje pravidla, jakým způsobem dojde k jejich vytvoření a zrušení. K tomu slouží především speciální metody initialize a dále pak metody typu getter a setter, které v Ruby mají název `attr_reader`, `attr_writer`, `attr_accessor`. Jejich podrobnější popis lze najít dále v této kapitole. Metody definované v třídách mají stejnou syntaxi jako funkce a definují se pomocí speciálního příkazu def, který je pak ukončen příkazem end. Vše mezi `def` a `end` je pak kód dané metody.

Nyní si na příkladu ukážeme, jak se taková třída definuje. Základem definice každé třídy je metoda initialize, která má speciální význam. Jak popisuje David Flanagan ve své knize [FLANAGAN, MATSUMOTO, 2009, s. 215], jde o metodu, která provede nastavení prvotního stavu objektu. Je tedy vykonána vždy při vytváření nového objektu.

Příklad definice třídy s metodou initialize:

```
class OsobniAutomobil           # definice začíná klíčovým slovem class

  def initialize(znacka,typ)   # metoda se definuje stejně jako funkce
    @znacka=znacka      # zadané parametry uložíme do instančních
    @typ=typ            # proměnných, jejichž název začíná na '@'
   end

   def to_s                      # ještě jedna metoda
     "X=#{@znacka}, Y=#{@typ}" # vrací řetězec s obsahem proměnných
   end
end                             # definice třídy končí slovem end
```

U výše uvedeného příkladu se na chvíli zastavíme. Ruby je striktní v otázce zapouzdření a pokud bychom se pokusili přistoupit k instančním proměnným napřímo, Ruby vyvolá výjimku, jak popisuje Peter Cooper ve své knize A Beginning Ruby [COOPER, 2009, s. 131]. Pro přístup k instančním proměnným z vnějšku objektu slouží obdobně jako u Javy metody typu getter a setter. Pro přístup obsahuje Ruby speciální proměnnou attr_reader, která se definuje v těle třídy:

```
attr_reader :znacka
```

Použití této metody je ekvivalentní následujícímu zápisu:

```
def znacka
  @znacka
end
```

Pokud potřebujeme naopak měnit obsah instančních proměnných z vnějšku objektu, je nutné k tomu vytvořit setter metodu. V Ruby je toto možné udělat zkráceně pomocí speciální metody `attr_writer`:

```
attr_writer :typ
```

Tato metoda je ekvivalentní zápisu:

```
def typ=(value)
  @znacka = value
end
```

Pokud potřebujeme zároveň číst i zapisovat do instančních proměnných, lze využít služeb speciální metody `attr_accessor`:

```
attr_accessor :znacka, :typ
```

Její použití je ekvivalentní následujícímu zápisu:

```
attr_reader :typ
attr_writer :typ
```

Přidáním výše uvedených metod do definice třídy získáme možnost přistupovat a měnit hodnoty instančních proměnných z vnějšku objektu:

```
class OsobniAutomobil
    attr_reader :znacka
    attr_writer :typ
  .
  .
  .
end

auto = OsobniAutomobil.new(‘‘Ford’’,’’Mondeo’’)
```

Metoda `new` vytvoří nový objekt auto a nastaví jeho prvotní stav podle hodnot argumentů. Po vytvoření můžeme vyzkoušet, zda jde k jednotlivým instančním proměnným přistupovat a v případě proměnné `typ` můžeme i měnit její obsah:

```
auto.znacka
Ford
auto.typ
Mondeo
auto.typ=’’Ka’’
auto.typ
Ka
```

Ne vždy je ovšem dobré mít instanční proměnné takto přístupné. Jelikož jde ve skutečnosti o metody, tak Ruby obsahuje mechanizmus jak řídit přístup k metodám. Pokud není určeno jinak používají metody úroveň oprávnění `public`, tedy přístup k nim se nijak neomezuje. Jak popisuje článek Ruby

Access Control [TALIM, 2006], existují v Ruby tři stupně řízení přístupu k metodám: `public`, `protected` a `private`.

Public
Tyto metody může zavolat kdokoli, a pokud neurčíme jinak, jde o standardní nastavení pro všechny metody.

Protected
Tyto metody mohou být zavolány pouze objekty stejné třídy a její podtřídy.

Private
Takto označené metody mohou být zavolány pouze v rámci konkrétního objektu (self kontext), nelze tedy zavolat `private` metodu jiného objektu.

4.1.3.2 Dědičnost, mixin a polymorfismus

V této kapitole budou představeny pokročilejší formy OOP a to dědičnost, „mixins“ a polymorfismus.

Dědičnost
Dědičnost je základní vlastnost v OOP a umožňuje třídám a podtřídám dědit strukturu a metody přístupem rodič => potomek, kde potomek dědí strukturu a metody obsažené v třídě rodič. Příklad dědičnosti jak ho uvádí Dalibor Šrámek ve svém článku [ŠRÁMEK, 2002]:

```
class Movable < Basic  # třída 'Movable' je potomkem třídy 'Basic'

  def initialize(xpos,ypos,mass,direction=0,velocity=0)
    super(xpos,ypos,mass)   # voláme metodu initialize předka
    @direction=direction    # inicializace přidaných proměnných
    @velocity=velocity
  end

  attr_accessor :direction,:velocity   # přístupové metody k
přidaným proměnným

end
```

Takto vznikne třída `Movable`, která je potomkem třídy `Basic`. Dědičnost se tomuto postupu říká z toho důvodu, že třída `Movable` zdědí všechny metody třídy `Basic`. Ale pokud je potřeba, může třída Movable zděděné metody přepsat svými vlastními.

Pokud chceme zavolat metodu se stejným názvem získanou zděděním z třídy rodič, slouží k tomu příkaz `super`. Následující příklad pak ukazuje použití příkazu `super` v praxi [BRITT, 2014]:

```
module Vehicular
  def move_forward(n)
    @position += n
  end
end

class Vehicle
  include Vehicular   # přidá modul Vehicular
end

class Car < Vehicle
  def move_forward(n)
    puts "Vrooom!"
    super             # zavolá Vehicular#move_forward
  end
end
```

Výše uvedený příklad ukazuje též další běžnou OOP praktiku a to sdílení metod mezi více třídami pomocí modulů. V tomto případě obsahuje modul `Vehicular` pouze jednu metodu, ale jinak není počet metod v modulu nijak omezen.

Mixins

Jelikož Ruby nepodporuje vícenásobné dědění, tak bylo nutné chybějící podporu vícenásobného dědění vyřešit. Řešením je technika zvaná „`mixins`“, což je pouze rozšíření výše uvedeného skládáním více modulů do sebe. Takto je možné jednoduchým způsobem nahradit funkcionalitu vícenásobného dědění a výsledek je pak možné vložit do třídy. Z tohoto postupu ostatně vznikl název této techniky – „`mixin`“.

Příklad použití „`mixin`“ modulů:

```
module D
  def initialize(name)
    @name =name
  end
  def to_s
    @name
```

```
  end
end

module Debug
  include D

  def who_am_i?
    "#{self.class.name}              (\##{self.object_id}):
#{self.to_s}"
  end
end

class Phonograph
  include Debug
  # ...
end

class EightTrack
  include Debug
  # ...
end

ph = Phonograph.new("West End Blues")
et = EightTrack.new("Real Pillow")
puts ph.who_am_i?
puts et.who_am_i?
```

Metody obsažené v modulu `Debug` jsou dostupné v obou instancích (`ph` a `et`) a pomocí mixins získal Ruby funkcionalitu podobnou vícenásobnému dědění. Více příkladů „mixins“ lze nalézt v článku Satishe Talima [TALIM, 2006], ze kterého výše uvedený popis a příklad vychází.

Polymorfismus

Polymorfismus je schopnost objektů různě reagovat na stejnou zprávu. Ruby jako plně objektový jazyk tuto vlastnost podporuje, a v Ruby je dokonce možnost dosáhnout polymorfismu více způsoby, a to pomocí dědění či pomocí techniky zvané „mixins“. Blíže se polymorfismu věnuje článek Fredericka Cheddy, [CHEDDA,2012].

4.2 Ruby on Rails

Po kapitole věnované jazyku Ruby následuje kapitola, která se zaměří na známý webový framework Ruby on Rails. Ruby on Rails je moderní webový framework napsaný kompletně v programovacím jazyce Ruby. Rails si klade za cíl zjednodušit programování webových aplikací a udržet tak vysokou produktivitu práce programátora. Obdobně jako Ruby má otevřené zdrojové kódy a je vydaný pod svobodnou MIT licencí.

Následující podkapitoly jsou koncipované jako úvod k Rails a pro kompletní popis jednotlivých vlastností doporučuji využít výbornou knihu o Rails od Sama Rubyho a jeho kolegů [RUBY, THOMAS, HANSSON, 2011], ze které následující kapitoly teoreticky vycházejí, pokud není uvedeno jinak.

4.2.1 Základní vlastnosti Rails

Primárním cílem frameworku Ruby on Rails je zjednoušení a zrychlení tvorby webových aplikací. Využívá k tomu nejen vlastností moderního programovacího jazyka Ruby, ale přidává k tomu některé své vlastní přístupy. Tyto přístupy často ovlivňují nejen adresářovou strukturu Rails aplikace, ale také například pojmenování jednotlivých proměnných. Přístupů nebo chcete-li konvencí, využívá Ruby on Rails několik.

Zde jsou vyjmenovány ty hlavní:

- **DRY (“Don’t repeat yourself”)** – jak již název napovídá, cílem tohoto přístupu je minimalizovat duplicitní zdrojový kód a také konfiguraci. Proto Rails standardně obsahují podporu pro partials, templaty, využívají dědičnosti a »mixins« atd. Výhodou tohoto přístupu je zjednodušení úpravy zdrojového kódu webové aplikace, kdy úpravy provádíme ideálně pouze na jednom místě.
- **CoC (“Convention over configuration”)** – tímto přístupem je také výhodné se řídit, jde například o jmennou konvenci tříd, metod, tabulek, souborů atd. Tento přístup má výhodu v tom, že při dodržování obecných Rails konvencí není potřeba téměř žádná další konfigurace a Rails si jednotlivé komponenty mezi sebou propojí pouze na základě vhodně zvoleného názvu. Tento přístup tedy nejen ulehčuje práci programátorovi, ale také zpřehledňuje výsledný zdrojový kód pro případné další programátory. U velkých projektů také snižuje počet

nutných rozhodnutí, protože není potřeba řešit pojmenování jednotlivých metod, tříd atd.

- **MVC ("Model-View-Controller")** – jde o návrhový vzor, který rozděluje aplikaci do tří navzájem oddělených vrstev – model, view, controller. Tyto vrstvy pokrývají jednotlivé oblasti interakce Rails aplikace s vnějším světem (daty, uživatelem, business logikou). MVC architektura je detailně představena v kapitole MVC.

4.2.2 Historie

Historie Ruby on Rails je poměrně krátká. Vše odstartovalo rokem 2004, kdy byla vydána první veřejně dostupná verze. Autorem Ruby on Rails je mladý dánský programátor David Heinemeier Hansson, který framework Ruby on Rails napsal jako součást aplikace Basecamp pro svého zaměstnavatele. Ruby on Rails tedy existoval již před vydáním první veřejně dostupné verze a teprve po rozhodnutí v rámci firmy 37signals, která stojí za aplikací Basecamp, došlo k oddělení Rails od Basecamp a jejich samostatnému vydání pod otevřenou licencí a to včetně zdrojových kódů. Další velký mezník v popularitě zaznamenal Ruby on Rails v roce 2005, kdy po spolupráci s firmou Apple byl Ruby on Rails framework přidán jako standardní součást vyvojářských programů pro Mac OS X Leopard. Nasazení Ruby on Rails na velkých webových projektech jako je například Github, Twitter, Shopify, Basecamp ukazuje, že i velké webové aplikace lze stavět na Rails.
V současné době (březen 2014) byla vydána stabilní verze Rails 4.0, verze 4.1 je ve stavu Release Candidate 1 a předchozí stabilní řada Rails 3.2 začíná postupně dosluhovat. S ohledem na prohlášení tvůrce Ruby, Davida Hanssona [HANSSON, 2014], k vydání Rails 4.1 RC1, je tato verze připravená na produkční nasazení, protože je již několik měsíců používána v produkční verzi aplikace Basecamp. Ovšem moje vlastní zkušenost je taková, že verzím řady 4 chybí stále podpora u některých gems balíků a proto je dle mého názoru výhodnější setrvat u Rails řady 3.2.

Článek Sayanee Basu věnovaný historii Rails [BASU, 2013] obsahuje velmi pěkný přehled nové funkcionality hlavních vydání Rails:

- Rails 1.0 (prosinec 2005) – opravné vydání, došlo k vyčištění zdrojového kódu.
- Rails 1.2 (leden 2007) – REST.
- Rails 2.0 (prosinec 2007) – lepší routovaní zdrojů, multiview, HTTP základní autentifikace, cookie store sessions.
- Rails 2.0 (listopad 2008) - i18n – multi jazyková podpora, thread safe volba, connection pool, Ruby 1.9, Jruby.
- Rails 2.3 (březen 2009) - Templaty, Rack podpora.

- Rails 3.0 (srpen 2010) – active record – objektový přístup ke zdrojům v databázi, nový router pro kontroler, kontroler pro maily, CRSF ochrana (cross site scripting).
- Rails 3.1 (Aug 2011) - jQuery, SASS, CoffeeScript, Sprockets.
- Rails 3.2 (Jan 2012) – opět nový routovací systém, faster development mode, automatic query explains, tagged loggin for multi-user application.
- Rails 4.0 (Jun 2013) – podpora pouze pro Ruby 1.9.3 a vyšší, přidává nové možnosti pro cachování obsahu, zapnutý threadsafe mód v základu.

4.2.3 Architektura Rails

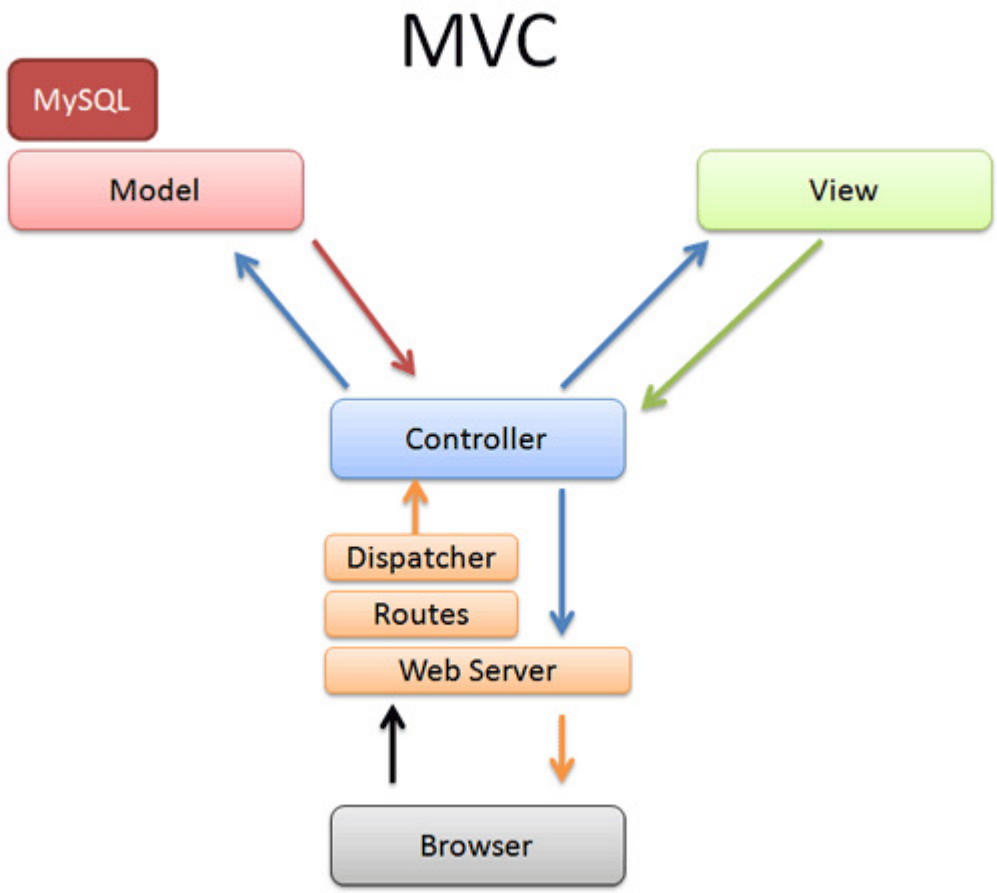
Ruby on Rails obsahuje řadu zajímavých technologií, které usnadňují vývoj. V následujících podkapitolách budou některé z nich přiblíženy. Popisované technologie jsou vybrány s ohledem na jejich použití v projektové Rails aplikaci.

4.2.3.1 MVC

Ruby on Rails, jak již bylo zmíněno v úvodu, využívá prezentačního návrhového vzoru model – view – controller (MVC). Tento návrhový vzor rozděluje logiku aplikace mezi tři oddělené vrstvy.

- Model – obsahuje data, datovou strukturu a bussiness logiku aplikace
- View – obsahuje kompletní uživatelské prostředí

- Controller – stará se o tok událostí a interakci mezi jednotlivými vrstvami a obsahuje též aplikační logiku



Obr. 4 Interakce jednotlivých vrstev MVC podle [AZAD, 2007]

Jak uvádí Borek Bernard ve svém článku věnovaném MVC architektuře [BERNARD, 2009], u implementace MVC často dochází k problému, kdy toto rozdělení aplikace do tří oddělených vrstev nedá programátorovi dostatek informací pro jeho práci. Zejména pokud narazí při programování na problém, který má například funkčně spadat pod `View` vrstvu, ale obsahuje i celkem komplexní logiku a ta naopak spíše patří pod `Controller` vstvu. U Rails je tento problém vyřešen použitím konvence (CoC), která pokud je dodržována, tak daný problém vůbec nenastane. Pokud by se tak stalo, je velmi pravděpodobné, že s ohledem na popularitu Rails obdobný problém už někdo řešil.

Nejlépe je zakomponování MVC v Rails vidět na adresářové struktuře, kdy je celá složka app/ rozdělena na tři hlavní podadresáře app/view/, app/controller/ a app/model/. Adresářové struktuře Rails aplikace se podrobně věnuje kapitola Struktura Rails aplikace.

4.2.3.2 Convention over Configuration

Ruby on Rails přináší přístup zvaný Convention over Configuration (CoC). Za tímto názvem se skrývá celý systém od pojmenování modelu, přes pojmenování tabulky v databázi až po pojmenování HTML (Erb) souborů pro jednotlivé akce controlleru. Cílem celého systému je omezit nutnou konfiguraci Rails aplikace na naprosté minimum a zbytek lze již odvodit z použitých názvů třídy, modelu, controleru atd.

Následující příklad ukazuje, jak taková jmenná konvence funguje:

- Třída: Car – často jediný argument scaffold příkazu, ze kterého si Ruby „domyslí“ zbytek
- Instance třídy: @cars = current_user.cars.all
- Helper soubor: app/helpers/car_helper.rb
- Model: app/models/car
- Views: app/views/cars/
- Controller: app/controllers/cars_controller.rb, třída: CarsController

4.2.3.3 Struktura Rails aplikace

Všechny Rails aplikace používají víceméně stejnou adresářovou strukturu. Ač se to může zdát jako nevýhoda, zejména u malé Rails aplikace, tak toto lze po nějaké době práce v Rails hodnotit jako výhodu. Časem totiž není problém se zorientovat v jakékoliv Rails aplikaci. Níže uvedená adresářová struktura odpovídá adresářové struktuře projektové aplikace. Při začátcích práce s Rails je vhodné využít nějaký dobře zpracovaný tutoriál s příklady, jeden takový napsal Michael Hartl [HARTL, 2012, kapitola 1.2.6] a tato kapitola z něj čerpá.

├── app
│ ├── assets – obsahuje soubory s JavaScriptem, CSS soubory atd.
│ ├── **controllers** – obsahuje kód kontrolérů
│ ├── **helpers** – obsahuje tzv. „pomocné“ skripty
│ ├── mailers – obsahuje kontroléry pro odesílání emailů
│ ├── **models** – obsahuje soubory s kódem jednotlivých modelů
│ └── **views** – obsahuje template .erb soubory (HTML a Ruby)
├── config
│ ├── **environments** – obsahuje konfigurace pro jednotlivá prostředí

```
|   ├── initializers
|   └── locales – obsahuje konfigurace pro jednotlivé jazykové mutace
├── db
|   └── migrate – obsahuje soubory, které mění strukturu databáze
├── doc – obsahuje generovanou dokumentaci k dané aplikaci
├── lib
|   ├── assets
|   └── tasks
├── log – obsahuje logy dané aplikace
├── public
|   └── assets
├── script
├── spec
├── test – obsahuje automaticky generované testy – skripty v Ruby zajišťující automatické testování Rails aplikace
├── tmp – temp adresář, obsahuje např. cache, údaje o jednotlivých „sessions“, atd.
└── vendor – místo pro software třetích stran
    ├── assets – css, fonty, javascript, atd.
    └── plugins – skripty, ruby moduly, helper soubory, atd.
```

Takto vypadá adresářová struktura běžné Rails aplikace. Za povšímnutí stojí rozdělení `app/` adresáře na `app/controllers`, `app/views`, `app/models`. Tyto tři adresáře jsou součístí MVC a obsahují většinu uživatelského kódu. Lze je tedy označit za nejdůležitejší adresáře v Rails.

V Rails, i přes snahu minimalizovat počet nutných nastavení, existuje několik konfiguračních souborů. Následující soupis obsahuje jen ty důležité, které bylo nutné editovat v průběhu práce na projektové Rails aplikaci:

- **Gemfile** – obsahuje soupis použitých modulů, lze vynutit použití konkrétní verze daného modulu, umístit modul do konkretního prostředí atd.
- **config/application.rb** – obsahuje obecné nastavení pro vlastní Rails aplikaci, tedy například jazyk, časovou zónu, kódování atd.
- **config/database.yml** – obsahuje nastavení pro jednotlivé databáze (`development`, `test`, `production`), nastavuje se zde použitý databázový engine, jméno databáze, uživatelské jméno a heslo atd.
- **config/environments/production.rb** – obsahuje nastavení pro produkční prostředí

- **config/environments/development.rb** – obsahuje nastavení pro vývojové prostředí
- **config/environments/test.rb** – obsahuje nastavení pro testovací prostředí

Samotné Rails kromě samotného frameworku obsahují též webový server Webrick pro spuštění Rails aplikace bez nutnosti další konfigurace. To je vhodné především pro rychlý vývoj aplikace. Dále konzoli pro objektový přístup k vytvořené aplikaci a sadu skriptů pro zjednodušení správy Rails aplikace. Rails konzoli lze vyvolat pomocí příkazu `rails console` uvnitř app/ adresáře.

4.2.3.4 Gems a Gemfile

Gems je balíčkovací systém jazyka Ruby. Ač se může zdát nelogické zařadit kapitolu Gems pod Ruby on Rails, tak to svůj důvod má. Ruby on Rails totiž na této funkcionalitě staví a využívá ji pro různá rozšíření. Protože je tato diplomová práce zaměřena na vývoj aplikací v Ruby on Rails, tak gems jsou použity pouze v součinnosti s Rails frameworkem.

V prostředí GNU/Linuxu se k jednotlivým balíčkům přistupuje skrze příkaz `gem`, který můžeme v Rails použít také, ovšem pro Rails je rozhodující obsah souboru Gemfile v kořenovém adresáři Rails aplikace. V tomto souboru jsou vyjmenované součásti, které daná Rails aplikace používá. Například řádkou `gem 'rails', '3.2.17'` v Gemfile souboru říkáme, že chceme pro tuto Rails aplikaci použít balík rails verze 3.2.17. Jednotlivé balíky ale mohou mít i závislosti, a proto se nainstalováním gem balíku rails nainstaluje velké množství zavislých balíků. Například MVC je v Rails implementován pomocí několika balíků – Active Model a Action Pack obsahující tyto tři balíky Action Controller, Action View a Action Dispatcher.

Přehled standardních Rails balíčků popisuje Adrian Mejia ve svém článku [MEJIA, 2011] a následující přehled z jeho článku vychází:

- **Action Mailer** – poskytuje podporu pro odesílání emailů
- Action Pack
  - **Action Controller** – poskytuje funkce controller vstvy v MVC
  - **Action Dispatcher** – spravuje "routes" v Rails aplikaci, tedy překlad URL na konkrétní dotazy na jednotlivé controllery potažmo modely
  - **Action View** – poskytuje funkcionalitu view vrstvy v MVC
- **Active Model** – poskytuje modelům funcionalitu jakou mají objekty (databázové záznamy) skrze Active Record

- **Active Record** – poskytuje abstraktní přístup k DB, kdy jednotlivé tabulky v databázi jsou převedny na třídy, jednotlivé záznamy (řádky) na instance třídy a jednotlivé atributy (sloupce) jsou přístupné skrze metody dáné třídy
- Active Resource – již není standardní součástí Rails, poskytuje přístup k externím REST webovým službám
- **Active Support** – poskytuje podporu pro použití více jazykových mutací aplikace současně
- **Railties** – poskytují podporu pro nové moduly, např. vazby jmenné konvence, atd.
- **Rails** – tento balík spojuje vše dohromady tak, aby poskytovaná funkcionalita opravdu fungovala, jak je zamýšleno

Oficiální stránka `http://rubygems.org` uvádí okolo 77 tisíc dostupných `gems`, ale to je celkové číslo pro Ruby a další Ruby framework a projekty. I tak ale je pro Rails dostupných několik tisíc balíčků. Pokud se chystáme do Rails aplikace přidat novou funkcionalitu, je vhodné nejdříve prohledat databázi gems. Je totiž pravděpodobné, že vhodný modul již existuje a není potřeba vše programovat znovu. Často se ale stává, že existuje na daný problém několik podobných „`gem`“ balíčků. Vybrat proto vhodné `gems` není lehké, pokud nechceme jít cestou pokus omyl. Stefano Mancini ve svém článku [MANCINI,2013] připravil přehled doporučovaných Rails balíků. Tento přehled byl poté doplněn s ohledem na `gems` balíčky použité v projektové Rails aplikaci:

- **simple forms** – gem pro zjednodušené generování formulářů
- **bootstrap-sass** – gem přinášející podporu pro bootstrap3 CSS3 framework spolu s SASS
- **will_paginate** – gem starající se o stránkování při větším než malém množství vypsaných záznamů
- **bcrypt-ruby** – gem obsahující zabezpečení ve formě generování hashe z hesla uživatelů
- **active-merchant** – gem, který přidává podporu pro e-commerce, tj. platby kartou atd.
- **device** – gem zabezpečující kompletní autentifikaci uživatelů
- **bootstrap3-datetimepicker-rails** – bootstrap3 kompatibilni gem pro CSS3 výběry dat
- **rspec** – gem pro testování Rails za pomocí uživatelských scénářů
- **active admin** – gem pro snadnější generovaní administrační části aplikace

Standardně se instalace nových „gems“ provádí příkazem `gem install <název_balíku>`, který po spuštění balíček stáhne a nainstaluje. Rails ale obsahují příkaz `bundle`, který instaluje „gems“ podle zadání obsažené v konfiguračním souboru. Rails pro definici použitých balíků obsahují soubor `Gemfile`. V tomto souboru jsou uvedeny všechny použité „gem“ balíky v Rails aplikaci a lze určit zda se balík má použít v konkrétní verzi, poslední verzi či se má udržovat poslední verze určité řady.

```
gem 'rails', '3.2.17' # použitá bude pouze verze 3.2.17
gem 'bootstrap-sass' # použije se poslední dostupná verze
gem 'bcrypt-ruby', '>= 3.0.1' # použije se verze minimálně 3.0.1
gem 'execjs', '~> 1.4.0' # použije se nejvyšší verze řady 1.4
```

Pokud tedy potřebujeme nainstalovat nový balík do Rails aplikace nejvhodnější je upravit soubor `Gemfile` a příkaz `bundle update` se pak postará o zbytek.

4.2.3.5 Templates, partials a helpers soubory

Ruby on Rails staví na přístupu DRY (Don´t repeat yourself), což představuje snahu o co nejmenší duplicitu kódu. To je zajištěno používáním templatů, „partials“ a „helper“ souborů.

Templates
Templaty (templates) obsahují mix Ruby kódu s HTML kódem a jsou základem uživatelského prostředí Rails aplikace. Jde tedy o view vstvu v MVC architektuře. Rails v základu podporují dva druhy templatů a to ERB – Embedded Ruby a HAML. Ač mohou templaty obsahovat i kód Ruby, není obecně doporučováno používat mnoho logiky v templatech a je vhodné tuto logiku oddělit do speciálních helpers souborů.

Helpers
Helpers soubory obsahují většinou metody pro daný controller či model a využívá funkcionalitu „`mixins`“ pro import svého obsahu do požadované třídy.

Partials
Partials jsou naopak části kódu, které se vkládají do jednotlivých templatů. Může jít například o formulář, který je stejný jak pro editaci (metoda `edit`), tak pro vytváření nových položek (metoda `new`). Úspora je zjevná, není důvod udržovat dvě stejné části kódu, když stačí pouze jedna.

Templaty jsou součástí View vrstvy MVC, proto je najdeme v adresáři `app/vieiws/<název_modelu>`. Protože jde primárně o soubory obsahující HTML kód, používá Ruby podobně jako PHP znak <%= pro oddělení logiky od HTML:

```
<h1>Nové vozidlo</h1>
<%= render 'form' %>
```

Použití <%= znamená, že dojde k vytištění návratové hodnoty této metody. Pokud chceme v .erb souborech použít například podmínky, je nutné použít pro uvození Ruby kódu znak <%. Následující příklad je ze souboru `_header.html.erb` obsahující hlavičku projektové Rails aplikace:

```
<% if current_user.admin? %>
   <li><%= link_to "Uživatelé", users_path %></li>
<% end %>
```

Řádka `<%= render 'form' %>` volá metodu render pro načtení a vygenerování obsahu partial souboru. Partials mají dle jmenné konvence název začínající podtržítkem. V tomto případě jde o soubor `_form.html.erb`. Jak je vidět z přípony souboru, jde o Embedded Ruby soubor, platí tedy všechna výše uvedená pravidla. Rozdíl je ten, že můžeme metodu `render` volat z více míst najednou a sdílet takto oddělený kód bez nutnosti vytvářet duplicitní zápis.
Narozdíl od templates a partials nevyužívají helper soubory Embeded Ruby zápis. Helper soubory fungují jako „mixins“ a jde tedy o moduly jazyka Ruby, tento konkrétní příklad je též z projektové Rails aplikace:

```
module SessionsHelper
.
.
.
  private
   def signed_in?
     !current_user.nil?
   end
end
```

A poté je tento modul přidán do `ApplicationController` třídy, tím se stane dostupný pro ostatní kontroléry, jelikož všechny uživatelské kontroléry jsou podtřídou `ApplicationController`:

```
class ApplicationController < ActionController::Base
  protect_from_forgery
  include SessionsHelper
  .
  .
  .
end
```

4.2.3.6 Další technologie obsažené v Rails

Rails podporují velké množství technologií, ať už přímo či skrze gems balíčky. Nejde je všechny s ohledem rozsah této diplomové práce probrat. V této kapitole budou některé ve stručnosti popsány.

Migrations

Migrace je způsob jakým, Rails řeší průběžný vývoj aplikace. Není vždy pravidlem, že se definice modelu povede na první pokus a v průběhu životnosti aplikace se nebude měnit. Pro tyto případy poskytuje framework Rails řešení v podobě migrací (migrations). Pokud například pomocí scaffold příkazu vygenerujeme nový model, vygeneruje se i soubor migrations_<datum>.rb, který definuje novou strukturu tabulky v SQL databázi, která odpovídá konkrétnímu modelu. Jelikož model je již pomocí scaffold příkazu vytvořen, je zde nekonzistence mezi modelem a databází. Pro odstranění této nekonzistence a vytvoření dané tabulky v databázi slouží příkaz rake db:migrate, který aplikuje danou změnu nejprve na vývojovou databázy (databáze development), pomocí proměnné `RAILS_ENV=production` a zopakováním stejného příkazu se změna provede i na produkční databázi. Každá provedená migrace se pak zapíše do databáze do tabulky migrations, je tedy možné mít databázi development v jiné revizi než databázi produkční.

REST

REST neboli Representational State Transef je přístup, který Rails používá pro interakci mezi View vrstvou a vnějším světem. Jak popisuje Sam Ruby ve své knize věnované Rails [RUBY, HANSSON, 2011, s. 314], je REST založen na bezstavové komunikaci mezi Rails a jednotlivými klienty. K určení stavu slouží parametry zakomponované do jednotlivých dotazů a logické oddělení zdrojů pomocí URL. Zdrojů může být více typů, standardní je poskytovat

výstup v HTML, ale k dispozici jsou i další XML pro RSS čtečky, JSON pro AJAX a jQuery. Tímto standardizovaným rozhraním umožňuje REST nejen vcelku jednoduché přidávání nové funkcionality do Rails aplikace, ale zároveň i umožňuje komunikaci s jinými produkty, ať už se jedná o jQuery JavaScript knihovnu, RSS čtečky či mobilní aplikace.

Scaffolding

Scaffolding je proces generování MVC struktury v Rails. Pokud chceme vytvořit novou třídu objektů, můžeme to udělat více způsoby. Buď ručně vytvořením každého souboru zvlášť a naplnit je potřebným obsahem, a nebo využít služeb příkazu `scaffold`, který vygeneruje konstru aplikační struktury za nás. Jak popisuje kniha „Agile Web Development with Rails“ [RUBY, HANSSON, 2011, s. 82] na příkladu generujícím třídu Product, tak následujícím příkazem dojde k vytvoření této třídy a to včetně všech vrstev MVC, CSS stylů, unit testů, helper souborů a vše je navzájem prolinkované a funkční.

```
rails generate scaffold Product \
title:string description:text image_url:string price:decimal
```

Tento přístup opět šetří programátorovi čas a zrychluje vývoj. Příkaz scaffold umí ale daleko více. Z důvodu použití CSS3 frameworku Bootstrap z dílny Twitteru v projektové Rails aplikaci je nutné zakázat generování CSS souborů pomocí přepínače.

jQuery a AJAX

Spolu s Rails 3. řady přišla i zabudovaná podpora pro AJAX a jQuery. jQuery je multiplatformní JavaScript knihovna, která zjednodušuje práci s JavaScriptem. AJAX (Asynchronous JavaScript and XML) technologie umožňuje přidat více interaktivity mezi aplikací a klientem. Lze tedy na základě akce uživatele načíst nový obsah části stránky bez nutnosti obnovit celou stránku. Přidávání technologie AJAX do fungující Rails aplikace se detailně věnuje kapitola 11 knihy Sama Rubyho a jeho kolegů [RUBY, HANSSON, 2011, s. 143].

RSS

Pokud vytváříme například blog v Rails či naše Rails aplikace obsahuje stránku s novinkami, je vhodné umožnit uživatelům přidat si naši stránku mezi sledované weby do RSS čtečky. Interně je RSS pouze další ze způsobu přístupu k jednotlivým zdrojům. V knize „Agile Web Development with Rails“

[RUBY, HANSSON, 2011, s. 177] je uveden návod jak přidat RSS do Rails aplikace.

HAML
Rails kromě Embedded Ruby (Erb) jako templatovací technologii umožňuje využít i templatovací jazyk HAML(HTML abstraction markup language), který je úspornější v zápisu HTML a Ruby konstrukcí. Soubory pak používají místo `.erb` příponu `.haml`. Jde o oblíbenou náhradu za Erb především z důvodu úspornějšího zápisu.

CoffeeScript
CoffeeScript je technologie, která souvisí opět s JavaScriptem. Jde o jazyk, který po zkompilování vygeneruje JavaScript. Jelikož je jeho zápis úspornější než výsledný JavaScript, jde opět o další technologii podporující koncept napiš méně, vytvoř více.

SASS
SASS (Syntactically Awesome Stylesheets) poskytuje podobnou funkcionalitu jako CoffeeScript, ale SASS ji poskytuje pro CSS kaskádové styly. Jde tedy o jazyk, který zjednodušuje zápis kaskádových stylů a po jeho kompilaci vznikne CSS soubor obsahující všechny definované styly. Implementace SASS je opensource a používá jazyk Ruby.

Bootstrap
Bootstrap je CSS3 framework, který vytvořil Twitter a vydal ho pod svobodnou licencí. Tento framework se do Rails přidává balíčkem `bootstrap-sass` a po jeho načtení do základního CSS a JavaScript souboru v adresáři app/view/assets můžeme bootstrap začít používat. Ve frameworku jsou předvytvořené kaskádové styly, které lze pak v .scss SASS souborech upravit podle konkrétní potřeby. Framework obsahuje přednastavené styly pro navigační panel, tlačítka, tabulky, rozvržení, patičku, seznamy a spoustu dalšího. Navíc je ve své kompletní seznam podporovaných atributů lze najít na oficiální stránce projektu Bootstrap [getbootstrap.com, 2014].

TTD
TTD je zkratka pro Test Driven Development, což je nový přístup k vývoji softwaru. V praxi to znamená, že programátor nejdříve napíše test, což není nic jiného než automatizovaný test napsaný nejčastěji ve stejném programovacím jazyku jako vyvíjená aplikace, a až poté naprogramuje žádanou funkcionalitu. Výhodou tohoto přístupu je, že jakmile programátor implementuje novou funkcionalitu, může hned přejít k testování. Jelikož má

test hotový, tak může programátor hned ověřit, zda je jeho implementace funkční. V Ruby on Rails tento přístup plně podporuje a dokonce příkaz `scaffold` automaticky vytvoří testy při generovaní nové struktury. TTD přístup při vývoji Rails aplikace využívá i velmi pěkně zpracovaný tutorial Michaela Hartla [HARTL, 2013].

4.3 UML

Jazyk UML (Unified Modeling System), jak uvádí Martin Pavus [PAVUS, 2006], je univerzální jazyk pro modelování, specifikaci a dokumentaci systémů. Nejčastěji je použit k modelování objektově orientovaných systémů, ale lze ho použít i na modelování ostatních systémů. Jazyk UML není vázán na žádnou metodiku ani programovací jazyk a ani neurčuje způsob, jakým máme systém modelovat. Poskytuje pouze prostředky jak vizuálně vyjádřit systém a jeho vazby. Historie jazyka UML se začala psát v 90. letech minulého století, kdy se na trhu objevilo několik autorů technik objektového modelování a také několik rozdílných metodik. Jazyk UML vznikl spojením tří nejpoužívanějších a v roce 1997 se jazyku ujala standardizační nezisková organizace OMG (Object Management Group), která přijala UML jako mezinárodní standard pro modelování systémů. Jazyk UML umí popsat jak statické vlastnosti systémů, tedy jejich strukturu a vazby mezi jednotlivými objekty. Tak umí popsat i dynamickou stránku systému, tedy vývoj objektů v čase. Tedy jak se mění jednotlivé objekty modelu vyvíjí v čase.

4.3.1 Class diagram

Pokud chceme vizualizovat jednotlivé třídy, tak existuje v UML pro tyto účely class diagram neboli diagram tříd. Tento diagram popisuje statickou strukturu systému, tedy třídy a vazby mezi nimi. U tříd se popisují jednotlivé atributy a metody. Vazby pak mohou mít podobu asociace, agregace, kompozice a generalizace [PAVUS, 2006].

4.4 Složitost softwaru

Jak uvádí profesor Vaníček ve svých skriptech „Měření a Hodnocení jakosti informačních systémů“ [VANÍČEK, 2004, s. 193], má pojem složitost software více významů. Jedním z nich je výpočetní složitost, kterou se v této kapitole zabývat nebudeme. Druhým je složitost projekční nebo též intelektuální, tedy celkové vynaložené usilí pro návrh, implementaci a provoz softwaru. Měření složitosti softwaru je možné provést ve více fázích. Je možné měřit složitost ve fázi specifikace, což s ohledem na dostupné, nebo spíše nedostupné podklady bývá nepřesné. Vyšší přednosti měření dosáhneme, pokud máme dostupné všechny potřebné podklady. To ovšem bývá často až ve fázi implementace, protože ve fázi testování se ještě mohou objevit nějaké změny. Ve fázi

implementace se již žádné změny nepředpokládají a proto můžeme využít metod, které pracují na základě zdrojového kódu měřeného softwaru.

4.4.1 LOC

Touto metodou se zjišťuje počet zdrojových řádků kódu a jak zmiňuje profesor Vaníček ve svých skriptech [Vaníček, 2004, s. 250], jde o nejjednodušší míru používající zdrojový kód. V této metodě se sčítají řádky zdrojového kódu s vynecháním komentářů a prázdných řádků. Jelikož jde o velmi jednoduchou metodu, má řadu z toho plynoucích nevýhod [Vaníček, 2004, s. 251] a to především:

- Míra LOC závisí velmi na použitém programovacím jazyce
- Počet řádků zdrojového kódu závisí na použitém programovacím stylu
- Počet řádků závisí na stylu zápisu programu

Jak konstatuje profesor Pavlíček [PAVLÍČEK, 2004, s. 251], tak i přes tyto nevýhody lze míru LOC považovat za dobrý indikátor pracnosti zejména pokud jde o porovnávání projektů podobného či stejného typu.

4.4.2 Metoda funkčních jednic

U metody funkčních jednic (function point analysis) nejde, jak zmiňuje profesor Pavlíček ve svých skriptech [Pavlíček, 2004, s. 218], o jedinou metodu, ale o celý soubor metod, které se navzájem velmi liší. Pro účely této diplomové práce byla vybrána metoda „Back – fire“, která umožňuje jednoduchým způsobem převést LOC na funkční jednice (Function Points) za pomoci průměrných koeficientů pro jednotlivé programovací jazyky. Tyto průměrné koeficienty jsou odvozené z výsledků reálných výpočtů FP velkého množství projektů. Autorem metody „Back – fire“ je Casper Jones, který se měření složitosti software věnuje od roku 1984.

Funkční jednice (FP, Function Point) je univerzální jednotka užívána pro měření složitosti softwaru. Pomocí této jednotky lze porovnat složitost dvou a více řešení vytvořených pomocí jiných programovacích jazyků.

Výhodou metody „Back – fire“ je [JONES, 2011]:

- Rychlost a jednoduchost výpočtu při zachování stále dobré porovnávatelnosti.
- Použitelná na malé aplikace a systémy, kterým končí životnost.

Nevýhodou je naopak [JONES, 2011]:

- menší přesnost oproti standardizované IFPUG metodě funkčních jednic. Ovšem u metody IFPUG velmi záleží na jejím přesném provedení certifikovaným analytikem.

- Použitelná pouze na hotový zdrojový kód

:

Jazyk	LOC / FP
PHP	53.33
Ruby	45.71
Smalltakl	21.33
Objective-C	26.67

Tab. 2 Koeficienty pro jednotlivé jazyky podle [JONES, 2013]

Pro výpočet je použitý následující vzorec, kde po dosazení výpočtené míry LOC a koeficientu jazyka z tabulky 2 dostaneme výsledný počet funkčních jednic.

$$FP = \frac{LOC}{\text{koeficient jazyka}}$$

II. Projekt

5 Vlastní řešení

V této kapitole bude postupně přestaveno původní řešení webové aplikace a poté vlastní řešení v Ruby on Rails. Vlastním řešením je webová aplikace - kniha jízd, která má za úkol spravovat údaje o firemních vozidlech a jejich tankování. Aplikace též obsahuje přehled o používání jednotlivých vozidel, tedy kdo řídil, kam se s vozidlem jelo a kdy. Jednou za měsíc je pak z vložených údajů vygenerován report nákladů za celý měsíc pro každé vozidlo zvlášť. Tento report po vytisknutí dále zpracovává kontrolní oddělení a poté je report uložen do archivu.

Funkční požadavky jsou tedy následující:

- Správa vozového parku
- Správa tankování pro jednotlivá vozidla
- Evidence jednotlivých jízd u vozidel
- Generování měsíčních reportů pro jednotlivá vozidla

Původní aplikace Kniha jízd je napsána v PHP4 a je součástí podnikového Intranetu, který je též napsaný v PHP4.

Technické požadavky na novou verzi Knihy jízd jsou následující:

- Přehledný a lehce spravovatelný kód
- Dobrá rozšiřitelnost o nové funkce
- PHP5 či obdobný skriptovací jazyk
- Modernější vzhled

S ohledem na požadavky byl vybrán framework Ruby on Rails jako platforma pro novou verzi aplikace. Použitá verze Rails je 3.2.17. Rails 4.0 po prvotních testech neobstály z důvodu chybějící podpory některých `gems` a naopak výhodou starší řady Rails by měla být celková odladěnost a stabilita.

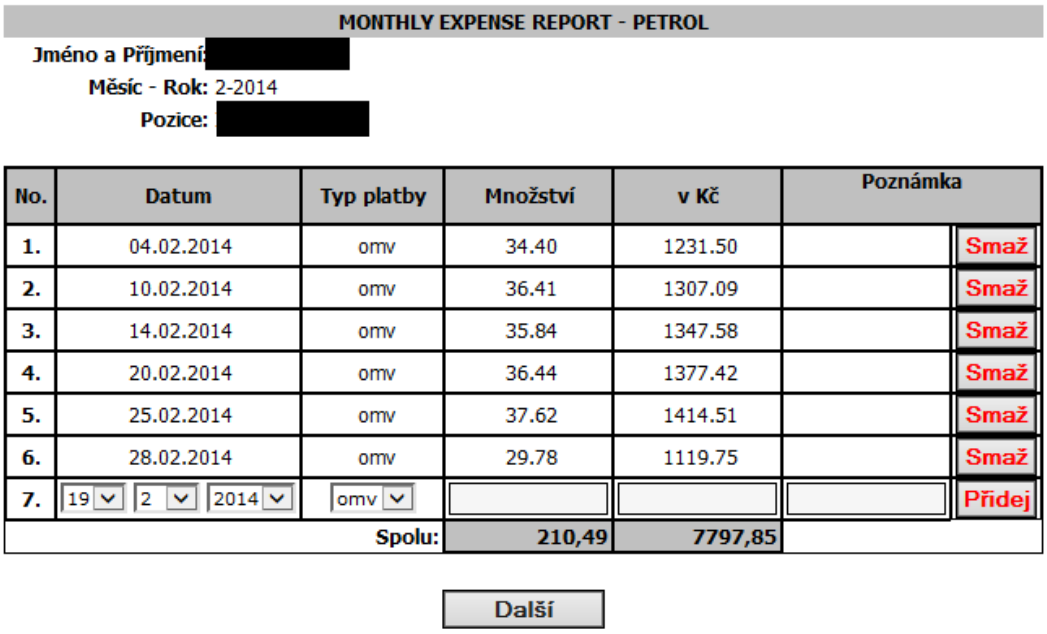
5.1 Kniha jízd – původní verze

Původní verze je obsažená v několika PHP souborech, které jsou navzájem prolinkované a pomocí POST / GET parametrů si předávají jednotlivé argumenty.

Původní řešení se skládá z následujících souborů:

- car.php
- car_php.php
- car_zahr.php
- car_report.php

- cars.php
- save_other.php
- save_phm.php
- save_report.php
- set_car.php
- common.php
- template.php



MONTHLY EXPENSE REPORT - PETROL

Jméno a Příjmení: [redacted]

Měsíc - Rok: 2-2014

Pozice: [redacted]

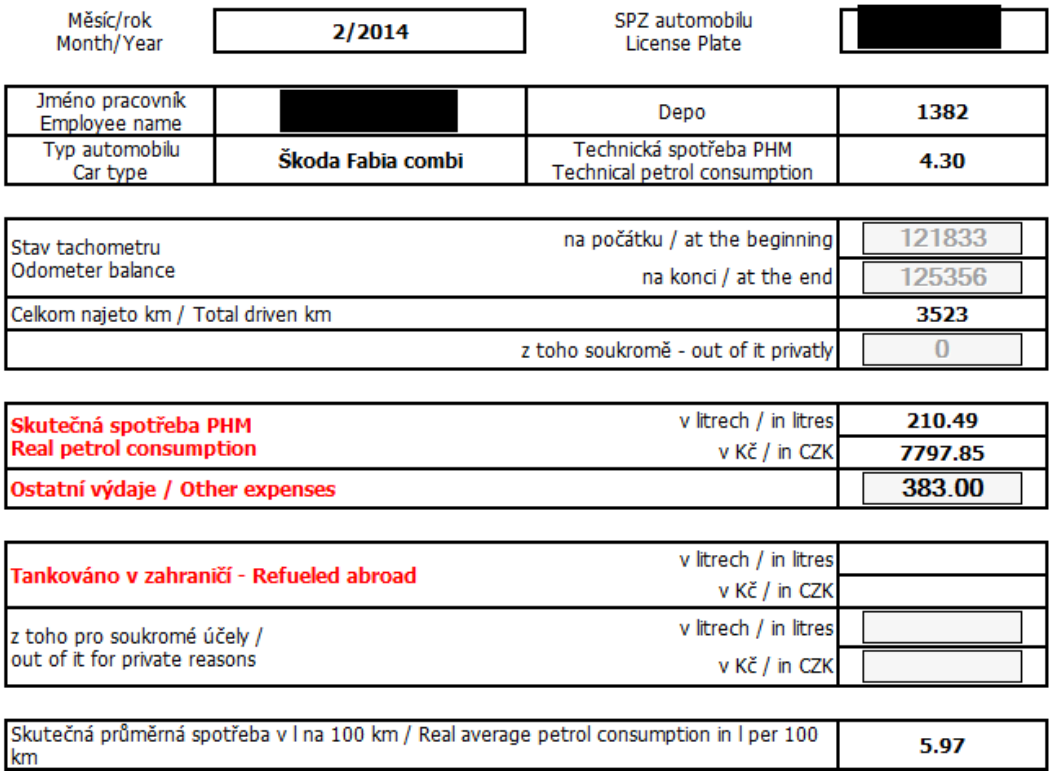
No.	Datum	Typ platby	Množství	v Kč	Poznámka	
1.	04.02.2014	omv	34.40	1231.50		Smaž
2.	10.02.2014	omv	36.41	1307.09		Smaž
3.	14.02.2014	omv	35.84	1347.58		Smaž
4.	20.02.2014	omv	36.44	1377.42		Smaž
5.	25.02.2014	omv	37.62	1414.51		Smaž
6.	28.02.2014	omv	29.78	1119.75		Smaž
7.	19 2 2014	omv				Přidej
		Spolu:	210,49	7797,85		

Další

Obr. 5 Původní PHP aplikace - tankování

Původní řešení má několik nevýhod:

- Značnou duplicitu kódu – odhadem okolo 30%
- Bussiness logika aplikace je umístěná spolu s kódem starající se o uživatelské rozhraní
- Velké množství SQL příkazů, které jsou účelově napsané, a při změně struktury databáze bude nutná jejich úprava
- Zastaralý vzhled
- Duplicita dat v databázi



Měsíc/rok Month/Year	2/2014	SPZ automobilu License Plate	[redacted]

Jméno pracovník Employee name	[redacted]	Depo	1382
Typ automobilu Car type	Škoda Fabia combi	Technická spotřeba PHM Technical petrol consumption	4.30

Stav tachometru Odometer balance	na počátku / at the beginning	121833
	na konci / at the end	125356
Celkom najeto km / Total driven km		3523
	z toho soukromě - out of it privatly	0

Skutečná spotřeba PHM Real petrol consumption	v litrech / in litres	210.49
	v Kč / in CZK	7797.85
Ostatní výdaje / Other expenses		383.00

Tankováno v zahraničí - Refueled abroad	v litrech / in litres	
	v Kč / in CZK	
z toho pro soukromé účely / out of it for private reasons	v litrech / in litres	
	v Kč / in CZK	

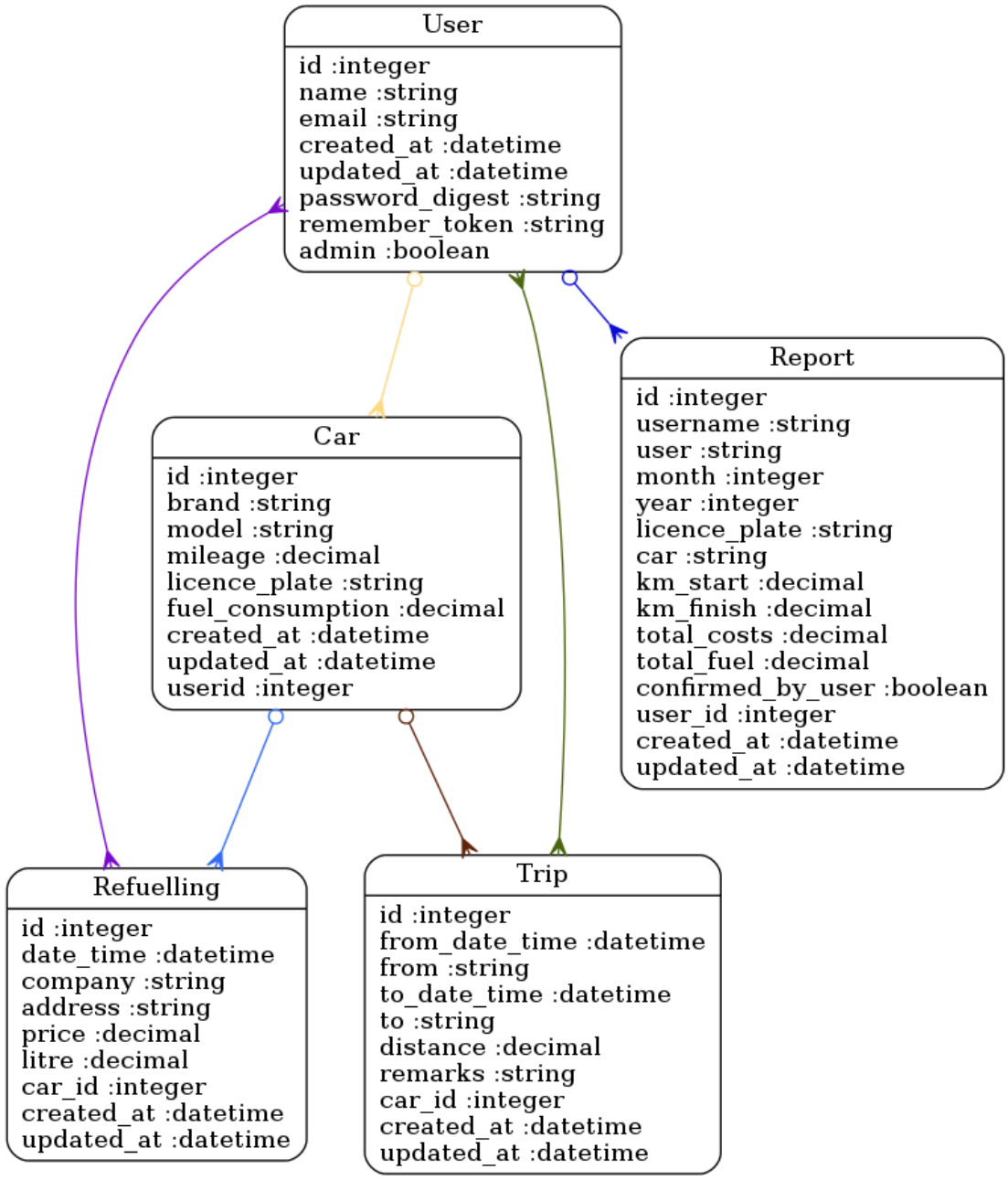
Skutečná průměrná spotřeba v l na 100 km / Real average petrol consumption in l per 100 km	5.97

Obr. 6 Původní PHP aplikace - report

5.2 Rails aplikace

Nové řešení v Rails se drží plně objektového návrhu, dokonce i přístup k datům v databázi využívá plně objektového návrhu pomocí „gem“ balíku Active Record.

Objektový model Rails aplikace



Obr. 7 Class diagram modelů Rails aplikace

5.2.1 Příprava prostředí pro Rails

V této kapitole bude představeno několik standardních postupů, které jsou základem tvorby jakékoliv Rails aplikace. Postup zde uvedený byl použit při tvorbě projektové aplikace popsané v předchozí kapitole.

5.2.1.1 Nastavení webového a databázového serveru

V Rails existují celkem tři oddělená prostředí – test, development a production. Jelikož naši aplikaci budeme i provozovat, je dobré se již od začátku zamyslet nad produkčním prostředí.

Pro naši aplikaci bylo zvoleno následující produkční prostředí, které není úplně standardní, ale i tak jde o prověřené řešení:

- Operační systém: GNU/Linux – distribuce Ubuntu 12.04
- Webový server: Nginx
- Aplikační server: Phusion Passanger ve formě modulu do Nginx
- Databázový server: PostgreSQL 9.1
- Ruby: 1.9.3
- Verzovací systém: Git

Instalací distribuce Ubuntu, webového a databázového serveru se tato diplomová práce zabývat nebude. Následuje pouze doporučení pro databázový server a krátká ukázka nastavení webového serveru. U databáze je vhodné vytvořit samostatné uživatele pro každou databázi zvlášť a jednotlivým uživatelům nastavit opravnění pouze na konkrétní databázi. Nastavení pro server Nginx bylo použito následující:

```
http {
...
        passenger_root /usr/local/lib/ruby/gems/1.9/gems/passenger;
        passenger_ruby /usr/local/bin/ruby;
...
        server {
            listen       80;
            server_name  rails.ffuu.eu;
            root         /var/www/rails/kniha/public;
            passenger_enabled on;
            rails_spawn_method smart;
            passenger_load_shell_envvars on;
            rails_env production;
            location = /50x.html {
                root   /var/www/nginx-dist;
            }
    }
}
```

V závislosti na konkrétním nastavení Linuxového serveru se výše uvedené cesty k souborům mohou lišit.

5.2.1.2 Instalace jazyka Ruby

Některé linuxové distribuce obsahují Ruby již v základní instalaci, jako například SUSE, které má nově do Ruby přepsaný svůj hlavní konfigurační nástroj Yast. Pro následující příklady je použitá distribuce Ubuntu Linux 12.04, kde lze Ruby doinstalovat následujícím příkazem:

apt-get install ruby

ovšem tato verze ruby je pouze verze 1.8.7 a to neodpovídá našemu navrhnutému prostředí, proto i z důvodu následné instalace webového frameworku Ruby on Rails nainstalujeme Ruby následujícím postupem [SVERDLOV, 2012]:

1. Provedeme aktualizaci listu dostupných instalačních balíčků

sudo apt-get update

2. Nainstalujeme curl

sudo apt-get install curl

3. Pomocí příkazu curl stáhneme RVM skript (#poznamka pod carou co je RVM) a nainstalujeme ho

\curl -L https://get.rvm.io | bash -s stable

4. Otevřeme novou konzolovou instaci či provedeme relogin a poté spustíme následující příkaz, který načte všechny základní proměnné rvm do proměnných našeho prostředí

source ~/.rvm/scripts/rvm

5. Pokud je nutné doinstalujeme některé chybějící závislosti rvm, nejspíše bude nutné též zadat heslo k účtu uživatele root

rvm requirements

6. Jakmile má rvm všechny komponenty potřebné ke svému spuštění je instalace ruby již jednoduchá

rvm install 1.9.3

7. Posledním krokem je pak instalace gems, což je obdoba CPANu u Perlu, tedy balíčkovací systém Ruby modulů

rvm rubygems current

5.2.1.3 Instalace Ruby on Rails

Jelikož již máme nainstalovaný interpret jazyka Ruby, můžeme pokročit k vlastní instalaci Ruby on Rails.

1. Nejprve je nutné se ujistit zda máme nainstalovanou poslední verzi ruby gems (pokud je již instalace provedena v průběhu instalace ruby, lze tento krok přeskočit)

   rvm rubygems current
2. Za pomoci příkazu gems nainstalujeme Ruby on Rails

   gem install rails

Spolu s Ruby on Rails se pomocí příkazu „gem“ nainstalovali i všechny závislé „gems“ a lze pokročit vytvoření samotné Rails aplikace.

5.2.2 Vytvoření rails aplikace

Pokud chceme začít programovat v Ruby on Rails, musíme nejdříve vytvořit kostru Rails aplikace. Spouštením následujícího příkazu se vytvoří kompletní adresářová struktura. Spolu s adresářovou strukturou se také se vygenerují všechny základní soubory včetně konfiguračních.

```
rails new kniha
      create
      …
      create  Gemfile
      create  app
      create  app/assets/images/rails.png
      …
      create  app/controllers/application_controller.rb
      create  app/helpers/application_helper.rb
      create  app/models
      create  app/views/layouts/application.html.erb
```

```
      …
      create  config
      create  config/routes.rb
      create  config/application.rb
      create  config/environment.rb
      …
      create  vendor/plugins
      create  vendor/plugins/.gitkeep
         run  bundle install
Fetching source index for https://rubygems.org/
…
Your bundle is complete! Use `bundle show [gemname]` to
see where a bundled
gem is installed.
```

Jako druhý krok je potřeba nastavit pár základních věcí.

5.2.2.1 Nastavení databáze

Databáze se nastavuje v souboru `config/database.yml` v adresáři Rails aplikace. V základu je vše nastaveno na použití souborové SQL databáze SQLite, která stačí na vývojovou část. Ale jelikož máme přístup k serveru s plnohodnotnout SQL databází je nutné nastavení změnit na PostgreSQL. Následující příklad je pro databázi „`development`“, ale obdobný postup je použit na i databáze „test“ a „production“:

```
development:
  adapter: postgresql
  encoding: unicode
  database: kniha_development
  pool: 5
  username: rails_test
  password: <heslo>
  host: localhost
```

5.2.2.2 Další nastavení

V aplikaci jsou použité i některé nestandardní balíky, a proto je vhodné je v tuto chvíli nainstalovat. Instalaci provedeme pomocí příkazu `bundle install`. Poté již je aplikace připravena k použití. Ovšem doporučuji z důvodu ochrany uživatelských hesel vynutit SSL šifrování. To se nastaví v souboru `config/environments/production.rb` nastavením:

```
config.force_ssl = true
```

5.2.3 Scaffolding

Nyní již můžeme pokročit k vlastnímu programování. Protože rozsah této diplomové práce nedovoluje vše krok po kroku popsat byl vybrán následující komplexní příklad jak práce na Rails aplikaci vypadá.

Následující ukázka je scaffolding modelu Jízd, pro které byl zvolen název Trip. Příkaz provede kompletní vygenerování kostry modelu `Trip` na základě zadaných parametrů:

.

```
rails generate scaffold Trip from_date_time:datetime from:string
to_date_time:datetime to:string distance:decimal remarks:string
car_id:integer --no-stylesheets
```

Po spuštění vypíše příkaz podobný výstup:

```
invoke  active_record
create    db/migrate/20140212002009_create_trips.rb
create    app/models/trip.rb
invoke    rspec
create      spec/models/trip_spec.rb
invoke  resource_route
 route    resources :trips
invoke  scaffold_controller
create    app/controllers/trips_controller.rb
invoke    erb
create      app/views/trips
create      app/views/trips/index.html.erb
create      app/views/trips/edit.html.erb
create      app/views/trips/show.html.erb
create      app/views/trips/new.html.erb
create      app/views/trips/_form.html.erb
invoke    rspec
create      spec/controllers/trips_controller_spec.rb
create      spec/views/trips/edit.html.erb_spec.rb
create      spec/views/trips/index.html.erb_spec.rb
create      spec/views/trips/new.html.erb_spec.rb
create      spec/views/trips/show.html.erb_spec.rb
create      spec/routing/trips_routing_spec.rb
invoke      rspec
```

```
      create        spec/requests/trips_spec.rb
      invoke    helper
      create      app/helpers/trips_helper.rb
      invoke      rspec
      create        spec/helpers/trips_helper_spec.rb
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/trips.js.coffee
```

Jak je z výstupu poznat příkaz scaffold právě vytvořil všechny výše uvedené soubory. Poté je potřeba vytvořit v databázi požadavanou tabulku, do které budeme ukládat jednotlivé záznamy. „Migrate“ soubor obsahuje následující záznamy:

```
cat db/migrate/20140212002009_create_trips.rb
```

```
class CreateTrips < ActiveRecord::Migration
  def change
    create_table :trips do |t|
      t.datetime :from_date_time
      t.string :from
      t.datetime :to_date_time
      t.string :to
      t.decimal :distance
      t.string :remarks
      t.integer :car_id

      t.timestamps
    end
  end
end
```

A provedeme ho následujícím příkazem:

```
bundle exec rake db:migrate
```

Po otestování zda migrace proběhne v pořádku na databázi development a po zkontrolování všech údajů můžeme tabulku nainstalovat i na produkční prostředí:

```
bundle exec rake db:migrate:up RAILS_ENV=production
```

Výstup:

```
==                              CreateTrips:              migrating
=================================================
-- create_table(:trips)
NOTICE:      CREATE   TABLE   will   create   implicit   sequence
"trips_id_seq" for serial column "trips.id"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index
"trips_pkey" for table "trips"
   -> 0.2260s
==                  CreateTrips:         migrated         (0.2263s)
========================================
```

Z výše uvedeného výstupu je vidět, že byl automaticky přidán sloupec s názvem id, byla k němu vytvořena sekvence pro generování jednotlivých id a pro tento sloupec v tabulce trips byl též vytvořen index.

A nyní lze zkontrolovat výsledek pomocí URL

```
https://<url_aplikace>/trips
```

Pravděpodobně tento postup skončí zobrazením chybové hlášky. Je to z důvodu, že chybí záznam v souboru routes.rb pro třídu `Trip`. Přidáme tedy do souboru `routes.rb` následující řádek:

```
resources :trips
```

který vytvoří všechny dostupné cesty (URL) k nově vytvořenému modelu. V závislosti na použitém prostředí je vhodné aplikaci zrestartovat a to provede příkaz:

```
touch tmp/restart.txt
```

Pokud stránku obnovíme, tak by nyní měl prohlížec vykreslit základní strukturu stránky `https://<url_aplikace>/trips` se kterou můžeme dále pracovat. Výhodou scaffolding je především v tom, že máme model již plně funkční. Lze tedy přidávat jednotlivé jízdy, editovat je, mazat a tak dále.

Jelikož dle našeho modelu (tříd) má třída `Trip` závislost na třídě `Car`, je vhodné do souboru `app/models/car.rb` přidat následující podmínku:

```
has_many :trips, dependent: :destroy
```

Tato podmínka vytvoří metody pro daný zdroj tak, že můžeme skrze objekt auto přistoupit I k jízdám, které k danému vozidlu přísluší.

Podobnou vazbu je nutné přidat též do souboru `app/models/trip.rb`

```
belongs_to :car
```

Jelikož tento model je neobsahuje zatím žádné validace vstupu, je dobré je nastavit co nejdříve.

```
validates :car_id, presence: true
```

A poté již můžeme přidat odkaz na nově vytvořenou třídy na hlavní stránku aplikace do souboru `app/views/layouts/_header.html.erb`:

```
<li><%= link_to "Jízdy", trips_path %></li>
```

Jelikož základní vzhled není graficky nijak propracovaný, v následující kapitole bude ukázáno, jak v Rails aplikaci použít Twitter Bootstrap CSS3 framework.

5.2.4 Bootstrap CSS3 framework

Bootstrap je Ruby „gems“ balíček, který obsahuje CSS3 framework vytvořený společností Twitter. Framework je vydaný pod svobodnou open source licencí, ovšem existuje mnoho firem, které komerčně nabízejí CSS3 templaty především pro starší Bootstrap 2. V naší aplikaci budeme používat Boostrap řady 3, který jsme přidali příkazem `bundle install` v kapitole 4.2.3.

Bootstrap framework se hodí pro lepší vzhled `scaffolding` modelů. Zde je ukázka SASS kódu stylizující logo aplikace:

```
/* header */

#logo {
  float: left;
  margin-right: 10px;
  font-size: 1.7em;
  color: white;
  text-transform: uppercase;
  letter-spacing: -1px;
```

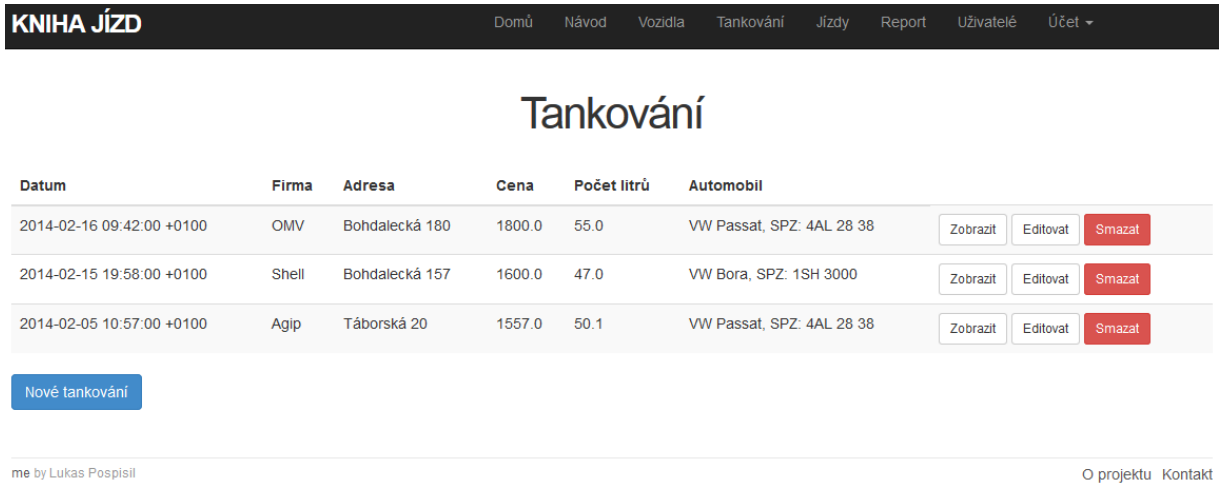
```
    font-weight: bold;
    padding-top: 15px;
    line-height: 1;
    &:hover {
      color: white;
      text-decoration: none;
    }
  }
```

Jinak ale Bootstrap obsahuje velkou řadu předpřipravených CSS3 objektů, například tlačítka:

```
<%= link_to 'Zobrazit', refuelling, :class => 'btn btn-sm btn-default' %>
```

Kterými lze vzhled Rails aplikace dále vylepšit. Kompletní přehled obsažených CSS3 objektů lze najít na oficiálním webu Bootstrapu [GETBOOSTRAP, 2014].

Výsledná aplikace pak vypadá takto:



Obr. 8 Výsledná Rails aplikace za použití Bootstrapu

5.3 Výpočet složitosti obou řešení

Na základě postupu popsaného v kapitole 4.3 provedeme nejprve výpočet LOC, tedy řádků programového kódu pro původní řešení a nové řešení. Poté následuje výpočet funkčních jednic metodou backfiring popsanou v kapitole 4.3.2.

5.3.1 Výpočet LOC

U metody LOC se sčítají pouze řádky obsahující zdrojový kód bez komentářů a prázdných řádků. Pro původní řešení byl výpočet proveden ručním sčítáním. U řešení v Ruby on Rails byl použit příkaz `rails stats` pro automatizovaný výpočet LOC.

5.3.1.1 Původní řešení v PHP

Původní řešení je složené z několika navzájem navazujících php souborů, následující tabulka zahrnuje počty LOC pro tyto jednotlivé soubory.

Soubor	počet LOC
car.php	330
car_php.php	198
car_zahr.php	212
car_report.php	154
cars.php	145
save_other.php	98
save_phm.php	104
save_report.php	167
set_car.php	56
common.php	50
template.php	35
Celkem	**1549**

Tab. 3 Výpočet LOC pro původní řešení v PHP

Po sečtení LOC za jednotlivé soubory dostaneme celkový výsledek LOC pro původní řešení:
LOC = 1549

5.3.1.2 Rails řešení

Použitý příkaz rails stats vrací výsledné LOC odděleně pro jednotlivé součásti aplikace. Jelikož Test LOC jsou řádky kódu používané v TTD popsané v kapitole 4.2.3.6, nebude jejich počet do výsledku zahrnut. Jako výsledek použijeme tedy pouze hodnotu Code LOC, což je počet řádků zdrojového kódu.

Name	Lines	LOC	Classes	Methods	M/C	LOC/M
Controllers	507	366	6	47	7	5
Helpers	110	87	0	17	0	3
Models	105	47	5	2	0	21
Libraries	0	0	0	0	0	0
Integration tests	0	0	0	0	0	0
Functional tests	98	78	2	0	0	0
Unit tests	70	15	5	0	0	0
Model specs	10	8	0	0	0	0
View specs	277	249	0	0	0	0
Controller specs	328	224	0	4	0	54
Helper specs	30	8	0	0	0	0
Routing specs	70	52	0	0	0	0
Request specs	22	18	0	0	0	0
Total	1627	1152	18	70	3	14

Code LOC: 500 Test LOC: 652 Code to Test Ratio: 1:1.3

Obr. 9 Výsledný počet LOC pro Rails řešení

Výsledek:
LOC = 500

5.3.2 Výpočet FP

Vypočet FP metodou backfiring popsané v kapitole 4.4.2 probíhá dosazením do vzorce:

$$FP = \frac{LOC}{\text{koeficient jazyka}}$$

Míru LOC máme vypočítanou z předchozí kapitoly a koeficient jazyka získáme z tabulky uvedené v kapitole 4.4.2.

5.3.2.1 Výpočet FP pro původní řešení

Výpočet provedeme dosazením do výše uvedeného vzorce:

$$FP = \frac{1549}{53.33}$$

Výsledek:
FP = 29

5.3.2.2 Výpočet FP pro řešení v Rails

Výpočet u Rails provedeme opět dosazením do vzorce:

$$FP = \frac{500}{45,71}$$

Výsledek:
FP = 11

6 Zhodnocení výsledků a doporučení

Ná základě výsledků z kapitoly 5.3.. lze konstatovat, že výsledné řešení webové aplikace v Ruby on Rails je výrazně úspornější, co do počtu řádku kódu. I když výsledek u míry LOC velmi závisí na použitém programovacím jazyku a stylu, je zřejmé, že naprogramovat 1549 řádků kódu u původního PHP řešení oproti 500 řádkům kódu u řešení v Ruby on Rails je výrazný rozdíl. Zde nepochybně pomohla snaha Rails o co nejmenší duplicitu kódu, kterým původní PHP řešení trpí. Dle méno názoru k menšímu počtu řádků kódu přispěl i plně objektový návrh, který je použit i pro přístup do databáze. Tento přístup zjednodušuje přístup ke datům v databázi a odbourává tak zcela nutnost použít SQL pro základní úkony. Příkazů v jazyce SQL naopak PHP řešení hojně využívá a to sebou přináší další duplicitu kódu.

Pokud porovnáme výsledky metody funkčních jednic, zde došlo k překvapení. Ač jsou obě řešení funkcionalitou téměř stejná, tedy v ideálním případě by měl být počet funkčních jednic stejný, tak se obě řešení výsledkově velmi liší. Pokud budeme brát v úvahu vyšší nepřesnost výpočtu FP danou použitou metodou, i tak je rozdíl velmi výrazný. Pokud se podíváme do tabulky uvedené v kapitole 4.4, je zde pro Ruby uveden koeficient na úrovni jazyků PHP a Pythonu. Ruby je přitom plně objektový na úrovni jazyka Smalltalk a Objective-C, tedy koeficient by měl tomu také odpovídat. Metoda Backfiring ač doporučována pro výpočet FP u menších projektů se tak ukázala jako nespolehlivá s ohledem na použitý koeficient. Pokud by byl koeficient jazyka Ruby mezi 21-26, tedy v rozmezí koeficientů jazyka Smalltalk a Objective-C vyšel by daleko přesnější výsledek. Navíc pokud by původní PHP řešení neobsahovalo žádný duplicitní kód, byla by obě řešení zcela porovnatelná.

Ve výsledku ale řešení v Ruby on Rails prokázalo, že má nižší složitost a tedy i pracnost. Předpoklad daný v úvodu této práce byl pravdivý.

7 Závěr

Na základě položených hypotéz v úvodní kapitole bylo prokázáno, že nové řešení webové aplikace v Ruby on Rails poskytuje celou řadu výhod oproti staršímu, neobjektovému, řešení. Programovací jazyk Ruby, na kterém staví framework Ruby on Rails, je jen další z řady výhod výsledného řešení.

V praktické části této diplomové práce byl proveden výpočet softwarové složitosti obou řešení, který objektivně odpověděl na otázky vznesené v úvodní kapitole. U nového řešení klesl počet řádků kódu z více než 1500 na 500 při zachování obdobné funkcionality a u metody funkčních jednic se počet snížil z 29 funčních jednich na výsledných 10 u nového řešení. Výsledné řešení je tedy méně složité než původní řešení a zároveň díky menšímu počtu řádků zdrojového kódu také jednodušší na správu, provoz a ladění chyb. Předpoklad daný v úvodu této práce tak byl splněn a s ohledem na výsledky i kladně zodpovězen. Výsledné řešení má menší softwarovou složitost.

Na základě získaných poznatků je vidět, že jazyk Ruby a webový framework Ruby on Rails se navzájem dobře doplňují a tvorba aplikací v Rails je nejen zábavná, ale i produktivní. Tím, že framework Ruby on Rails obsahuje velké množství moderních technologií, zaručuje nové aplikaci dlouhou životnost, protože nové řešení tak rychle nezastará a již nyní využívá technologií jako HTML5 a CSS3 či plně objektového přístupu k relační databázi. Výslednou aplikaci bude možné v průběhu let jednodušeji rozšiřovat o nové funkce a vlastnosti než jak tomu bylo u původní aplikace. To je dáno zejména modulární strukturou Rails frameworku. Již nyní je výsledné řešení připravené na implementaci AJAX či je možné implementovat vcelku jednoduše API pro přístup k Rails z mobilní aplikace.

8 Použitá literatura

FLANAGAN David, MATSUMOTO Y., 2008, *The Ruby Programming Language*, Sebastopol USA, O`Reilly, 444 s., ISBN 0-596-51617-7

RUBY S., THOMAS D., HANSSON D. H., 2011, *Agile Web Development with Rails, 4th Edition, Texas USA*, The Pragmatic Programmers, 527 s., ISBN 978-1934356548

COOPER P., 2009, *Beginning Ruby – From Novice to Professional*, Berkley USA, Apress 658 s., ISBN 978-1430223634

VANÍČEK Jiří, 2004, *Měření a hodnocení jakosti informačních systémů*, ČZU v Praze, Provozně ekonomická fakulta, 328 s., ISBN 80-213-1206-8

JONES Casper, 2013, *Function Point as a Universal Software Metric*, http://namcookanalytics.com/wp-content/uploads/2013/07/Function-Points-as-a-Universal-Software-Metric2013.pdf

JONES Casper, 2011, *Using Function Point Metrics for Software Economic Studies*, www.mstb.org/softec/download/SOFTEC2011_Speech5FuncPtEcon.ppt

GUNNER TECHNOLOGY, 2001, *The history of Ruby*, http://www.gunnertech.com/2011/11/the-history-of-ruby/

MOLIČ Jan, 2005, Ruby on Rails, Root.cz,: http://www.root.cz/serialy/ruby-on-rails/#ic=serial-box&icc=more

ŠRÁMEK Dalibor, 2002, *Ruby z rychlíku*, Root.cz, http://www.root.cz/serialy/ruby-z-rychliku/#ic=serial-box&icc=title

ŠRÁMEK Dalibor, 2002, *Ruby a OOP*, Root.cz, http://www.root.cz/serialy/ruby-a-oop/#ic=serial-box&icc=title

STRUGLING WITH RUBY, 2010, *struglingwithruby.blogspot.dk*, http://strugglingwithruby.blogspot.dk/2010/03/variables.html

BARD Adam, 2013, *Top Github Languages for 2013 (so far),* http://adambard.com/blog/top-github-languages-for-2013-so-far/

ZI HENG ZHOU, 2012, *An interview with Yukihiro „Matz" Matsumoto*, http://fredwu.me/post/36493181321/an-interview-with-yukihiro-matz-matsumoto

STEWART BRUCE, 2001, *An Interview with the Creator of Ruby*, http://www.linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html

LAI HONGLI, 2014, *Phusion Passenger now supports the new Ruby 2.1 Out-Of-Band GC*, http://blog.phusion.nl/2014/01/31/phusion-passenger-now-supports-the-new-ruby-2-1-out-of-band-gc/

HEMPEL BRIAN, 2014, *Is Ruby fast yet?*, http://www.isrubyfastyet.com/

MOSES John, 2013, *Is Ruby dying?*, http://jmoses.co/2013/12/21/is-ruby-dying.html

IRETON Doug, 2013, *Ruby -p -i -e*, http://dougireton.com/blog/2013/03/24/ruby-p-i-e/

CARRIERE Mathew, 2008, *Using select, reject, collect, inject and detect*, http://matthewcarriere.com/2008/06/23/using-select-reject-collect-inject-and-detect/

TALIM Satish, 2006, *Ruby Access Control*, http://rubylearning.com/satishtalim/ruby_access_control.html

TALIM Satish, 2006, *Modules Mixins*, http://rubylearning.com/satishtalim/modules_mixins.html

HANSSON David, 2014, *Rails 4.1.0: Release candidate 1*, http://weblog.rubyonrails.org/2014/2/18/Rails-4-1-rc1/

BASU Sayanee, 2013, *Ruby on Rails Study Guide: The History of Rails*, https://code.tutsplus.com/articles/ruby-on-rails-study-guide-the-history-of-rails--net-29439

BERNARD BOREK, 2009, *Úvod do architektury MVC*, http://www.zdrojak.cz/clanky/uvod-do-architektury-mvc/

HARTL Michael, 2013, *Ruby on Rails Tutorial - Learn Web Development with Rails*, http://ruby.railstutorial.org/ruby-on-rails-tutorial-book?version=3.2

MEJIA Adrian, 2011, *Ruby on Rails Architectural Design*, http://adrianmejia.com/blog/2011/08/11/ruby-on-rails-architectural-design

MANCINI Stefano, 2013, *The 15 best gems for ruby on rails web applications*, http://www.devinterface.com/blog/en/2013/12/le-15-migliori-gemme-per-web-applications-ruby-on-rails/

HAINING Ryan, 2013, *Ruby super keyword*, http://stackoverflow.com/questions/2597643/ruby-super-keyword

AZAD Kalid, 2007, *Intermediate Rails: Understanding Models, Views and Controllers*,http://betterexplained.com/articles/intermediate-rails-understanding-models-views-and-controllers/

PAVUS Martin, 2006, *Class diagram – diagram tříd*,

http://mpavus.wz.cz/uml/uml-s-class-3-3-1.php

TUTORIALSPOINT, 2014, *Ruby if...else, case, unless*,

http://www.tutorialspoint.com/ruby/ruby_if_else.htm

BRITT James, 2014, *Object*, http://ruby-

doc.org/docs/keywords/1.9/Object.html

CHEDDA Frederick, 2012, *Polymorphism in Ruby*, http://www.runtime-

era.com/2012/08/polymorphism-in-ruby.html

SVERDLOV Etel, 2012, *How To Install Ruby on Rails on Ubuntu 12.04 LTS*

(Precise Pangolin) with RVM,

https://www.digitalocean.com/community/articles/how-to-install-ruby-on-rails-on-ubuntu-12-04-lts-precise-pangolin-with-rvm

GETBOOSTRAP, 2014, Bootstrap - official site, http://getbootstrap.com/

III. Přílohy

9 Přílohy

- Příloha A: Zdrojové kódy Rails aplikace ve formě externího archivu
- Příloha B: Zdrojové kódy PHP aplikace ve formě externího archivu